LISA Reference Manual (WIP)

Laboratory for Automated Reasoning and Analysis
Swiss Federal Institute of Technology Lausanne

February 16, 2022

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.

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** 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:

$$
\begin{aligned}
\mathcal{T} := \; & \mathrm{Var}(\mathrm{Id}) \\
& \mid \mathrm{Fun}(\mathrm{Id}, \mathrm{Arity})(\mathrm{List}[\mathcal{T}]) \\
& \mid ?\mathrm{Sfun}(\mathrm{Id}, \mathrm{Arity})(\mathrm{List}[\mathcal{T}])
\end{aligned}
\tag{1.1}
$$

I.e. a term is either a variable, described by some identifier, or one of two kinds of functions. A function node is labelled by an identifier and an arity, and the list of term is called the "children" of the node. The number of children should always be equal to the arity. Function symbols of arity 0 are also called constants.

As the definition states, we have to kind of function symbols: Normal ones and Schematic ones denoted with an question mark. Those Schematic symbols stand between variables and normal functions: they can be substituted for other terms, giving some flavour of second order logic, but they can't be bound. Those schematic symbols are sometimes also called unknowns, and their main usages are to express axiom schemas and some meta-theorems.

An Id is a string, and an Arity is a positive integer, In Lisa, the (Id, Arity) part of a term is called a label and is encapsulated in its own structure. A variable label contains only an identifier, while schematic and non-schematic function labels contain an id and an arity.

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

$$\begin{aligned} \mathcal{F} := &\operatorname{Pred}(\mathrm{Id}, \mathrm{Arity})(\mathrm{List}[\mathcal{T}]) \\ &\mid ?\mathrm{Spred}(\mathrm{Id}, \mathrm{Arity})(\mathrm{List}[\mathcal{T}]) \\ &\mid \mathrm{Connector}(\mathrm{Id}, \mathrm{Arity})(\mathrm{List}[\mathcal{F}]) \\ &\mid \mathrm{Binder}(\mathrm{Id})(\mathrm{Var}(\mathrm{Id}), \mathcal{F}) \end{aligned} \tag{1.2}$$

A formula can be a predicate, normal or schematic, labelled by and Id and an Arity, with a list of terms as children. We call these formulas *Atomic*. A special predicate is the equality symbol, with Id "=" and arity 2. A formula can also be given by a logical connector and a set of children formulas. A contrario to predicate and function symbols, which can be freely created, there is only the following finite set of connector symbols in LISA:

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

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 binders.

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

In this document as well as in the code documentation, we generally write terms and formula in a more conventional way, generally hiding the arity of symbols. When the arity is relevant, we write it with an superscript, for example:

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

and

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

We also use other usual notations for propositional logic, such as infix with right associativity and usual precedence rules.

1.1.2 Operations

Important concepts that are defined along FOL are substitution (of variables) and instantiation (of schematic functions and predicates). Both are implemented in a capture-avoiding way:

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

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

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

if y is not free in ψ and

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

Where z is not free in r and ϕ otherwise.

This definition of substitution is justified by the notion of alpha equivalence: Two terms which are identical up to renaming of bound variables are considered equivalent.

Definition 4 (Instantiation)**.** What we call instantiation is some form of higher-order substitution for schematic functions and predicates. A schematic function can be instantiated by a "function-like" term, by which we mean a term with a distinguished set of variables that corresponds to the argument of the function. Instead of giving a convoluted definition, we show an example. Suppose a, b, x, z, y are variables and we have:

$$t :=?f(x, x + y)$$

$$s := \cos(a - z - b) * b$$

then:

$$t[?f/(s, (a, b))] = \cos(x - z - (x + y)) * (x + y)$$

$?f$ is replaced by r, and within r, the variables a and b are replaced by the arguments inside $?f$: x and $x + y$. If s contains a non-parametric free variable (in this example, z), then we need to avoid capture similarly as in substitution.

This language of first order logic is defined in LISA in the package lisa.kernel.fol.

1.1.3 The Equivalence Checker

While proving theorem, trivial syntactical transformations such as $p \land q \equiv q \land p$ 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. In particular, two formulas $p \land q$ and $q \land p$ 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 NP-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[?]. 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 α-equivalence, by which we mean renaming of bound variables.

L1:	$x \lor y = y \lor x$	L1':	$x \land y = y \land x$
L2:	$x \lor (y \lor z) = (x \lor y) \lor z$	L2':	$x \land (y \land z) = (x \land y) \land z$
L3:	$x \lor x = x$	L3':	$x \land x = x$
L4:	$x \lor 1 = 1$	L4':	$x \land 0 = 0$
L5:	$x \lor 0 = x$	L5':	$x \land 1 = x$
L6:	$\neg\neg x = x$	L6':	same as L6
L7:	$x \lor \neg x = 1$	L7':	$x \land \neg x = 0$
L8:	$\neg(x \lor y) = \neg x \land \neg y$	L8':	$\neg(x \land y) = \neg x \lor \neg y$
L9:	$x \implies y = \neg x \lor y$		
L10:	$x \leftrightarrow y = (\neg x \lor y) \land (\neg y \lor x)$		
L11:	$x\exists!x.P = \exists y.\forall x.(x = y) \leftrightarrow P$		

Table 1.1: Laws of an algebraic structures $(S, \land, \lor, 0, 1, \neg)$. LISA's equivalence-checking algorithm is complete (and log-linear time) with respect to laws L1-L8 and L1'-L8'.

1.2 Proofs in Sequent Calculus

1.2.1 Sequent Calculus

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

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

$$a_1, a_2, ..., a_m \vdash b_1, b_2, ..., b_n$$

The intended semantic of such a sequent is:

$$(a_1 \wedge a_2 \wedge ... \wedge a_m) \implies (b_1 \vee b_2 \vee ... \vee b_n) \tag{1.3}$$

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. (This is in particular needed to make double negation elimination.)

A deduction rule, also called a proof step, has (in general) between zero and two prerequisite sequent (premises) 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 and substitution of equal-for-equal and equivalent-for-equivalent in Figure 1.2.

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

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. Figure 1.4 shows an example of a proof tree for Pierce's Law in 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.

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

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

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

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

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

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

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

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

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

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

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

Figure 1.1: Basic set of deduction rules

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

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

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

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

Figure 1.2: Additional deduction rules to account for axioms and equality

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \texttt{Rewrite} \qquad \frac{\cdots \qquad \cdots \qquad \cdots}{\Gamma \vdash \Delta}\ \texttt{Subproof}$$

Figure 1.3: 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.

$$\begin{array}{l}
0\ \texttt{Hypothesis}\quad \phi \vdash \phi \\
1\ \texttt{RightWeakening}(0)\quad \phi \vdash \phi, \psi \\
2\ \texttt{RightImplies}(1)\quad \vdash \phi, (\phi \to \psi) \\
3\ \texttt{Hypothesis}\quad \phi \vdash \phi \\
4\ \texttt{LeftImplies}(2,3)\quad (\phi \to \psi) \to \phi \vdash \phi \\
5\ \texttt{RightImplies}(4)\quad \vdash ((\phi \to \psi) \to \phi) \to \phi
\end{array} \tag{1.4}$$

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

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 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 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.
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 are the same, up to some transformation. 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.

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 1.3, one should use the `Rewrite` step. The proof checking function will output a triple of type (Boolean, List[Int], String. The first element indicate if the proof is valid or erroneous. If it is valid, the second and thirs element will be respectively an empty list and an empty string. Otherwise, the list will indicate the position of the error in the proof. FOr example, the list $(2,7)$ indicates an error in the 2nd step of the proof (which will be a subproof), then in the 7th step of that subproof.

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 supposed 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 elements of set theory($\in$, $\emptyset$, $\bigcup$ and set theory axioms, see 2.1) or of any other theory.

To prove a sequent 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. The theory 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. 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.

The user can also introduce definitions in the theory. Extension by definition is the mechanism by which a theory can be augmented by some symbol and definition axiom without changing what is and isn't provable in the theory. Moreover, the theory keeps track of all symbols which have been defined so that it can detect and refuse conflicting definitions.

1.3.1 Definitions

LISA's kernel allows to define two kinds of objects: Function 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 (or variables). For example "f is derivable" cannot be stated about a function symbol. We will come back to this in section 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 **??** 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 complicated: To define a function f, one must first prove a statement of the form $\forall \vec{x} \exists ! y.\phi$. Then we obtain for free the property that $\forall \vec{x} \exists ! x.(f(\vec{x}) = y) \leftrightarrow \phi$. 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*. It is well known that it produces a conservative extension of the base theory.

Definition 6. 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 $\mathcal{T}_1 \vdash \phi$

For function definitions, it is common in logic textbooks to only require the existence of y and not its uniqueness. The property one 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. $\square$

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

Definition 7. 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. (See for example Duparc.)

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 only axioms of $\mathcal{T}_1$.*

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. □

In LISA, definitions are objects of one of two sorts: Predicate definitions and Function definitions

$$\begin{aligned} \mathcal{T} :=& PredDefinition(p : Label, \vec{x} : List[Var], \phi : Formula) \\ & \mid FunctionDefinition(f : Label, \vec{x} : List[Var], y : Var, \phi : Formula) \end{aligned} \tag{1.5}$$

They correspond to the following definitions:

$$\text{let } p^n(\vec{x}) := \phi_{\vec{x}}$$

and

$$\text{let } f(\vec{x}) \text{ be the unique element such that } \phi[f(\vec{x})/y]$$

Formally, those definitions provide the following statements as if they were axioms:

$$\vdash \forall \vec{x}.p(\vec{x}) \leftrightarrow \phi$$

$$\vdash \forall \vec{x} \forall y.(f(\vec{x}) = y) \leftrightarrow \phi$$

To obtain a predicate definition, the user simply has to provide the various elements $p, \vec{x}, \phi$. To obtain a function (or constant) definition however, the user must first prove the existence and uniqueness of an element satisfying ϕ. Said otherwise, it is necessary to prove that ϕ is *functional*. Formally, the proof must be of the form: $\vdash \forall \vec{x} \exists ! x.\phi$ or equivalently $\vdash \forall \vec{x} \exists y.\forall x.\phi \leftrightarrow (x = y)$

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

1.4 Kernel: Supplements

Printer The printer file that comes with the kernel is able to produce a representation of terms, formulas and proofs that is concise, readable and close to the usual writing. The proof pretty printer is taking an optional error argument. This argument should be the output of the proof checker. It will show the position of the error in the proof with a small arrow.

To avoid having too large representations of a proof, it is possible to hide subproofs in the representation of a proof by having the display argument put to false in the subproof object.

Chapter 2

Set Theory

2.1 Axioms of Set Theory

The classical set theory is called Zermelo-Frankel Set Theory, or simply ZF. It is made of the 7 axioms of Zermelo, which are sufficient to formalize a large portion of mathematics, plus the axiom of replacement of Frankel.

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

Z4 (union). $\forall x, z.(x \in \mathrm{U}(z)) \iff (\exists y.(x \in y) \wedge (y \in z))$

Z5 (power). $\forall x, y.(x \in \mathcal{P}(y)) \iff (x \subset y)$

Z6 (foundation). $\forall x.(x \neq \emptyset) \implies (\exists y.(y \in x) \wedge (\forall z. z \in x))$

Z7 (comprehension schema). $\forall z, \vec{v}. \exists y. \forall x.(x \in y) \iff ((x \in z) \wedge \phi(x, z, \vec{v}))$

Figure 2.1: 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 2.2: Axioms for Zermelo-Fraenkel set theory.