Lisa Manual

January 19, 2025

Laboratory for Automated Reasoning and Analysis

EPFL, Switzerland

Contributors:

Simon Guilloud, Sankalp Gambhir, Viktor Kunčak

Contents

Chapter 1

Starting with Proofs in Lisa

Lisa is a proof assistant. Proof assistants support development of formal proofs of mathematical statements.

The centerpiece of Lisa (called the *kernel*) contains a mechanized implementation of first order logic (FOL), a logical framework to write mathematical statements and their proofs. This kernel is what provides correctness guarantees to the user. The kernel only accepts a small set of formal deduction rule such as "if a is true and b is true then $a \wedge b$ is true". This is in contrast to human-written proofs, which may contain a wide variety of complex or implicit arguments. If a proof is accepted as being correct by the kernel, it is expected to meet objective criteria for valid proofs according to the field of formal mathematical logic.[1] Lisa's kernel is described in more detail in chapter 2.

Writing mathematical theories (for example, group theory, combinatorics, topology, theory of computation) directly from these primitive constructions would be tedious. Instead, we use them as building blocs that can be combined and automatized. Beyond the correctness guarantees of the kernel, Lisa's purpose is to provide tools to make writing formal proofs easier. This include automation, via search procedures that automatically prove theorems, and layers of abstraction (helpers, domain specific language), which make the presentation

[1] It is possible that the kernel itself has an implementation bug, but because it is a very small and simple program available in open source, we can build strong confidence that it is correct.

of formal statements and proofs closer to the traditional, human way of writing proofs. This is similar to programming languages: machine language is sufficient to write any program on a computer, but high level programming languages offer many convenient features which make writing complex programs easier and which are ultimately translated into assembly automatically. chapter 3 explains how these layers of abstraction and automation work. The rest of the present chapter gives a quick guide on how to use Lisa.

1.1 Installation

Lisa requires the Scala programming language to run. You can download and install Scala following the instructions at the Scala home page[2]. Subsequently, clone the Lisa git repository:

```
> git clone https://github.com/epfl-lara/lisa
```

To test your installation, do

```
> cd lisa
> sbt
```

`sbt` is a tool to run a Scala project and manage versions and dependencies. Once inside sbt, run the following commands:

```
> project lisa-examples
> run
```

Wait for the Lisa codebase to be compiled and then press the number corresponding to ”Example”. You should obtain the result demonstrating some example theorems proven, such as the following:

[2] www.scala-lang.org/

```
Theorem fixedPointDoubleApplication :=
  ∀'x. 'P('x) ⟹ 'P('f('x)) ⊢ 'P('x) ⟹ 'P('f('f('x)))

Theorem emptySetIsASubset := ⊢ subsetOf(emptySet, 'x)

Theorem setWithElementNonEmpty :=
  elem('y, 'x) ⊢ ¬('x = emptySet)

Theorem powerSetNonEmpty := ⊢ ¬(powerSet('x) = emptySet)
```

1.2 Development Environment

To write Lisa proofs, you can use any text editor or IDE. We recommend using *Visual Studio Code* (henceforth VSCode) with the *Metals* plugin.

A Note on Special Characters

Math often uses symbols beyond the Latin alphabet. Lisa usually admits both an English alphabet name and a unicode name for such symbols. By enabling *font ligatures*, common character sequences, such as `==>` are rendered as $\Longrightarrow$. The present document also uses the font Fira Code. Once installed on your system, you can activate it and ligatures on VSCode the following way:

1. Press Ctrl-Shift-P
2. Search for "Open User Settings (JSON)"
3. in the `settings.json` file, add:

```
"editor.fontFamily": "'Fira Code', Consolas, monospace",
"editor.fontLigatures": true,
```

<table>
<tr><th>Rendering</th><th>Input</th><th>Name</th></tr>
<tr><td>≡</td><td>===</td><td>equality</td></tr>
<tr><td>∨</td><td>\/</td><td>or</td></tr>
<tr><td>∧</td><td>/\</td><td>and</td></tr>
<tr><td>⟹</td><td>==></td><td>implies</td></tr>
<tr><td>⊢</td><td>|-</td><td>vdash</td></tr>
<tr><td>∀</td><td>U+2200</td><td>forall</td></tr>
<tr><td>∃</td><td>U+2203</td><td>exists</td></tr>
<tr><td>∈</td><td>U+2208</td><td>in</td></tr>
<tr><td>⊆</td><td>U+2286</td><td>subseteq</td></tr>
<tr><td>∅</td><td>U+2205</td><td>emptyset</td></tr>
</table>

Table 1.1: Frequently used Unicode symbols and ligatures.

Other symbols such as ∀ are Unicode symbols, which can be entered via their unicode code, depending on your OS[3], or by using an extension for VS Code such as *Fast Unicode Math Characters*, *Insert Unicode* or *Unicode Latex*. A cheat sheet of the most common symbols and how to input them is in Table 1.1. Note that by default, Unicode characters may not be printed correctly on a Windows console. You will need to activate the corresponding charset and pick a font with support for unicode in your console's options, such as Consolas.

1.3 Writing theory files

Lisa provides a canonical way of writing and organizing kernel proofs by mean of a set of utilities and a domain-specific language (DSL) made possible by some of Scala's features. To prove some theorems by yourself, start by creating a file named `MyTheoryName.scala` right next to the Example.scala file[4]. Then simply write:

[3] alt+numpad on windows, Ctrl-Shift-U+code on Linux

[4] The relative path is lisa/lisa-examples/src/main/scala

```
object MyTheoryName extends lisa.Main {

}
```

and that's it! This will give you access to all the necessary Lisa features. Let see how one can use them to prove a theorem:

$$\forall x. P(x) \implies P(f(x)) \vdash P(x) \implies P(f(f(x)))$$

To state the theorem, we first need to tell Lisa that x is a variable, f is a function symbol and P a predicate symbol.

```
object MyTheoryName extends lisa.Main {
  val x = variable
  val f = function[1]
  val P = predicate[1]

}
```

where `[1]` indicates that the symbol is of arity 1 (it takes a single argument). The symbols `x`, `f`, `P` are scala identifiers that can be freely used in theorem statements and proofs, but they are also formal symbols of FOL in Lisa's kernel. We now can state our theorem:

```
object MyTheoryName extends lisa.Main {
  val x = variable
  val f = function[1]
  val P = predicate[1]

  val fixedPointDoubleApplication = Theorem(
    ∀(x, P(x) ⟹ P(f(x))) ⊢ P(x) ⟹ P(f(f(x)))
  ) {
    ???  // Proof
  }
}
```

The theorem will automatically be named `fixedPointDoubleApplication`, like the name of the identifier it is assigned to, and will be available to reuse in future proofs. The proof itself is built using a sequence of proof step, which will update the status of the ongoing proof.

```
 1  object MyTheoryName extends lisa.Main {
 2    val x = variable
 3    val f = function[1]
 4    val P = predicate[1]
 5
 6    val fixedPointDoubleApplication = Theorem(
 7      ∀(x, P(x) ⟹ P(f(x))) ⊢ P(x) ⟹ P(f(f(x)))
 8    ) {
 9      assume(∀(x, P(x) ⟹ P(f(x))))
10      val step1 = have(P(x) ⟹ P(f(x))) by InstantiateForall
11      val step2 = have(P(f(x)) ⟹ P(f(f(x)))) by InstantiateForall
12      have(thesis) by Tautology.from(step1, step2)
13    }
14  }
```

First, we use the `assume` construct in line 6. This tells to Lisa that the assumed formula is understood as being implicitly on the left hand side of every statement in the rest of the proof.

Then, we need to instantiate the quantified formula twice using a specialized tactic. In lines 7 and 8, we use `have` to state that a formula or sequent is true (given the assumption inside `assume`), and that the proof of this is produced by the tactic `InstantiateForall`. We'll see more about the interface of a tactic later. To be able to reuse intermediate steps at any point later, we also assign the intermediates step to a variable.

Finally, the last line says that the conclusion of the theorem itself, `thesis`, can be proven using the tactic `Tautology` and the two intermediate steps we reached. `Tautology` is a tactic that is able to do reasoning with propositional connectives. It implements a complete decision procedure for propositional logic that is described in section 1.4.

Lisa is based on set theory, so you can also use set-theoretic primitives such as in the following theorem.

```
1  val emptySetIsASubset = Theorem(
2    ∅ ⊆ x
3  ) {
4    have((y ∈ ∅) ⟹ (y ∈ x)) by Tautology.from(
5                            emptySetAxiom of (x := y))
6    val rhs = thenHave (∀(y, (y ∈ ∅) ⟹ (y ∈ x))) by RightForall
7    have(thesis) by Tautology.from(
8                            subsetAxiom of (x := ∅, y := x), rhs)
9  }
10
```

We see a number of new constructs in this example. `RightForall` is another tactic (in fact it corresponds to a core deduction rules of the kernel) that introduces a quantifier around a formula, if the bound variable is not free somewhere else in the sequent. We also see in line 6 another construct: `thenHave`. It is similar to `have`, but it will automatically pass the previous statement to the tactic. Formally,

```
    have(X) by Tactic1
    thenHave (Y) by Tactic2

```

is equivalent to

```
    val s1 = have(X) by Tactic1
    have (Y) by Tactic2(s1)

```

`thenHave` allows us to not give a name to every step when we're doing linear reasoning. Finally, in lines 5 and 8, we see that tactic can refer not only to steps of the current proof, but also to previously proven theorems and axioms, such as `emptySetAxiom`. The `of` keyword indicates the axiom (or step) is instantiated in a particular way. For example:

```
    emptySetAxiom             // ==  !(x ∈ ∅)
    emptySetAxiom of (x := y) // ==  !(y ∈ ∅)

```

Lisa also allows to introduce definitions. There are essentially two kind of definitions, *aliases* and definition via *unique existence*. An alias defines a constant, a function or predicate as being equal (or equivalent) to a given formula or term. For example,

```
  val succ = DEF(x) --> union(unorderedPair(x, singleton(x)))

```

defines the function symbol `succ` as the function taking a single argument x and mapping it to the element $\bigcup\{x, \{x\}\}$[5].

The second way of defining an object is more complicated and involve proving the existence and uniqueness of an object. This is detailed in chapter 2.

You can now try to run the theory file you just wrote and verify if you made a mistake. To do so again do `> run` in the sbt console and select the number corresponding to your file. If all the output is green, perfect! If there is an error, it can be either a syntax error reported at compilation or an error in the proof. In both case, the error message can sometimes be cryptic, but it should at least consistently indicates which line of your file is incorrect.

Alternatively, if you are using IntelliJ or VS Code and Metals, you can run your theory file directly in your IDE by clicking either on the green arrow (IntelliJ) or on “run” (VS Code) next to your main object.

1.4 Common Tactics

Restate

Restate is a tactic that reasons modulo ortholattices, a subtheory of boolean algebra (see [?] and subsection 2.1.4). Formally, it is very efficient and can prove

[5] This correspond to the traditional encoding of the successor function for natural numbers in set theory.

a lot of simple propositional transformations, but not everything that is true in classical logic. In particular, it can't prove that $(a \land b) \lor (a \land c) \iff a \land (b \lor c)$ is true. It can however prove very limited facts involving equality and quantifiers. Usage:

```
have(statement) by Restate
```

tries to justify `statement` by showing it is equivalent to `True`.

```
have(statement) by Restate(premise)
```

tries to justify `statement` by showing it is equivalent to the previously proven `premise`.

Tautology

`Tautology` is a propositional solver based upon restate, but complete. It is able to prove every formula inference that holds in classical propositional logic. However, in the worst case its complexity can be exponential in the size of the formula. Usage:

```
have(statement) by Tautology
```

Constructs a proof of `statement`, if the statement is true and a proof of it using only classical propositional reasoning exists.

```
have(statement) by Tautology.from(premise1, premise2, ... )
```

Construct a proof of `statement` from the previously proven `premise1`, `premise2`,... using propositional reasoning.

RightForall, InstantiateForall

`RightForall` will generalize a statement by quantifying it over free variables. For example,

```
have(P(x)) by ???
thenHave(∀(x, P(x))) by RightForall
```

Note that if the statement inside `have` has more than one formula, x cannot appear (it cannot be *free*) in any formula other than $P(x)$. It can also not appear in any assumption.

`InstantiateForall` does the opposite: given a universally quantified statement, it will specialize it. For example:

```
have(∀(x, P(x))) by ???
thenHave(P(t)) by InstantiateForall
```

for any arbitrary term `t`.

Substitution

Substitutions allows reasoning by substituting equal terms and equivalent formulas. Usage:

```
have(statement) by Substitution.ApplyRules(subst*)(premise)
```

`subst*` is an arbitrary number of substitution. Each of those can be a previously proven fact (or theorem or axiom), or a formula. They must all be of the form `s === t` or `A <=> B`, otherwise the tactic will fail. The `premise` is a previously proven fact. The tactic will try to show that `statement` can be obtained from `premise` by applying the substitutions from `subst`. In its simplest form,

```
val subst = have(s === t) by ???
have(P(s)) by ???
thenHave(P(t)) by Substitution.ApplyRules(subst)
```

Moreover, `Substitution` is also able to automatically unify and instantiate subst rules. For example

```
val subst = have(g(x, y) === g(y, x)) by ???
have(P(g(3, 8))) by ???
thenHave(P(g(8, 3))) by Substitution.ApplyRules(subst)
```

If a `subst` is a formula rather than a proven fact, then it should be an assumption in the resulting statement. Similarly, if one of the substitution has an assumption, it should be in the resulting statement. For example,

```
val subst = have(A ⊢ Q(s) <⇒ P(s)) by ???
have(Q(s) ∧ s===f(t)) by ???
thenHave(A, f(t) === t ⊢ P(s) ∧ s===t)
    .by Substitution.ApplyRules(subst, f(t) === t)
```

File Options

Some options can be set at the start of a file, which will affect the behaviour of Lisa. These options are intended for use at a development stage.

`draft()`	Theorems outside of the current file are assumed to be true and not checked for correctness. This can speed up repetitive runs during proof drafts.
`withCache()`	Kernel proofs will be stored in binary files when theorems are constructed. On future runs with the option enabled, theorems will be constructed from the stored low level proofs. This skips running tactics to construct or search for proofs.

Chapter 2

Lisa's Trusted 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 traditional (in the mathematical community) foundational theory of all mathematics, but with some extensions and modifications to more closely match common mathematical practice:

- The syntax of Lisa's statement is an extension of first-order logic with lambda terms named **λ-FOL** (see section 2.1).
- The deductive system of Lisa's kernel is **Sequent Calculus**.
- The axiomatic theory is **ZFC Set Theory**, but the kernel is actually theory-agnostic and is sound to use with any other set of axioms. Set theory (see chapter 5).

2.1 λ-FOL: First Order Logic with Lambda Terms

First-order logic has many useful properties, but the way it is usually defined is not very convenient for practical use. In particular, it does not allow to write self-contained terms binding a variable. A typical example of this is integrals:

$$\int_0^1 x^2 dx$$

x is a subterm of this expression, but it is bound by the integral sign $\int ...dx$. Another example is notation for set comprehensions:

$$\{x \in \mathbb{N} \mid \exists y.x = y^2\}$$

here, x is again a bound variable, and moreover the expression contains a subformula. Both the integral and the comprehensions are supposed to be terms: they denote elements of the universe, unlike formulas. But in pure first-order logic, terms cannot bind variables nor contain formulas. To adress this, Lisa uses an extension of first-order logic with lambda terms, called λ-FOL.

2.1.1 Expressions

The basic elements of Lisa are called *Expressions*, generalising terms and formulas. Expressions are terms of the simply typed lambda-calculus, with two basic types: *Prop*, or propositions, corresponding to formulas, and *Ind*, or individuals, corresponding to terms. To disambiguate from Scala types and set-theoretic types, we call Prop and Sort *Sorts*. Formally:

Definition 1 (Identifiers)**.** Identifiers are pairs of strings and positive integers used to name symbols. The integer partis convenient to quickly compute fresh identifiers and.

$$ID := \mathrm{ID}(\mathit{String}, \mathit{Int}) \tag{2.1}$$

$$\tag{2.2}$$

Identifiers cannot contain the symbols `()[]{}?,;_` nor whitespace. The canonical representation of an identifier is

$$\mathrm{ID}(\text{"foo"}, i) = \begin{cases} \text{foo} & \text{if } i = 0 \\ \text{foo\_}i & \text{else} \end{cases}$$

Definition 2 (Sorts)**.** Sorts are defined by the following grammar:

$$\mathcal{S} := \text{Prop} \mid \text{Ind} \mid \mathcal{S} \to \mathcal{S} \tag{2.3}$$
$$\tag{2.4}$$

$A \to B$ is the sort of expressions taking arguments of sort A and returning a result of sort B. Note that $\to$ associates to the right, i.e. $A \to B \to C$ is the same as $A \to (B \to C)$.

Definition 3 (Expressions)**.** We define sets of *variables* $\mathcal{V}$ and *constants* $\mathcal{C}$:

$$\mathcal{V} := \text{Var}(ID, \mathcal{S}) \tag{2.5}$$
$$\mathcal{C} := \text{Cst}(ID, \mathcal{S}) \tag{2.6}$$
$$\tag{2.7}$$

When unambiguous, we typically represent variables and constants using the representation of its identifier, optionally with the sort following a colon, as in $x : \text{Ind}$.

An expression is either a variable, a constant, an application of an expression to another expression, or an abstraction of a variable over an expression. Expressions are always uniquely *sorted.* Formally:

$$\mathcal{E} := \mathcal{V} : \mathcal{S} \tag{2.8}$$
$$\mid \mathcal{C} : \mathcal{S} \tag{2.9}$$
$$\mid \text{App}(\mathcal{E} : \mathcal{S}_1 \to \mathcal{S}_2, \mathcal{E} : \mathcal{S}_1) : \mathcal{S}_2 \tag{2.10}$$
$$\mid \text{Abs}(\mathcal{V} : \mathcal{S}_1, \mathcal{E} : \mathcal{S}_2) : \mathcal{S} : 1 \to \mathcal{S}_2 \tag{2.11}$$
$$\tag{2.12}$$

We usually represent $\text{App}(f, x)$ as $f(x)$ and $\text{Abs}(x, e)$ as $\lambda x.e$. In $\lambda x.e$, All the occurences of x in e are called bound. Every expression must belong to a sort. Ill-sorted expressions are not forbidden.

Expressions of sort Propare called *formulas.* Expressions of sort Indare called *terms.* Expressions of sort Ind → Ind → ... → Ind are called *functionals.* Expressions of sort Prop → Prop → ... → Prop are called *predicates.*

Definition 4 (Constants)**.** We predefine some important logical constants:

$$\begin{aligned}
= \quad &:\text{Ind} \to \text{Ind} \to \text{Prop} \\
\top \quad &:\text{Prop} \\
\bot \quad &:\text{Prop} \\
\neg \quad &:\text{Prop} \to \text{Prop} \\
\wedge \quad &:\text{Prop} \to \text{Prop} \to \text{Prop} \\
\vee \quad &:\text{Prop} \to \text{Prop} \to \text{Prop} \\
\Rightarrow \quad &:\text{Prop} \to \text{Prop} \to \text{Prop} \\
\Leftrightarrow \quad &:\text{Prop} \to \text{Prop} \to \text{Prop} \\
\forall \quad &:(\text{Ind} \to \text{Prop}) \to \text{Prop} \\
\exists \quad &:(\text{Ind} \to \text{Prop}) \to \text{Prop} \\
\epsilon \quad &:(\text{Ind} \to \text{Prop}) \to \text{Ind}
\end{aligned}$$

They have special meaning for the deduction system of lisa, but syntactically behave the same as user-defined constants.

We call $\forall$, $\exists$ and ϵ *binders.* We often write bound expressions such as $\forall(\lambda x.P(x) \wedge Q(x))$ as $\forall x.P(x) \wedge Q(x)$, like in traditional first-order logic.

Example 1. The following are examples of expressions:

Ind	The sort of individuals, i.e. elements of the universe such as sets, numbers, etc.
$\emptyset : \text{Ind}$	The empty set
$7 : \text{Ind}$	The number 7
Prop	The sort of formulas, which can be either true or false.
$\top : \text{Prop}$	The constant true
$\bot : \text{Prop}$	The constant false
$\text{Ind} \to \text{Ind}$	The sort of functionals, taking one individuals as argument and returning an individual.
$\mathcal{P} : \text{Ind} \to \text{Ind}$	The powerset operator
$\mathcal{P}(\emptyset) : \text{Ind}$	The powerset of the empty set
$(21+6)/3 : \text{Ind}$	The result of the division of an arithmetic expression
$\text{Ind} \to \text{Ind} \to \text{Prop}$	The sort of predicates of arity 2.
$\in: \text{Ind} \to \text{Ind} \to \text{Prop}$	The membership predicate
$3 \in \mathbb{N} : \text{Prop}$	The formula stating that 3 is a natural number
$\text{Prop} \to \text{Prop} \to \text{Prop}$	The sort of connectors of arity 2.
$\wedge : \text{Prop} \to \text{Prop} \to \text{Prop}$	The conjunction connector
$\bot \wedge 3 \in \mathbb{N} : \text{Prop}$	A formula
$\lambda x.x^2+1 : \text{Ind} \to \text{Ind}$	The functional mapping x to x^2+1
$\lambda x.x = f(x) : \text{Prop}$	The predicate mapping x to whether x is a fixpoint of f

Convention Throughout this document, and in the code base, we adopt the following conventions:We use e, e_1, e_2 to denote arbitrary expressions. We use r, s, t, u to denote arbitrary terms, a, b, c to denote constants of type Ind, x, y, z to denote variables of type Indand f, g, h to denote constant or variables of type Ind → ...Ind.

We use greek letters such as ϕ, ψ, τ to denote arbitrary formulas and X, Y, Z to denote variables of type Prop. We use P, Q, R to denote constants and variables of type Prop → ... → Prop. Sets or sequences of formulas are denoted with capital greek letters Π, Σ, Γ, Δ, etc.

$\equiv$ represents both $=$ and $\iff$, depending on whether the arguments are terms or formulas.

2.1.2 Capture-Avoiding Substitution and Beta-Reduction

An important operation on expressions is the substitution of variables by expressions of the same type.

Definition 5 (Capture-avoiding Substitution of variables)**.** Given a base expression e, a variable $x : A$ and another expression $e_1 : A$, the substitution of x by e_1 inside t is denoted by $t[x := e]$ and is computed by replacing all occurences of x by r.

Formally:

$$\begin{aligned}
(x)[x := e] \equiv &\quad e & \\
(y)[x := e] \equiv &\quad y & \text{if } x \neq y \\
(f(e_1))[x := e] \equiv &\quad f(e_1[x := e]) & \\
(\lambda x.e_1)[x := e] \equiv &\quad \lambda x.e_1 & \\
(\lambda y.e_1)[x := e] \equiv &\quad \lambda y.e_1[x := e] & \text{if } x \neq y \text{ and } y \notin \text{free}(e) \\
(\lambda y.e_1)[x := e] \equiv &\quad \lambda z.e_1[y := z][x := e] & \text{otherwise, with } z \text{ fresh}
\end{aligned}$$

This is called *capture-avoiding substitution*, because the last two lines ensure that the free variables of e stay free, independently of the name of bound variables.

Applications of an abstraction to an argument *beta-reduce*, as usual in lambda calculus. For example,

$$(\lambda x.x^2 + 1)(3) \rightsquigarrow 3^2 + 1$$

It is a theorem of the simply typed lambda calculus, called the Church-Rosser theorem, that when we keep applying such reduction, we eventually reach a normal form that cannot be further reduced. This is called the *beta normal form*. Two expressions with the same beta normal form are called *alpha-equivalent* Moreover, if two expressions are identical up to renaming of bound variables, the expressions are called *alpha-equivalent*. In Lisa, expressions whose beta normal forms are alpha-equivalent are considered logicaly the same. Note however that their representation as datastructure may not be the same, and this can influence the behaviour of programs acting on them.

2.1.3 Substitution

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

Definition 6 (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 $t[x := r]$ 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

$$(\phi \land \psi)[x := r] \equiv \phi[x := r] \land \psi[x := r] \ ,$$
$$P(t_1, t_2, \dots, t_n)[x := r] \equiv P(t_1[x := r], t_2[x := r], \dots, t_n[x := r]) \ ,$$

and for binders as

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

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

if $y \neq x$ and y does not appear in r, and

$$(\forall y.\psi)[x := r] \equiv \forall z.\psi[y := z][x := r] \ ,$$

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

Example 2 (Combined substitution and beta-reduction)**.**

Base term	Substitution	Result
$f(0,3)$	$f \quad \to \quad \lambda x.y.x+y$	$0+3$
$f(0,3)$	$f \quad \to \quad \lambda y.x.x-y$	$3-0$
$f(0,3)$	$f \quad \to \quad \lambda x.y.y+y-10$	$3+3-10$
$10\times g(x)$	$g \quad \to \quad \lambda x.x^2$	$10\times x^2$
$10\times g(50)$	$g \quad \to \quad \lambda x.`f(x+2,z)$	$10\times f(50+2,z)$
$f(x,x+y)$	$f \quad \to \quad \lambda x.y.\cos(x-y)*y$	$\cos(x-(x+y))*(x+y)$
$f(0,3)=f(x,x)$	$f \quad \to \quad \lambda x.y.x+y$	$0+3=x+x$
$\forall x.f(0,3)=f(x,x)$	$f \quad \to \quad \lambda x.y.x+y$	$\forall x.0+3=x+x$
$\exists y.`f(y)\leq `f(5)$	$f \quad \to \quad \lambda x.x+y$	$\exists y_1.y_1+y\leq 5+y$

2.1.4 The Equivalence Checker

While proving theorems, trivial syntactical transformations such as $p\land q\equiv q\land p$ 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 an equivalence checker able to detect a class of equivalence-preserving logical transformations. For example, we would like the formulas $p\land q$ and $q\land p$ 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. However, it is well known that this relationship is in general undecidable, and even the $\iff$ relation for propositional logic is coNP-complete. For practicality, we need a relation that is efficiently computable.

Orthologic is such a relation: It is a weaker theory than classical logic, because it does not include the distributivity law of $\land$ and $\lor$, but it admits a quadratic-time normalization algorithm for propositional formulas [?]. The structure underlying orthologic (its Lindenbaum algebra) is that of ortholattices, similar to the relationshipbetween classical logic and Boolean algebra. The laws of orthologic are shown in Table 2.1.

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 $a\leq b \iff a\land b=a$, which, by absorption (L9), is also equivalent to $a\lor b=b$.

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

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 2.1. We write $s \sim_{OL} t$ if both $s \leq_{OL} t$ and $s \geq_{OL} t$ hold. Theorem 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 a generalization of this algorithm to λ-FOL, which also includes additional reasoning rules. It first beta-normalize expressions, expresses the formula using de Bruijn indices, and desugars $\exists.\phi$ into $\neg\forall.\neg\phi$, $\phi \Leftrightarrow \psi$ into $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$, and $\phi \Rightarrow \psi$ into $\neg\phi \vee \psi$. It then applies OL normalization, with the the additional rules of Table 2.2.

A more detailed discussion of extension of ortholattices to first-order logic, proof of correctness and implementation details can be found in [**?**] and [**?**].

	To decide...	Try...
1	$\{\wedge, \vee, \to, \leftrightarrow, \neg\}(\vec{\phi}) \leq \psi$	Base algorithm
2	$\phi \leq \{\wedge, \vee, \to, \leftrightarrow, \neg\}(\vec{\psi})$	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 2.2: 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.

2.2 Proofs in Sequent Calculus for λ-FOL

2.2.1 Sequent Calculus

The deductive system used by Lisa is an extended version of the classical Sequent Calculus.

Definition 7. 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 $\vdash \phi \to \psi$. 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-hand side of the sequent and a single conclusion on the right. During proofs however, there may be multiple elements on the right side. [1]

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. All the basic deduction rules used in Lisa's kernel are shown in Figure 2.1 and Figure 2.2. 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.

2.2.2 Proofs

A sequent calculus proof is a tree whose nodes are proof steps. The root of the proof shows the concluding statement, and the leaves are either assumptions (for example, set theoretic axioms) or proof steps taking no premise (Hypothesis, RightRefl and RestateTrue). Figure 2.3 shows an example of a proof tree for Pierce's Law in strict Sequent Calculus.

In the Lisa 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. a proof step can also be referred to by multiple subsequent proof steps, so that proofs are actually directed acyclic graphs (DAG) rather than trees. For the proof to be the linearization of a rooted DAG, the proof steps must only refer to numbers smaller than 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. Figure 2.4 shows the proof of Pierce's Law as linearized in Lisa's kernel.
Note however that thanks to the $F(OL)^2$ equivalence checker, Pierce's law can be proven in a single step:

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

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

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

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

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

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

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

$$\frac{\Gamma, \phi[x := t] \vdash \Delta}{\Gamma, \forall x.\phi \vdash \Delta}\ \text{LeftForall} \qquad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \forall x.\phi, \Delta}\ \text{RightForall}$$

$$\frac{\Gamma, \phi \vdash \Delta}{\Gamma, \exists x.\phi \vdash \Delta}\ \text{LeftExists} \qquad \frac{\Gamma \vdash \phi[x := t], \Delta}{\Gamma \vdash \exists x.\phi, \Delta}\ \text{RightExists}$$

Figure 2.1: Deduction rules allowed by Lisa's kernel (Part 1). These are typical rules for a sequent calculus for FOL. Different occurrences of the same symbols need not represent equal elements, but only elements with the same F(OL)^2 normal form.

$$\frac{\Gamma \vdash \phi[x := t], \Delta}{\Gamma \vdash \phi[x := (\epsilon x.)], \Delta}\ \text{RightEpsilon}$$

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

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

$$\frac{\Gamma \vdash \Delta}{\Gamma[(x : A) := (e : A)] \vdash \Delta[(x : A) := (e : A)]}\ \text{InstSchema}$$

$$\frac{\Gamma, \phi[f := s] \vdash \Delta}{\Gamma, \forall \vec{x}.s(\vec{x}) \equiv t(\vec{x}), \phi[f := t] \vdash \Delta}\ \text{LeftSubstEq}$$

$$\frac{\Gamma \vdash \phi[f := s], \Delta}{\Gamma, \forall \vec{x}.s(\vec{x}) \equiv t(\vec{x}) \vdash \phi[f := t], \Delta}\ \text{RightSubstEq}$$

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

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

$$\frac{}{\Gamma_2 \vdash \Delta_2}\ \text{RestateTrue}\ \text{ if True } \sim_{F(OL)^2} (\bigwedge \Gamma_2 \rightarrow \bigvee \Delta_2)$$

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

$$\frac{}{\Gamma \vdash \Delta}\ \text{Sorry: admit a statement without proof. Usage transitively tracked.}$$

Figure 2.2: Deduction rules allowed by Lisa's kernel (Part 2). Different occurrences of the same symbols need not represent equal elements, but only elements with the same F(OL)^2 normal form.

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

Figure 2.3: A proof of Pierce's law in Sequent Calculus. The bottommost sequent (root) is the conclusion.

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)	$\vdash ((\phi \to \psi) \to \phi) \to \phi$

Figure 2.4: The proof of Pierce's Law as a sequence of steps using classical Sequent Calculus rules.

0 `RestateTrue` $\vdash ((\phi \to \psi) \to \phi) \to \phi$.

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 2.3). 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 bottom-most sequent of the last proof step of the proof the "conclusion" of the proof.
Figure 2.5 shows a proof using an import.

-1	`Imported Axiom`	$\vdash \neg(x \in \emptyset)$
0	`Restate`(-1)	$(x \in \emptyset) \vdash$
1	`LeftSubstEq`(0)	$(x \in y), y = \emptyset \vdash$
2	`Restate`(1)	$(x \in y) \vdash \neg(y = \emptyset)$

Figure 2.5: A proof that if $x \in y$, then $\neg(y = \emptyset)$, using the empty set axiom. x and y are free variables.

Finally, Figure 2.6 shows a proof with quantifiers.

0	`RestateTrue`	$P(x), Q(x) \vdash P(x) \land Q(x)$
1	`LeftForall`(0)	$P(x), \forall(x, Q(x)) \vdash P(x) \land Q(x)$
2	`LeftForall`(1)	$\forall(x, P(x)), \forall(x, Q(x)) \vdash P(x) \land Q(x)$
3	`RightForall`(2)	$\forall(x, P(x)), \forall(x, Q(x)) \vdash \forall(x, P(x) \land Q(x))$
4	`Restate`(3)	$\forall(x, P(x)) \land \forall(x, Q(x)) \vdash \forall(x, P(x) \land Q(x))$

Figure 2.6: A proof showing that $\forall$ factorizes over conjunction.

For every proof step, Lisa's kernel actually expects more than only the premises and conclusion of the rule. The proof step also contains some parameters indicating how the deduction rule is precisely applied. This makes proof checking much simpler, and hence more trustworthy. Outside the kernel, Lisa includes tactic which will infer such parameters automatically (see chapter 4), so that in practice the user never has to write them. Figure 2.7 shows how a kernel proof is written in scala.

```
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 */ )
```

Figure 2.7: The proof from Figure 2.3 written for Lisa's kernel. The second argument (empty here) is the sequence of proof imports. The symbols ⟹ and -| are ligatures for ==> and |- and are syntactic sugar defined outside the kernel.

Subproofs To organize proofs, Lisa's kernel also defines the Subproof proof step. A Subproof is a single proof step in a large proof with arbitrarily many premises:

```
SCSubproof(sp: SCProof, premises: Seq[Int])
```

The first argument contain a sequent calculus proof, with one conclusion and arbitrarily many *imports*. The second arguments must justify all the imports of the inner proof with previous steps of the outer proof. A Subproof only has an organizational purpose and allows to more easily write tactics (see chapter 4). In particular, the numbering of proof steps in the inner proof is independent of the location of the subproof step in the outer proof.

Sorry

2.2.3 Proof Checker

In Lisa, a proof object by itself has no guarantee to be correct. It is 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 deduction rule 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.

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, any number of consecutive LeftAnd, RightOr, LeftNot, RightNot, LeftImplies, RightImplies, LeftIff, LeftRefl, RightRefl, LeftExistsOne and RightExistsOne proof steps can always be replaced by a singleWeakening rule. This gives some intuition about how useful the equivalence checker is to simplify proof length.

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 (Definition 7), one should use the Rewrite or Weakening steps.

Depending on wether the proof is correct or incorrect, 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, if any), and the third argument is an error message hinting at why the step is incorrectly applied.

Note that there exists a proof step, called Sorry, used to represent unimplemented proofs. The conclusion of a Sorry step will always be accepted by the proof checker. Any theorem relying on a Sorry step is not guaranteed to be correct. The usage is, however, transitively tracked and the theorem is marked as relying on Sorry.

2.3 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 5) 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 a *justification* 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 theorem will be allowed to be used in all further proofs as an import, exactly like an axiom. Axioms and theorems also have a name.

2.3.1 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.

Figure 2.8 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

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

A definition in Lisa is one of those two kinds of objects: A predicate definition or a function definition.

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

So that

```
  PredicateDefinition(P, lambda(Seq(x1, … ,x2), φ))
```

corresponds to
“For any $\vec{x}$, let $P^n(\vec{x}) := \phi_{\vec{x}}$”

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

So that

```
  FunctionDefinition(f, y, lambda(Seq(x1, … ,x2), φ))
```

corresponds to
“For any $\vec{x}$, let $f^n(\vec{x})$ be the unique y such that φ holds.”

Figure 2.8: Definitions in Lisa.

Then we obtain the defining 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 definitional mechanism requiring a proof of unique existence is typically called *extension by definition*, and allows us to extend the theory without changing what is or isn't provable, see subsection 6.1.2.

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 and theorems.

Figure 2.9 shows the types of justification in a theory (Theorem, Axiom, Definition). Figure 2.10 shows how to introduce new justifications in the theory.

2.4 Using Lisa's Kernel

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 be used directly to formalise a large library, as doing so would be very verbose. It instead serves 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 and syntactic sugar that make the kernel more user-friendly.

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: ConstantAtomicLabel,
	  expression: LambdaTermFormula
	)

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

Figure 2.9: 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: ConstantAtomicLabel,`
	`  expression: LambdaTermFormula`
	`)`

Make a new function definition	`makeFunctionDefinition(`
	`  proof: SCProof,`
	`  justifications: Seq[Justification],`
	`  label: ConstantFunctionLabel,`
	`  out: VariableLabel,`
	`  expression: LambdaTermFormula`
	`)`

Figure 2.10: The interface of a `Theory` object to introduce new theorems, axioms and definitions.

2.4.1 Syntactic Sugar

Aliases Scala accepts most unicode symbols in identifiers, allowing Lisa to define alternative representation for logical symbols

Original symbol	Alias
`top`	`True, ⊤`
`bot`	`False, ⊥`
`And`	`and, ∧`
`Or`	`or, ∨`
`Implies`	`implies, ⟹`
`Iff`	`iff, ⟺`
`Forall`	`forall, ∀`
`Exists`	`exists, ∃`
`ExistsOne`	`existsOne, ∃!`

Identifiers An identifier is a pair of a string and a number. Note that Lisa kernel does not accept whitespace nor symbols among `()[]{}_` as part of identifiers. The underscore can be used to write both the string and the integer part of an identifier at once. For example,

```
val x: VariableLabel("x_4")
```

is automatically transformed to

```
val x: VariableLabel(Identifier("x", 4))
```

Application With the following symbols:

```
val x: VariableLabel("x")
val y: VariableLabel("x")
val f: SchematicFunctionLabel("f", 2)
```

the strict syntax to construct the term `f(x, y)` is

```
Term(f, Seq(Term(x, Nil), Term(y, Nil)))
```

Extensions and implicit conversions allow one to simply write

```
f(x, y)
```

for the same result. The same holds with predicates and connectors. Moreover, binary symbols can be written infix, allowing the following syntax:

```
(f(x, y) === f(y, x)) ==> (x === y)
```

Sequents The strict syntax to construct the sequent $\phi, \psi \vdash \gamma, \delta$ is

```
Sequent(Set(phi, psi), Set(phi, psi))
```

but thanks again to extensions and implicit conversions, Lisa accepts

```
(phi, psi) |- (phi, psi)
```

Where ⊢ is the ligature for `|-` or `|` + `-`. More generally, the left and right sides of ⊢ can be any of:

- `()` — `Unit`, translated to an empty set
- a `Formula`
- a `Tuple[Formula]`
- an `Iterable[Formula]` (`Set`, `List`...)

Lambdas A Lambda expression can be created with the lambda keyword, writing a single variable by itself, or providing a sequence if the function takes multiple arguments. For example,

```
lambda(x, x+x): LambdaTermTerm
lambda(Seq(x, y, z), x+y+z): LambdaTermTerm
lambda(x, x === x): LambdaTermFormula
lambda(Seq(x, y), (x+x) === y)): LambdaTermFormula
lambda(X, X /\ y === y): LambdaFormulaFormula
lambda(Seq(X, Y), X <=> Y): LambdaFormulaFormula
```

Moreover, a term `t` is automatically converted to a LambdaTermTerm `lambda(Seq(), t)` with an empty list of arguments when needed, and similarly for LambdaTermFormula and LambdaFormulaFormula.

2.4.2 How to write helpers

These helpers and syntactic sugar are made possible by Scala's extensions and implicit conversions. Extension allow to add methods to an object a posteriori of its definition. This is especially convenient for use, as it allows us to define such helpers outside of the kernel, keeping it small. For example, to write `f(x, y)`, the object `f: SchematicFunctionLabel` must have a method called `apply`. It is defined as

```
extension (label: TermLabel) {
  def apply(args: Term*): Term = Term(label, args)
}
```

where `Term*` indicates that the function can take arbitrary many Terms as arguments. We can also defines infix symbols this way. An expression `a === b` is in fact syntactic sugar for `a.===(b)`. So we define:

```
extension (t: Term) {
  infix def ===(u: Term): Term = Term(equality, Seq(t, u))
}
```

And similarly for other symbols such as ∧, ∨, ⟹, ⟺ .

Now, consider again

```
val x: TermLabel = VariableLabel("x")
val y: TermLabel = VariableLabel("x")
val f: SchematicFunctionLabel("f", 2)
```

even with the above `apply` trick, `f(x, y)` would not compile, since `f` can apply to `Term` arguments, but not to `TermLabel`. Hence we first need to apply `x` and `y` to an empty list of argument, such as in `f(x(), y())`. This can be done automatically with implicit conversions. Implicit conversion is the mechanism allowing to cast an object of a type to an other in a canonical way. It is defined with the `given` keyword:

```
given Conversion[TermLabel, Term] = (t: Term) => Term(t, Seq())
```

Now, every time a `TermLabel` is written in a place where a `Term` is expected, it will be converted implicitly.

To learn more about Scala 3 and its capabilities, see its documentation at https://docs.scala-lang.org/scala3/book/introduction.html.

Chapter 3

Developping Mathematics with Prooflib

Lisa's kernel offers all the necessary tools to develops proofs, but reading and writing proofs written directly in its language is cumbersome. To develop and maintain a library of mathematical development, Lisa offers a dedicate interface and DSL to write proofs: Prooflib 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. Listing 3.1 is a reminder from chapter 1 of the canonical way to write a theory file in Lisa.

In this chapter, we will describe how each of these construct is made possible and how they translate to statements in the Kernel.

3.1 Richer FOL

The syntax of Prooflib is similar to the syntax of Lisa's kernel, but the *Sorts*, such as Ind and Prop, are reflected in Scala's type system, making well-sortedness checked at compile time and offering more detailed documentation and features. Prooflib's syntax also supports custom printing, such as infix notation, special handling for binders, and more.

Listing 3.1: An example of a theory file in Lisa

```scala
object MyTheoryName extends lisa.Main:
  val x = variable[Ind]
  val y = variable[Ind]
  val f = function[Ind >>: Ind]
  val P = predicate[Prop >>: Ind]

  val fixedPointDoubleApplication = Theorem(
    ∀(x, P(x) ⟹ P(f(x))) ⊢ P(x) ⟹ P(f(f(x)))
  ) {
    assume(∀(x, P(x) ⟹ P(f(x))))
    val step1 = have(P(x) ⟹ P(f(x))) by InstantiateForall
    val step2 = have(P(f(x)) ⟹ P(f(f(x)))) by InstantiateForall
    have(thesis) by Tautology.from(step1, step2)
  }

  val emptySetIsASubset = Theorem(
    ∅ ⊆ x
  ) {
    have((y ∈ ∅) ⟹ (y ∈ x)) by Tautology.from(
                            emptySetAxiom of (x := y))
    val rhs = thenHave (∀(y, (y ∈ ∅) ⟹ (y ∈ x))) by RightForall
    have(thesis) by Tautology.from(
                            subsetAxiom of (x := ∅, y := x), rhs)
  }

  @main def show = println(emptySetAxiom)
```

3.1.1 Sorts and Expressions

Definition 8 (Sorts).

```
trait Ind
trait Prop
infix trait >>:[I, O]
```

Definition 9 (Expressions). Expressions in Prooflib always correspond to an underlying expression in lisa's kernel, which can bee accessed using `myExpr.underlying`. Expressions are always *sorted*, and this sort reflects in their scala type.

```
trait Expr[S]
case class Variables[S](name: String) extends Expr[S]
case class Constants[S](name: String) extends Expr[S]
case class App[S, T](f: Expr[S >>: T], arg: Expr[S]) extends Expr[T]
case class Abs[S, T](v: Variable[S], body: Expr[T]) extends Expr[S >>: T]
```

Expressions are usually built with the following helpers:

Example 3.

```
val x = variable[Ind] //the name "x" is used automatically
val c = variable[Ind]
val ∈ = constant[Ind >>: Ind >>: Prop]
val f = variable[(Ind >>: Prop) >>: Ind]

x ∈ c : Expr[Prop]
lambda(x, x ∈ c) : Expr[Ind >>: Prop]
f(lambda(x, x ∈ c)) : Expr[Ind]
```

Expressions also support substitutions.

Definition 10 (Substitution). Substitution are most often performed with `SubstPairs`, which guarantee well-sortedness.

```
trait SubstPair extends Product:
  type S
  val _1: Variable[S]
  val _2: Expr[S]

(x := f(∅)) : SubstPair
g(x, y).subst(x := f(∅), y := x) // == g(f(∅), x)
```

but can also be performed unsafely, when sorts are not necessarily known:

```
//if ill-sorted, may crash in unpredictable ways.
myExpr.substituteUnsafe(Map(x -> s, y -> t))

//with sanity runtime check for well-sortedness
myExpr.substituteWithCheck(Map(x -> s, y -> t))
```

3.1.2 Sequents

Expressions build into sequents, which again have an underlying sequent in the kernel.

Definition 11 (Sequents)**.** Sequents are formally pairs of sets of `Expr[Prop]`.

```
case class Sequent(left: Set[Expr[Prop]], right: Set[Expr[Prop]])
```

Sequent can be built from formulas and collections of formulas:

```
val s1 = (x ∈ c) ⊢ (f(x) ∈ f(c))
val s2 = () ⊢ (f(x) ∈ f(c))
val s3 = (x ∈ c) ⊢ ()
val s4 = Set(x ∈ c, y ∈ c) ⊢ Set(x = y, x \in y)
val s4 = assumptions ⊢ (x = c)
```

the logical semantics of sequents is the same as in the kernel, i.e. a sequent is valid if and only if the conjunction of its left side implies the disjunction of its right side. But it is usually discouraged to have multiple formulas on the right side of a sequent in theorems and lemmas, as it is harder to understand. Using multiple formulas on the right side of a sequent is however allowed in intermediate steps of a proof and in proof tactics.

Sequents, like expressions, support substitutions:

```
val s = Sequent(Set(x ∈ c), Set(f(x) ∈ f(c)))
s.substitute(x := g(∅))
  // == Sequent(Set(g(∅) ∈ c), Set(f(g(∅)) ∈ f(c)))
```

3.2 Proof Builders

3.2.1 Proofs

3.2.2 Facts

3.2.3 Instantiations

3.2.4 Local Definitions

The following line of reasoning is standard in mathematical proofs. Suppose we have already proven the following fact:

$$\exists x. P(x)$$

And want to prove the property ϕ. A proof of ϕ using the previous theorem would naturally be obtained the following way:

> Since we have proven $\exists x. P(x)$, let c be an arbitrary value such that $P(c)$ holds. Hence we prove ϕ, using the fact that $P(c)$: (...).

However, introducing a definition locally corresponding to a statement of the form

$$\exists x. P(x)$$

is not a built-in feature of first order logic. This can however be simulated by introducing a fresh variable symbol c, that must stay fresh in the rest of the proof,

```
    val existentialAxiom = Axiom(exists(x, in(x, emptySet)))
    val falso = Theorem( ⊥ ) {
      val c = witness(existentialAxiom)
      have( ⊥ ) by Tautology.from(
            c.definition, emptySetAxiom of (x := c))
    }

```

Figure 3.1: An example use of local definitions in Lisa

and the assumption $P(c)$. The rest of the proof is then carried out under this assumption. When the proof is finished, the end statement should not contain c free as it is a *local* definition, and the assumption can be eliminated using the LeftExists and Cut rules. Such a c is called a *witness*. Formally, the proof in (...) is a proof of $P(c)$ ϕ. This can be transformed into a proof of ϕ by mean of the following steps:

Not that for this step to be correct, c must not be free in ϕ. This correspond to the fact that c is an arbitrary free symbol.

This simulation is provided by Lisa through the `witness` method. It takes as argument a fact showing $\exists x.P(x)$, and introduce a new symbol with the desired property. For an example, see figure 3.1.

3.3 DSL

3.3.1 Instantiations with "of"

With lisa's kernel, it is possible to instantiate a theorem proving $P(x)$ to obtain a proof of $P(t)$, for any term t, using the `Inst` rule from Figure 2.1. Lisa's DSL provides a more convenient way to do so, using the `of` keyword. It is used like so:

```
  val ax = Axiom(P(x))
  val falso = Theorem(P(c) ∧ P(d)) {
    have(thesis) by RightAnd(ax of (x := c), ax of (x := d))
  }
```

`x := d` is called a substitution pair, and is equivalent to the tuple `(x, d)`. Arbitrarily many substitution pairs can be given as argument to `of`, and the instantiations are done simultaneously. `ax of (x := c)` is called an `InstantiatedFact`, whose statement is $P(c)$, and which can be used exactly like theorems, axioms and intermediate steps in the proof. Internally, Lisa produces a proof step corresponding to the instantiation using the `Inst` rule.

The `of` keyword can also instantiate universally quantified formulas of a fact, when it contains a single formula. For example, the following code is valid:

```
  val ax = Axiom(∀(x, P(x)))
  val thm = Theorem(P(c) ∧ P(d)) {
    have(thesis) by RightAnd(ax of c, ax of d)
  }
```

Here, `ax of c` is a fact whose proven statement is again $P(c)$. It is possible to instantiate multiple $\forall$ quantifiers at once. For example if `ax` is an axiom of the form $\forall x, \forall y, P(x, y)$, then `ax of (c, d)` is a fact whose proven statement is $P(c, d)$. It is also possible to combine instantiation of free symbols and quantified variables. For example, if `ax` is an axiom of the form $\forall x, \forall y, P(x, y)$, then `ax of (c, y, P := ===)` is a fact whose proven statement is $(c = y)$.

Formally, the `of` keyword takes as argument arbitrarily many terms and substitution pairs. If there is at least one term given as argument, the base fact must have a single universally quantified formula on the right (an arbitrarily many formulas on the left). The number of given terms must be at most the number of leading universal quantifiers. Moreover, a substitution cannot instantiate any locked symbol (i.e. a symbol part of an assumption or definition). The ordering of substitution pairs does not matter, but the ordering of terms does. The resulting fact is obtained by first replacing the free symbols in the formula by the given substitution pairs, and then instantiating the quantified variables in the formula by the given terms

In general, for the following proof

```
  val ax = Axiom(∀(x, ∀(y, P(x, y))))
  val thm = Theorem(c == d) {
    have(thesis) by Restate.from(ax of (c, d, P := ===))
  }
```

Lisa will produce the following inner statements:

```
 -1 Import 0                 (   ) ⊢  ∀(x, ∀(y, P(x, y)))
  0 SequentInstantiationRule (   ) ⊢  c === d
```

Chapter 4

Tactics: Specifications and Use

Congruence

The `Congruence` tactic is used to prove sequents whose validity directly follow from the congruence closure of all equalities and formula equivalences given left of the sequent. Specifically, it works in the following cases:

- The right side contains an equality s === t or equivalence a <=> b provable in the congruence closure.
- The left side contains an negated equality !(s === t) or equivalence !(a <=> b) provable in the congruence closure.
- There is a formula a on the left and b on the right such that a and b are congruent.
- There are two formulas a and !b on the left such that a and b are congruent.
- There are two formulas a and !b on the right such that a and b are congruent.
- The sequent is Ol-valid without equality reasoning

Note that congruence closure modulo OL is an open problem.

Example 4. The following statements are provable by `Congruence`:

```
val congruence1 = Theorem ((a === b, b === c) ⊢ f(a) === f(c)) {
  have(thesis) by Congruence
}
```

```
val congruence2 = Theorem (
  (F(F(F(F(F(F(F(x))))))) === x, F(F(F(F(F(x))))) === x)
  ⊢ (F(x) === x)
) {
  have(thesis) by Congruence
}
```

```
val congruence3 = Theorem (
  (a === b, b === c, P(f(c)) <=> Q, P(f(a)))
  ⊢ Q
) {
  have(thesis) by Congruence
}
```

The tactic computes the congruence closure of all terms and formulas, with respect to the given equalities and equivalences, using an egraph datastructure [?, ?]. The egraph contains two union-find datastructure which maintain equivalence classes of formulas and terms, respectively. The union-finds are equiped with an explain method, which can output a path of equalities between any two points in the same equivalence class, as in [?]. Each such equality can come from the left hand-side of the sequent being proven (we call those *external equalities*), or be consequences of congruence. For an equality labelled by a congruence, the equalities between all children terms can recursively be explained.

Example 5. Consider again the sequent

$$a = b, b = c \vdash f(a) = f(c)$$

the domain of our egraph is$\{a, b, c, f(a), f(c)\}$. When a and b are merged and then b and c are emrged, the egraph detects that $f(a)$ and $f(c)$

are congruent and should also be merged. The explanation of $f(a) = f(c)$ is then Congruence($f(a)$, $f(c)$), and the explanation of $a = c$ is External(a, b), External(b, c).

Once the congruence closure is computed, the tactic checks if the sequent is satisfies any of the above conditions and returns a proof if it does (and otherwise fails).

Goeland

Goeland[?] is an Automated Theorem prover for first order logic. The Goeland tactic exports a statement in SC-TPTP format, and call Goeland to prove it. Goeland produce a proof file in the SC-TPTP format, from which Lisa rebuilds a kernel proof.

Usage .

```
val gothm = Theorem (() ⊢ ∃(x, ∀(y, Q(x) ⟹ Q(y)))) {
  have(thesis) by Goeland
}

//or

val gothm = Theorem (() ⊢ ∃(x, ∀(y, Q(x) ⟹ Q(y)))) {
  have(thesis) by Goeland("goeland/Test.gothm_sol")
}
```

Goeland can only be used from linux systems, and the proof files produced by Goeland should be published along the Lisa library. Calling Goeland without arguments is only available in draft mode. It will produce a proof file for the theorem (if it succeeds). When the draft mode is disabled, for publication, Lisa will provide a file name that should be happended to the tactic. This ensures that the proof can be replayed in any system using the Lisa library.

Goeland is a complete solver for first order logic, but equality is not yet supported. It is a faster alternative to the Tableau tactic.

Chapter 5

Library Development: Set Theory

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 predicates. Indeed, a predicate on the natural numbers $\mathbb{N}$ is simply a subset of $\mathbb{N}$. For example a number is even if and only if it is in the set $E \subset \mathbb{N}$ of all even numbers. Similarly, the $\leq$ relation on natural numbers can be thought of as a subset of $\mathbb{N} \times \mathbb{N}$. There, E and $\leq$ are themselves sets, and in particular terms in first order logic. 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 5, 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.

Indeed, on one hand a predicate symbol defines a truth value 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.

Lisa is based on set theory. More specifically, it is based on ZF with (still not

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

Figure 5.1: The basic symbols of ZF.

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 5.1. It also contains the 7 axioms of Zermelo (Figure 5.2), which are technically sufficient to formalize a large portion of mathematics, plus the axiom of replacement of Fraenkel (Figure 5.3), 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 expressible 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.

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

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

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

Z4 (pair). $(z \in \{x, y\}) \iff ((x = z) \vee (y = z))$

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

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

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

Z8 (comprehension schema). $\exists y . \forall x . x \in y \iff (x \in z \wedge \phi(x))$

Z9 (infinity). $\exists x . \emptyset \in x \wedge (\forall y . y \in x \implies \mathrm{U}(\{y, \{y, y\}\}) \in x)$

Figure 5.2: Axioms for Zermelo set theory.

ZF1 (replacement schema).

$$\forall x . (x \in a) \implies \forall y, z . (\psi(x, y) \wedge \psi(x, y)) \implies y = z \implies$$

$$(\exists b . \forall y . (y \in B) \implies (\exists x . (x \in a) \wedge \psi(x, y)))$$

Figure 5.3: Axioms for Zermelo-Fraenkel set theory.

5.1 Using Comprehension and Replacement

In traditional mathematics and set theory, it is standard to use *set builder notations*, to denote sets built from comprehension and replacement, for example

$$\{-x \mid x \in \mathbb{N} \land \text{isEven}(x)\}$$

This also naturally corresponds to *comprehensions* over collections in programming languages, as in Table 5.1. Those are typically syntactic sugar for a more verbose expression. For example in scala, `(0 to 9).filter(x => x % 2 == 0).map(x => -x)`. However this kind of expressions is not possible in first order logic: We can't built in any way a term that contains formulas as subexpressions, as in `filter`. So if we want to use such constructions, we need to simulate it as we did for local definitions in subsection 3.2.4.

Language	**Comprehension**
Python	`[-x for x in range(10) if x % 2 == 0]`
Haskell	`[-x | x <- [0..9], even x]`
Scala	`for (x <- 0 to 9 if x % 2 == 0) yield -x`

Table 5.1: Comprehensions in various programming languages

It turns out that the comprehension schema is a consequence of the replacement schema when the value plugged for $\psi(x, y)$ is $\phi(x) \land y = x$, i.e. when ψ denotes a restriction of the diagonal relation. Hence, what follows is built only from replacement. Note that the replacement axiom Axiom ZF1 is conditional of the schematic symbol ψ being a functional relation. It is more convenient to move this condition inside the axiom, to obtain a non-conditional equivalence. This is the approach adopted in Isabelle/ZF [?]. We instead can prove and use

$$\exists B, \forall y. y \in B \iff (\exists x. x \in A \land P(y, e) \land \forall z. \psi(x, z) \implies z = y)$$

Which maps elements of A through the functional component of ψ only. If ψ is functional, those are equivalent.

Lisa allows to write, for an arbitrary term `t` and lambda expression `P: (Term, Term) \mapsto Formula`,

```
val c = t.replace(P)
```

One can then use `c.elim(e)` to obtain the fact $e \in B \iff (\exists x. x \in A \land P(x, e) \land \forall z. \psi(x, z) \implies z = y)$. As in the case of local definitions, this statement will automatically be eliminated from the context at the end of the proof.

Moreover, we most often want to map a set by a known function. In those case, Lisa provides refined versions `t.filter`, `t.map` and `t.collect`, which are detailed in table 5.2. In particular, these versions already prove the functionality requirement of replacement.

`val c =`	**`c.elim(e)`**
`t.replace(P)`	$e \in c \iff (\exists x. x \in t \land P(x, e) \land \forall z. P(x, z) \implies z = e)$
`t.collect(F, M)`	$e \in c \iff (\exists x. x \in t \land F(x) \land M(x) = e)$
`t.map(M)`	$e \in c \iff (\exists x. x \in t \land M(x) = e)$
`t.filter(F)`	$e \in c \iff e \in t \land F(e)$

Table 5.2: Comprehensions in Lisa

Note that each of those expressions is represented as a variable symbol in the kernel proof, and the definitions are only valid inside the current proof. They should not appear in theorem statements (in which case they should be properly introduced as defined constants).

Chapter 6

Selected Theoretical Topics

Algorithms and techniques to solve and reduce formulas in propositional logic (and its generalizations) are a major field of study. They have prime relevance in SAT and SMT solving algorithms. Lisa makes use of some of them: some of the in the kernel, others as available tactics.

6.1 Set Theory and Mathematical Logic

6.1.1 First Order Logic with Schematic Variables

6.1.2 Extensions 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 12 (Conservative Extension)**.** A theory $\mathcal{T}_2$ is a conservative extension of 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 13 (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 yield 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 14. A theory $\mathcal{T}_2$ is a fully conservative extension over a theory $\mathcal{T}_1$ if:

- it is conservative, and
- 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 2. *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 3. *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 containing a new symbol α, $\Gamma \vdash \alpha$ 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. □