RWTH AACHEN UNIVERSITY

BACHELOR-THESIS

ELFE – An interactive theorem prover for undergraduate students

German title:
ELFE – Ein Interaktiver Theorembeweiser für die Lehre

Author:
Maximilian Doré

First examiner:
Prof. Dr. Jürgen Giesl

Second examiner:
Dr. Krysia Broda

Aachen, September 4, 2017

The present work was submitted to the
Chair for Computer Science 2
Research Group Computer Science 2: "Programming Languages and Verification"

Abstract

ELFE is an interactive theorem prover with an easy to use language and user interface. Many present interactive theorem provers assume knowledge of automated theorem proving, ELFE tries to abstract away the technicalities. This allows the system to be used for teaching basic proof methods in discrete Mathematics.

The user inputs a mathematical text written in fair English. The text is then converted to a special data-structure of first-order formulas. The internal representation of the text implies certain proof obligations which are checked by automated theorem provers. The background provers try to either proof the obligations or find countermodels if a obligation is wrong. The result of this verification process is then returned to the user.

Background libraries for sets, relations and functions have been developed and allow for quickly writing proofs in these domains. The system can be accessed via a reactive web interface or from the command line.

Acknowledgements

I express my sincere gratitude to my supervisor, Krysia Broda, for her exceptional support during this work and for always nudging me into the right directions.

I thank my second marker, Alessandra Russo, for her useful feedback and Jürgen Giesl for giving me the opportunity to work on this project.

Finally, I thank my family and friends for their unconditional love and support.

Contents

Chapter 1

Introduction

In the following, we will first motivate our work in Section 1.1 and describe the objectives in Section 1.2. We conclude with giving an overview of the achieved work and describing the structure of this report in Section 1.3.

1.1 Motivation

Since the information age began to emerge in the 1950s, mathematics were a main interest for researchers. Mathematics is already somewhat formal, so it seemed to be possible to let computers do proof exploration. Newell and Simon developed in 1956 the Logic Theorist Machine [20], which already could look for derivations of statements with symbolic logic. The superior computational power of computers to humans was able to explore mathematical derivations and lead to new knowledge. Since computational power increased rapidly, it seemed likely that such systems would soon substitute humans as mathematicians. As it turned out, this was wrong.

Finding proofs in symbolic logic is a hard problem. The search tree for proofs grows exponentially, such that today it still takes a lot of resources to prove more complex problems. Interactive theorem provers aim to close the gap between computational and human reasoning. The user of such a system gives cornerstones of a proof, while the system supports by checking simple steps in the proof. Such tools as ISABELLE and COQ have evolved a lot and are used in active mathematical research, e.g., COQ was used in proving the Four color theorem by Georges Gonthier [14].

The mentioned systems are tailored for advanced users. Working with these system does not only require a long training period, but also deeper knowledge on how automated theorem proving works. In this work, we want to develop a system which verifies texts close to how one would write a mathematical proof on a sheet of paper. Consider the mathematical text in Text 1.1 which proves that if a composition of two functions f and g is injective, f must be injective as well. A mathematician would typically prove this by taking two elements x and x' such that f maps them to the same element (we have used curly brackets { and } to denote function application here). By using the injectivity of the composition, one can show that this already means that the two elements x and x' were the same and thus, f is injective. In fact, this is a correct ELFE proof which will be verified later on.

```
Lemma: g∘f is injective implies f is injective.
Proof:
    Assume g∘f is injective.
    Assume x ∈ A and x' ∈ A and (f{x}) = (f{x'}).
    Then ((g∘f){x}) = ((g∘f){x'}).
    Hence x = x'.
    Hence f is injective.
qed.
```

TEXT 1.1: Exemplary ELFE text

1.2 Objectives

> "Perhaps this is the most promising aspect of formal proof: it is not merely a method to make absolutely sure we have not made a mistake in a proof, but also a tool that shows us and compels us to understand why a proof works." — Georges Gonthier [14]

The main objective of ELFE was to create an interactive theorem prover which is easily usable by students in the beginning of their mathematical studies. It aims for having an easy language that is close to the intuitive mathematical language. The archetype of the system was the SYSTEM FOR AUTOMATED DEDUCTION (SAD) [28]. This system provides a comprehensive and complex mathematical language which allows to write down nearly natural proofs. We took its general approach of converting a mathematical text into a sequence of first-order formulas and giving resulting proof obligations to automated theorem provers (ATP) as well as many syntax constructs. Going from there, we want to extend the system by a reactive web interface, parallelizing the ATP work and a mathematical library for structures that mathematical students use in their first years. Additionally, we want to provide hints on why a wrong proof fails. In order to do this, we want to give counterexamples to proof obligations. This allows for locating breaking points in a proof.

Equipped with this, ELFE shall be a valuable tool for teaching mathematical proofs. Students can play around with different approaches and get immediate feedback on which proofs work. As a side-effect, the system may popularize formalized proving. Even though many advanced users use expert theorem provers like ISABELLE or COQ, most students of Mathematics and Computer Science have no contact with formalized proving in their undergraduate studies. By beginning formal proving with an easy system, this may lower the barrier to use more complex systems later.

1.3 Achievements

We developed a powerful data structure for representing mathematical proofs, so-called *statement sequences*. We can map many proving methods into this construction and prove the correctness of our construction by a clear soundness criteria. This may be used in the future as a framework for a theorem prover that combines other

proving methods.

So far, we already have implemented some of these proving methods. They are accessible through a clean syntax and allow for proving many problems in discrete mathematics. In combination with the created background libraries for sets, relations and functions, one can quickly start writing proofs. A proof can be entered via command line or via a web interface which provides feedback on the verification status. The verification is done in parallel with several background provers, which look for proofs of the obligations as well as countermodels.

As a whole, this gives us a complete system to formalize mathematical proofs in a fairly easy manner. This system may be extended in the future in many ways.

In Chapter 2, we will introduce the logical foundations of this work and understand the operating principle of the used ATP. After giving an overview of the components of the system in Chapter 3, we will introduce the internal proof representation and language constructs in Chapter 4. The library and interfaces of the system will be discussed in Chapter 5. Afterwards, we will take a look at certain implementation details in Chapter 6. We will critically evaluate our work in Chapter 7 and compare its working to other current theorem provers in Chapter 8. In the conclusion in Chapter 9, we will give an outlook on what future extensions to the system could be made.

Chapter 2

Background

We will first introduce the basic notions and lemmas used in this work in Section 2.1. With that we can give a fragmental overview of automated theorem proving in Section 2.2 and introduce the used background provers in Section 2.3. We will conclude by introducing the mathematical domains covered in the ELFE library in Section 2.4.

2.1 Logical background

First, we will introduce the used notations for first-order logic in Section 2.1.1. Then we will take a short look at the natural deduction system in Section 2.1.2. We will conclude with presenting the foundation for present automated theorem proving in Section 2.1.3, Herbrand's Theorem.

2.1.1 First-order logic

We will define first-order logic in the following over the alphabet $\mathcal{L}$.

Definition 2.1. Alphabet.
An alphabet $\mathcal{L}$ consists of

- quantifier symbols $\exists$ and $\forall$,
- logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$,
- parentheses (and),
- an infinite set of variables,
- predicate symbols, each with a fixed arity,
- function symbols, each with a fixed arity,
- the equality symbol $=$.

We will use x, y, z as variables in the following and, among others, $P, Q, R, positive$ as predicate and $f, g, union$ as function symbols. Even though predicates and functions have overlapping notations, it is always obvious which is which due to a position in a formula as we will see in the following. Over the alphabet we can define mathematical objects.

Definition 2.2. Terms.
Terms of an alphabet $\mathcal{L}$ are inductively defined with

- variables are terms,

- $f(t_1, ..., t_n)$ is a term if $t_1, ..., t_n$ are terms and f is a function symbol with arity n in $\mathcal{L}$.

Later on, we will consider a certain class of terms, so-called **ground terms**. Ground terms do not contain any variables. We call s a **subterm** of t, denoted $t[s]$, if s is contained in t. E.g., $g(z)$ is a subterm of the term $f(x, y, g(z))$. For arity 0 we omit the parentheses and write a for the function a. We will use the so-called constants a, b, c in the following.

In order to express mathematical statements, we want to define relations between terms. This is done with formulas.

Definition 2.3. Formulas.
Formulas are inductively defined with

- $\exists x.\phi$, $\forall x.\phi$ are formulas if ϕ is a formula and x is a variable,
- $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$ are formulas if ϕ and ψ are formulas,
- $P(t_1, ..., t_n)$ is a formula if $t_1, ..., t_n$ are terms and P is a predicate symbol with arity n,
- $t_1 = t_2$ is a formula if t_1, t_2 are terms.

We will primarily consider **sentences** in the following, i.e., formulas with only bound variables. A variable x is bound if it occurs in a formula initially quantified, i.e., in the form $\exists x.\phi$ or $\forall x.\phi$. Note also that we will write $t_1 \neq t_2$ for $\neg(t_1 = t_2)$. In order to avoid ambiguities, nested formulas are enclosed with parentheses. However, we will often omit these in the following and assume that $\neg$ has the highest precedence, then $\wedge$ and $\vee$, afterwards $\rightarrow$ and $\leftrightarrow$ and finally the quantifiers $\forall$ and $\exists$ with the least precedence. For example, we will write $\exists x.P(x) \wedge \neg Q(x) \rightarrow R(x)$ instead of $\exists x.((P(x) \wedge (\neg Q(x))) \rightarrow R(x))$.

In order to define the semantics of first-order logic, we have to take into account the **domain of discourse** D, i.e., which objects we want to make statements about. We then define a so-called **structure** $\mathcal{D}$, which assigns each predicate and function symbol of $\mathcal{L}$ an object of D and contains the predicative and functional relations between the objects. Additionally to the structure $\mathcal{D}$, we need a function β which assigns all free variables an object of D. This tuple $\mathcal{I} = (\mathcal{D}, \beta)$ is a so-called **interpretation**. An interpretation is called **model** of ϕ if it evaluates ϕ to be true. We refer to [15, p. 52] on how to evaluate a formula ϕ under an interpretation $\mathcal{I}$.

Consider the exemplary formula $\phi = \exists x.philosopher(x)$. Then the interpretation $\mathcal{I} = (\mathcal{D}, \beta)$ is a model of ϕ with $D = \{philosopher, socrates\}$, $\mathcal{D}$ maps each element of D to itself and contains the relation $philosopher(socrates)$, and the function $\beta = \emptyset$.

An interpretation $\mathcal{I}$ is called **model for a set of formulas** Γ if $\mathcal{I}$ is a model for all $\gamma \in \Gamma$. A formula is a **tautology** if it holds in all interpretations, **satisfiable** if it holds in at least one interpretation and **unsatisfiable** if it holds in no interpretation. Two formulas ϕ and ψ are called **equivalent** if an interpretation $\mathcal{I}$ is a model of ϕ if and only if $\mathcal{I}$ is a model of ψ. Two formulas ϕ and ψ are called **equisatisfiable** if ϕ is satisfiable if and only if ψ is satisfiable.

With the notion of models we can define the central property automated theorem provers want to prove: Given a set of axioms Γ, does a formula ϕ follow from them?

Definition 2.4. Semantical consequence.
Let Γ be a set of first-order formulas and ϕ a first-order formula. Then ϕ is a semantical consequence of Γ, denoted $\Gamma \vDash \phi$, iff every model of Γ is also a model of ϕ.

We can disprove semantical consequence by giving an interpretation that is model of Γ, but not a model of ϕ. Such an interpretation is called **countermodel**.

2.1.2 Natural Deduction

In order to answer the question of semantical consequence, many deduction systems have been developed. They all try to reduce the semantical question $\Gamma \vDash \phi$ to a syntactical one: If we develop a technique that allows for deriving $\Gamma \vdash \phi$ syntactically if and only if $\Gamma \vDash \phi$, we do not have to take into account all possible interpretations but can answer the question easily. We will present the natural deduction system developed by Gerhard Gentzen in 1934 [10]. His goal was to give a deduction system that is close to how a human writes a proof. Indeed, we will recognize some of the derivation rules in the ELFE system later on. Additionally, we use natural deduction to show soundness of our constructions. See [9] for a more in-depth discussion of natural deduction.

The system consists of the following deduction rules:

$$(\neg\neg E): \frac{\neg\neg P}{P} \qquad (\vee I): \frac{P}{P \vee Q} \qquad (\vee E): \frac{P \vee Q \quad P \vdash R \quad Q \vdash R}{R}$$

$$(\wedge I): \frac{P \quad Q}{P \wedge Q} \qquad (\wedge E): \frac{P \wedge Q}{P} \qquad (\rightarrow I): \frac{P \vdash Q}{P \rightarrow Q} \qquad (\rightarrow E): \frac{P \rightarrow Q \quad P}{Q}$$

$$(\exists I): \frac{P(a)}{\exists x.P(x)} \qquad (\exists E): \frac{\exists x.Q(x) \quad Q(a) \vdash P}{P} \ (P \text{ does not contain } a)$$

$$(\forall I): \frac{P(a)}{\forall x.P(x)} \ (a \text{ not occurring in } P(x)) \qquad (\forall E): \frac{\forall x.P}{P(a)}$$

For example, we want to show that $\{P, P \rightarrow Q\} \vDash P \wedge Q$ with natural deduction: We can derive from P, $P \rightarrow Q$ with $(\rightarrow E)$ that Q. Then, $(\wedge I)$ gives us from P, Q that $P \wedge Q$. We write $\Gamma \vdash_{nd} \phi$ if it is possible to derive ϕ from Γ with the given rules. In fact, natural deduction is a sensible proof system.

Theorem 2.1. Natural deduction soundness and completeness.
Natural deduction is sound and complete, i.e., $\Gamma \vDash \phi$ iff $\Gamma \vdash_{nd} \phi$.

We refer to [9, p. 90f.] for a proof.

Even though natural deduction can be implemented mechanically, most present systems use other approaches to automated theorem proving. The many deduction rules lead to an explosion of possible derivation paths. In particular, it is unclear when and how to instantiate variables.

2.1.3 Herbrand's Theorem

Jacques Herbrand laid the foundation for automated theorem proving in 1930. His main idea was to instantiate the variables of a first-order formula with the terms of the formula itself. Together with a structure that interprets the predicate and function symbols of a formula with itself, this is enough to show unsatisfiability of a formula. In particular, his procedure terminates after a finite number of steps if the formula is unsatisfiable (assuming a sensible proof search). Due to the correspondence that

$$\bigwedge\{\gamma|\gamma \in \Gamma\} \wedge \neg\phi \quad \text{is unsatisfiable iff} \quad \Gamma \vDash \phi,$$

this gives us a semi-decision procedure for semantical consequence [7].

Consider a sentence ϕ. In order to apply Herbrand's theorem, we will do some preprocessing ϕ in the right form. First, all quantifiers are put outwards, leading to the **prenex normal form**. Then, all existential quantifiers are removed by **skolemization**. The resulting formula is not equivalent to the original one, but equisatisfiable. Since we are only interested in unsatisfiability, this suffices. We refer to [16] for an in-depth discussion of this preprocessing.

At this point, we have a formula of the form $\forall x_1, ..., \forall x_n.\phi$ were ϕ is quantifier-free and only contains the variables $x_1, ..., x_n$. The crucial step now is to consider only ground instances, i.e., all ground terms interpreted as themselves. As we recall, ground terms are terms that can be built from constants and function symbols without the use of variables.

Definition 2.5. Herbrand universe.
The Herbrand universe of an alphabet $\mathcal{L}$ is the set of all ground terms. If $\mathcal{L}$ does not contain any constants, a constant c_0 is added.

For example, the alphabet that contains a constant a and a function f with arity one has the Herbrand universe $\{a, f(a), f(f(a)), ...\}$.

The Herbrand universe is possibly infinite. We can approximate it with the **Herbrand expansion**, which simply enumerates elements of the Herbrand universe. The Herbrand universe forms the domain of discourse for the **Herbrand structure**, which interprets each function symbol of the Herbrand universe with itself. Herbrand's theorem now shows us that it is sufficient to consider the Herbrand structure to investigate unsatisfiability of a formula.

Theorem 2.2. Herbrand's theorem.
A formula ϕ is unsatisfiable iff there is a finite unsatisfiable set of ground terms of ϕ.

We refer to [16, p. 156] for a proof of Herbrand's Theorem.

2.2 Automated theorem proving

With Herbrand's theorem, we have a mechanical procedure to answer the question of semantical consequence. We do not have to consider all possible interpretations of a formula, but only the (possibly infinite) set of ground terms. If we have found a contradiction in this set, we know the formula to be unsatisfiable.

We will give an incomplete overview of the algorithms and concepts that are relevant to background provers in ELFE in the following. We will start with its first implementation by Gilmore in Section 2.2.1, introduce the resolution principle in Section 2.2.2, give an overview to term rewriting in Section 2.2.3 and finally conclude with the state-of-the-art superposition calculus in Section 2.2.4.

2.2.1 Algorithm of Gilmore

One of the first naive implementations of Herbrand's theorem were done by Gilmore in 1960 [16]. It enumerate larger and larger sets of the Herbrand expansion and checks if they build up a contradiction. This can be done rather efficiently by putting the ground instances into disjunctive normal form and checking each disjunct for complementary literals. However, the set of ground instances grows very quickly. Since this proof search is not directed in any way, Gilmore's algorithm is not useful for more complex theorems. We need to navigate more cleverly through all possible instantiations.

2.2.2 Resolution

Humans tend to find instantiations intelligently based on some understanding of the problem. Correspondingly, a machine also should not blindly enumerate ground instances, but instantiate variables in a clever way. This can be done by unification. We will present resolution here, see [16] for other methods that use unification like tableaux and model elimination.

The goal of unification between two terms is instantiate their variables in such a way that the terms become equal. The instantiation of their variables is done by substitution.

Definition 2.6. Substitution.
A substitution σ is a mapping from a set of variables to a set of terms.

Consider the substitution $\sigma = \{x \mapsto a, y \mapsto g(b)\}$. If we apply σ to a term $t = P(f(x), y)$, denoted $t\sigma$, we replace all variables by the corresponding terms, leading to $t\sigma = P(f(a), g(b))$.

Definition 2.7. Unifier and unifiable.
A substitution σ is a unifier for two terms t_1 and t_2 with distinct variables iff $t_1\sigma = t_2\sigma$. We then say t_1 and t_2 are unifiable.

For two terms $t_1 = P(f(x), y)$ and $t_2 = P(z, g(b))$, the substitution $\sigma = \{x \mapsto a, y \mapsto g(b), z \mapsto f(a)\}$ is a unifier since $t_1\sigma = P(f(a), g(b)) = t_2\sigma$.

Resolution considers formulas in clausal normal form (CNF). Formulas in CNF are conjunctions of **clauses**. Clauses in turn are disjunctions of positive and negative atoms. Thus, a formula in CNF is of the form:

$$\bigwedge \bigvee (\neg) P_i(x_1, ..., x_n)$$

In order to show unsatisfiability of a formula in CNF, we transform the clauses with equivalence transformations until we produce an empty clause – then, no satisfying assignment for this clause exists. Since all clauses of a formula in CNF need to be satisfied for the formula to be satisfied, this shows unsatisfiability of the whole formula.

The resolution method implements the following observation: If two clauses C_1 and C_2 contain the same term p once positive and once negated, we have to consider both clauses together without p – otherwise, if one of the clauses is satisfiable without p, the other is also satisfiable by adequately interpreting p. In first-order logic, this corresponds to the following rule:

$$\frac{C_1 \vee p_1 \quad C_2 \vee \neg p_2}{C_1\sigma \vee C_2\sigma} \quad C_1 \text{ and } C_2 \text{ no common variables, } \sigma \text{ is unifier of } p_1 \text{ and } p_2$$

Consider the example in Figure 2.1. In order to show unsatisfiability of $(P(f(a)) \vee \neg Q(x)) \wedge \neg P(y) \wedge Q(g(a))$, we derive the empty clause with two unifications.

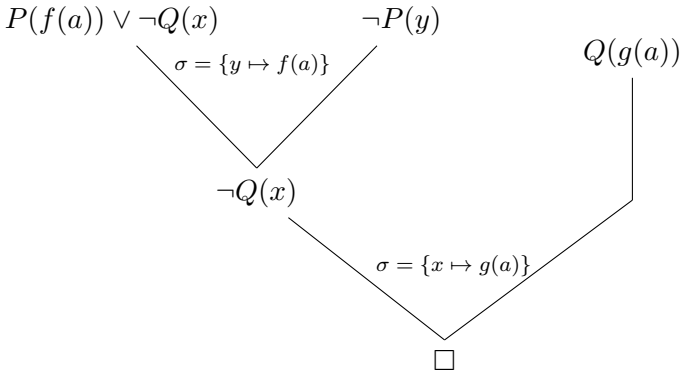


FIGURE 2.1: Example of first-order resolution

Resolution provides a more goal-oriented way of applying Herbrand's theorem and is refutationally complete. Many variants and refinements of resolution were developed, such as factoring, subsumption, tautology elimination and redundancy elimination [16]. Even though resolution is cleverer than just enumerating clauses, the search space for a proof is still quite big. In particular, the equality sign leads to problems.

2.2.3 Term rewriting

One could simply add the axioms for equality as an equivalence relation to the background theory – equality is in principal nothing more but a binary predicate. However, this leads to many possible derivation paths. The main idea of term rewriting is

to direct the equality sign. With this, it is possible to analyse the structure of the formulas more precisely and avoid redundant expansions which are not goal-oriented.

Definition 2.8. Rewrite rule.
A rewrite rule $l \to r$ is a directed equation between two terms l and r where r does not contain any variables l does not contain.

A rewrite rule $l \to r$ **rewrites** a term t if there is a **subterm** t' of t and a substitution σ such that $l\sigma = t'$. The rule is applied by replacing t' with $r\sigma$ in t. We

Definition 2.9. Term Rewriting System (TRS).
A set of rewrite rules $\mathcal{R}$ is called a Term Rewriting System (TRS).

If a term t can be rewritten to t' by applying a rule of a TRS R, we denote this as $t \to_{\mathcal{R}} t'$. If several rewrite steps are needed, we denote this as $t \twoheadrightarrow_{\mathcal{R}} t'$.

When transforming a set of equations $\mathcal{E}$ to a term rewriting system $\mathcal{R}$, one has to take care that $\mathcal{E}$ and $\mathcal{R}$ are equivalent, i.e., the congruence closures of $\mathcal{E}$ and $\mathcal{R}$ contain the same terms.

We are interested in two crucial properties of a TRS $\mathcal{R}$: $\mathcal{R}$ should be **terminating**, i.e., the relation $\to_{\mathcal{R}}$ should be well-founded. Additionally, $\mathcal{R}$ should be **confluent**, i.e., it should not matter in which way the rewrite steps are applied. This gives us an efficient way to check if two terms t_1 and t_2 are equal: We rewrite both with arbitrary rules of the TRS until no more rule can be applied. We then have $t_1 \twoheadrightarrow_{\mathcal{R}} s$ and $t_2 \twoheadrightarrow_{\mathcal{R}} s$ if and only if both terms are equal in the original equation system $\mathcal{E}$.

Since TRS are Turing-complete, showing termination of a TRS is in general not decidable due to the Halting problem. However, many methods have been developed to show termination for certain classes of TRS. An often successful method is to show that the rewrite relation $\to_{\mathcal{R}}$ is contained in another relation which is known to be terminating. These relations are called **orderings**. Examples for such orderings are the Recursive path ordering and the Lexicographic path ordering. These orderings specify a precedence for the function symbols used in the TRS.

If a TRS is not terminating, one can add rewrite rules to the system such that it terminates. Once we have a terminating TRS, we can show confluence significantly easier by showing the weaker property of local confluence with Newman's Lemma. We refer the reader to [12] for an in-depth discussion of these techniques.

Term rewriting could be applied directly in automated theorem proving to show a lemma with equality: If we construct for the context an equivalent TRS which is terminating and confluent, we can just rewrite both sides of the lemma and compare their normal forms. More often, term rewriting is used to simplify a given problem. By converting instance clauses into normal forms, comparing different clauses is more efficient.

The concept of term orderings is also used to improve resolution: Since the term orderings produce a precedence for the function symbols, one can introduce a certain

notion of maximality of terms. This maximality can be used to decide on which resolution step to apply. Ordered resolution frequently finds the right derivation path significantly faster than blind walking through the search space.

2.2.4 Superposition calculus

In order to handle equality efficiently, an evident way is to introduce inference rules for equality. Such an inference system was first introduced with paramodulation by Robinson and Wos in 1969 [24]. Paramodulation is refutationally complete and already more efficient than adding rules for equality to the background theory. The concept was refined with superposition in 1991 by Bachmair and Ganzinger [3]. The superposition calculus uses the following inference rules to derive an empty clause:

$$(= \text{Resolution}) : \frac{C \vee s \neq s'}{C\sigma} \text{ where } \sigma \text{ unifies } s \text{ and } s'$$

$$(= \text{Factoring}) : \frac{C \vee s = t \vee s' = t')}{(C \vee t \neq t' \vee s = t')\sigma} \text{ where } \sigma \text{ unifies } s \text{ and } s'$$

$$(\text{SP Right}) : \frac{D \vee t = t' \quad C \vee s[u] = s'}{(D \vee C \vee s[t'] = s')\sigma} \text{ where } \sigma \text{ unifies } t \text{ and } u, \, u \text{ not a variable}$$

$$(\text{SP Left}) : \frac{D \vee t = t' \quad C \vee s[u] \neq s'}{(D \vee C \vee s[t'] \neq s')\sigma} \text{ where } \sigma \text{ unifies } t \text{ and } u, \, u \text{ not a variable}$$

For example, to show the unsatisfiability of $f(a) = x \wedge g(x) = b$, we can apply (SP Right) to retrieve $g(f(a)) = b$. Since this equation does not contain any variables and we have $g(f(a)) \neq b$, we can apply (= Resolution) to retrieve the empty clause.

One can make the calculus terminating by finding well-founded term orderings. Analogue to ordered resolution, a prover can decide which rule to apply when based on the precedence of the function symbols. Finding appropriate termination orderings is one of the major costs in superposition-based theorem proving. Once this is done, superposition has turned out to be a very efficient calculus. It has been proven refutationally complete [3] and is used in many present theorem provers, e.g., VAMPIRE, E PROVER and SPASS [23] [26] [29].

2.2.5 Satisfiability modulo theories

So far, we only have considered progress made in automation theorem proving in the first-order predicate calculus. In parallel, the satisfiability problem for propositional logic (SAT) has been researched extensively. As a result, SAT checking, even though NP-complete, has become very efficient [18]. Satisfiability modulo theories (SMT) provers harness the efficiency of modern SAT solving for certain theories in first-order logic. Consider if we want to check the correctness of a statement in integer arithmetic:

$$x < y \wedge x + y < y$$

Then we are not interested in non-standard interpretation of the predicate $<$ and the function symbol $+$, but only in $<$ as the normal ordering of integers and $+$ as the addition operation. Thus, we can try to convert a first-order problem into a propositional one with regard to a certain background theory. Since first-order logic is in general not reducible to propositional logic, the conversions may be unsound. Hence, we have to check the generated assignments for validity in our first-order formula and feedback possible conflicts. This give us the broad operating principle of SMT checkers:

- SAT checkers are used to generate propositionally satisfiable assignments
- Underlying theory solvers test their satisfiability in the background theories
- Possible conflicts are back-tracked in the SAT solvers

A popular approach to SAT is the The Davis–Putnam–Logemann–Loveland (DPLL), whose basic principle was developed in 1962. Broadly speaking, DPLL assigns a value to a variable and checks if it breaks overall satisfiability. If it does not, it goes on with the next variable. If it does, it alters the variable and possibly the assignment of the previous variable. This back-tracking is a memory-efficient way to navigate through the space of possible assignments and has been refined in many ways since then. [18]

We refer to [18] for a detailed discussion of SMT procedures.

2.3 Implemented background provers

The ELFE system can use arbitrary background provers which interface with the TPTP format. We will introduce this format in Section 2.3.1 and then shortly introduce the currently implemented background provers E PROVER in Section 2.3.2, SPASS in Section 2.3.3, Z3 in Section 2.3.4 and BEAGLE in Section 2.3.5.

2.3.1 TPTP format

The TPTP library (*Thousands of Problems for Theorem Provers*) is maintained by the University of Miami [25]. It provides a language standard for expressing first-order formulas, conjunctive normal forms and typed higher-order logic. See an exemplary TPTP file in Figure 2.2. Such a file consists of a list of `fof` formulas. The first argument gives the user the possibility to give an identifier. The second argument is `axiom` for all formulas in the context and `conjecture` for the formula to be proven. In the example, the conjecture is that there exists a `philosopher`. This can be proven since both `hume` and `socrates` are philosophers. Strings starting with an uppercase letter are variables, e.g., `X`. Predicates and functions are denoted with a starting lowercase letter. Functions with arity 0 are constants.

```
fof(socrates,axiom,(philosopher(socrates))).
fof(hume,axiom,(philosopher(hume))).
fof(philosopher,conjecture,( ? [X] : (philosopher(X)))).
```

FIGURE 2.2: Exemplary TPTP file

2.3.2 E PROVER

E PROVER is a theorem prover for first-order logic with equality developed by Stephan Schulz at the University Stuttgart. It implements superposition calculus and rewriting. A heuristic evaluation function is used to decide which clauses are resolved. [26]

If we run E PROVER to solve the TPTP problem given in Figure 2.2, it transforms the problem into clausal form. As we see in Figure 2.3, it yields the first-order formulas relevant for the proof, i.e., the statement that Hume is a philosopher proves the conjecture.

```
# Proof found!
# SZS output start CNFRefutation
fof(philosopher, conjecture, (?[X1]:philosopher(X1)), philosopher)).
fof(hume, axiom, (philosopher(hume)), hume)).
```

FIGURE 2.3: Exemplary proof of E PROVER

2.3.3 SPASS

SPASS is a first-order prover with equality developed at the Max-Planck-Institute for Computer Science. It implements inference rules based on resolution, paramodulation and superposition [2].

As we see in Figure 2.4, SPASS gives the user the refutation sequence that proved the conjecture. Since `socrates` was found to be a `philosopher`, the system concludes that there is a `philosopher`. The program also yields the premises used for the refutation.

```
SPASS beiseite: Proof found.
% SZS status Theorem
Here is a proof with depth 0, length 3 :
% SZS output start Refutation
1[0:Inp] ||  -> philosopher(socrates)*.
3[0:Inp] philosopher(u) ||  -> .
4[0:UnC:3.0,1.0] ||  -> .
% SZS output end Refutation
Formulae used in the proof : socrates philosopher
```

FIGURE 2.4: Exemplary proof of SPASS

2.3.4 Z3

The SMT checker Z3 is developed by Microsoft. It uses the DPLL procedure. Among others, background theories for linear and nonlinear arithmetic are implemented [19].

As we see in Figure 2.5, Z3 types the initially untyped input formulas. The predicate `philosopher` gets the type $\iota \rightarrow o$, `socrates` the type ι. In this example, Z3 proves the conjecture with the resolution method. After transforming the conjecture into negated normal form, the relevant clause in the refutation sequence is

philosopher(X)) in formula 5. Since socrates is a philosopher and it is possible to unify X and socrates, the system derives the empty clause and thus proves the original conjecture that there is a philosopher.

```
% SZS status Theorem
% SZS output start Proof
tff(philosopher_type, type, (
   philosopher: $i > $o)).
tff(socrates_type, type, (
   socrates: $i)).
tff(1,plain,
    (![X1: $i] : ((~philosopher(X1)) <=> (~philosopher(X1)))),
    inference(reflexivity,[status(thm)],[])).
tff(2,plain,
    ((![X: $i] : (~philosopher(X)) <=> ![X: $i] : (~philosopher(X)))),
    inference(quant_intro,[status(thm)],[1])).
tff(3,axiom,((~?[X: $i] : philosopher(X))), philosopher).
tff(4,plain,
    (![X1: $i] : $oeq((~philosopher(X1)), (~philosopher(X1)))),
    inference(reflexivity,[status(thm)],[])).
tff(5,plain,(
    ![X: $i] : (~philosopher(X))),
    inference(nnf-neg,[status(sab)],[3, 4])).
tff(6,plain,
    (![X: $i] : (~philosopher(X))),
    inference(modus_ponens,[status(thm)],[5, 2])).
tff(7,axiom,(philosopher(socrates)), socrates).
tff(8,plain,
    (((~![X: $i] : (~philosopher(X))) | (~philosopher(socrates)))),
    inference(quant_inst,[status(thm)],[])).
tff(9,plain,
    ($false),
    inference(unit_resolution,[status(thm)],[8, 7, 6])).
% SZS output end Proof
```

FIGURE 2.5: Exemplary proof of Z3

2.3.5 BEAGLE

BEAGLE is a theorem prover for first-order logic with equality. It uses a novel approach of hierarchic superposition, i.e., it combines a superposition based prover with an SMT checker. Currently, linear integer and linear rational arithmetic background theories have been built in. [4]

Consider the TPTP problem in Figure 2.6. It states that all philosophers are scapegraces, which is wrong since Socrates was not. If we run BEAGLE, it yields the clause set that disproved the conjecture in Figure 2.7. This acts as a countermodel under the Herbrand structure. A user thus can conclude that its proof obligation was wrong since socrates does not comply with it. The constant #skF_1 is a result of internal skolemization and another counter example for the conjecture.

```
fof(philosopher,axiom,(philospher(socrates))).
fof(scapegrace,axiom,(~(scapegrace(socrates)))).
fof(lazy,conjecture,( ! [X] : ((philosopher(X)) => scapegrace(X)))).
```

FIGURE 2.6: Exemplary TPTP file

```
% SZS status CounterSatisfiable for counter.tptp
Saturated clause set:
¬scapegrace(#skF_1)
¬scapegrace(socrates)
philospher(socrates)
philosopher(#skF_1)
```

FIGURE 2.7: Exemplary countermodel from BEAGLE

2.4 Covered mathematical domains

The ELFE system comes with a library which will be introduced in Section 5.1. The library is build for basic mathematical domains. We assume familiarity with these domains and will only introduce the used notations in the following.

2.4.1 Relations

Relations assign truth values to tuples of elements with fixed arity. We write $R[x_1, ..., x_n]$ if the relation R has arity n and assigns true to the tuple $(x_1, ..., x_n)$. The **complement** R^C consists of all tuples that are not in R. We will primarily consider binary relations in the following. The **inverse** of a binary relation reverses the order of the tuples, i.e., $R^{-1}[y, x]$ iff $R[x, y]$.

Another relation S is called a **subrelation** of R, denoted $S \subseteq R$, if every tuple of S is also in R. A tuple x belongs to the **union** $R \cup S$ if it is in at least one of the relations. A tuple x belongs to the **intersection** $R \cap S$ if it is in both relations.

2.4.2 Sets

A set is a collection of distinct objects. If an object x belongs to a set A, this is denoted with $x \in A$. The **complement** of A, denoted A^C, consists of all elements that are not in A.

Another set B is called a **subset** of A, denoted $B \subseteq A$, if every element of B is also in A. The powerset of A, denoted $\mathcal{P}(A)$, is the set of all subsets of A. An element x belongs to the **union** $A \cup B$ if it is in at least one of the sets. An element x belongs to the **intersection** $A \cap B$ if it is in both sets.

2.4.3 Functions

A function $f : A \to B$ is a left-total and right-unique relation between two sets A and B. In other words, every element of A is related to exactly one element of B. We say a **maps to** b if f relates a and b, denoted $f(a) = b$. A function f is called **injective**

if every element of A is mapped to a different element of B. A function f is called **surjective** iff every element of B is mapped on. If a function is both injective and surjective, it is called **bijective**. Any bijective function has an **inverse** function f^{-1} with $f^{-1}(y) = x$ iff $f(x) = y$.

The **composition** of two functions $f : A \to B$ and $g : B \to C$, denoted $g \circ f$, results from the transitive application of both functions, i.e., $(g \circ f)(x) = z$ if $f(x) = y$ and $g(y) = z$.

Chapter 3

Architecture of ELFE

The ELFE system accepts texts in a pseudo-natural mathematical language. The user may enter a text via the *command line* or a *web interface*, see Figure 3.1. The text is then transformed into an internal representation, so-called statement sequences. This happens inside the *parser*. The internal representation imply certain proof obligations that need to be checked by background provers. This is done within the *verifier*. The verifier will call several background provers that try to prove the obligations in *prover*.

An entered proof may be incomplete or contain errors. To make it easier for the user to see these mistakes and correct the proof, we try to give countermodels to the user. This is represented with the module *countermodels*. The results are given back to the verifier. From there, we can give feedback to the user via their chosen interface.

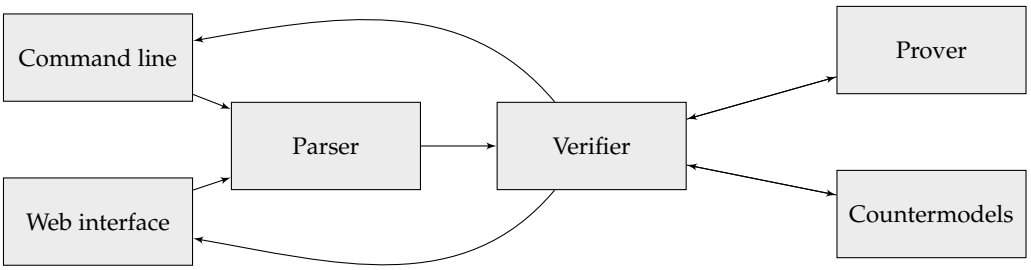


FIGURE 3.1: Architecture of the ELFE system

In the following, we will introduce the internal representation of mathematics used in ELFE in Chapter 4. With that, we can present the language constructs of ELFE and their conversion to the internal representation. We will show the soundness of our constructions and what proof obligations need to be checked by background provers.

Afterwards, we will take a look at the provided libraries about sets, relations and functions in Chapter 5. We will shortly introduce the command line and web interface.

We will conclude with an explanation of crucial algorithms of the system in Chapter 6. In particular, we will take a look at how a proof is parsed, parallel execution of the background provers and the algorithm for checking the correctness of statement sequences.

Note that Appendix B gives a short tutorial to the language constructs for the impatient.

Chapter 4

The ELFE language

In order to introduce the ELFE language, we will first take a look at how mathematical texts are represented internally in ELFE. This is done in Section 4.1. Since we use background provers to check certain formulas, we take a look at how this is done in Section 4.2. We define a soundness criterion for statements which allows us define what a sensible proof is in Section 4.3. With that, we can introduce the implemented language features in Section 4.4, Section 4.5 and Section 4.6.

4.1 Statement sequences

The data structure used within ELFE are so-called *statement sequences*, where a statement consists of an identifier, a first-order formula and a proof method as formalized in Definition 4.1.

Definition 4.1. Statement.
A statement S is a tuple of the form ID $\times$ GOAL $\times$ PROOF where

- ID is an alphanumeric string which is unique for each statement
- GOAL is a formula in first-order logic
- PROOF is either
  ASSUMED or
  BYCONTEXT or
  BYSUBCONTEXT $Id_1, ..., Id_n$ or
  BYSEQUENCE $S_1, ..., S_n$ or
  BYSPLIT $S_1, ..., S_n$

If a statement S is proved BYSEQUENCE $S_1, ..., S_n$ or BYSPLIT $S_1, ..., S_n$, we call $S_1, ..., S_n$ the children of S. If we want to access S from a child S_i, we write S_i.PARENT in the following. On the top level, a statement has no parent, thus S.PARENT = EMPTY.

A **statement sequence** is a finite list of statements $S_1, ..., S_n$.

Statements annotated with ASSUMED will be used to make assumptions in the following. Statements BYCONTEXT are those which require actual proof work. BYSUBCONTEXT is a special case of this if we want to exclude preceding lemmas from proof work. A proof BYSEQUENCE and BYSPLIT is both by a sequence. However, the interpretation is different: BYSEQUENCE is indeed an hierarchical sequence. BYSPLIT meanwhile splits the goal into several sub-goals with distinct contexts. This allows

a finer scoping of statements as we will see in Section 4.2.2.

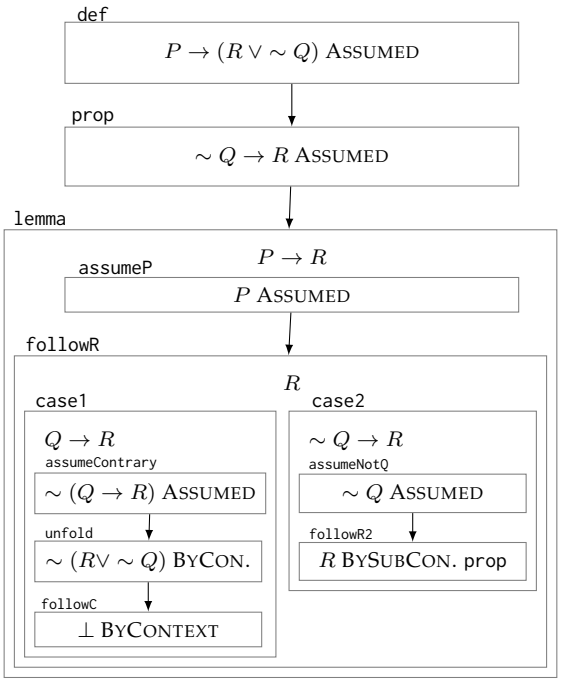


FIGURE 4.1: Exemplary statement sequence

We can illustrate a statement sequence as a sequence of boxes. Consider Figure 4.1. On the top level, we have 3 statements. Their IDs `def`, `prop` and `lemma` are represented in the left upper corner as a label. The GOAL is displayed in the first line of a box. The PROOFs of `def` and `prop` are ASSUMED, we write this kind of proof directly after the goal. In our example, these statements express some relations between propositional atoms P, Q and R.

From these two statements we want to derive the goal of `lemma`, $P \to R$. The proof consists of the sequence `assumeP` and `followR`. We write proofs BYSEQUENCE below the goal of the statement. The statement `assumeP` is again just ASSUMED, whereas `followR` is proved BYSPLIT. We represent this by putting the corresponding statements `case1` and `case2` side by side beneath the goal. Both cases combined prove R. Since we have assumed P in `assumeP`, this proves $P \to R$.

The cases are proved separately BYSEQUENCE. The proof of `case1` is BYSEQUENCE, more exactly by contradiction: We assume the converse of the goal, make a transformation and derive $\bot$. In other words: We show that the converse of the goal leads to an inconsistency. `unfold` and `followC` are annotated with BYCONTEXT – these are the statements that actually require proof work. We will see in the following that their proof work will be given to background provers. Depending on their result, we will consider the statements correct.

The proof of `case1` is straightforward: We assume the left hand side of the goal and show that the right hand side follows. The statement `folowR2` is annotated with BYSUBCONTEXT `prop` – this means its goal will also be given to background provers.

However, the context will be limited to the statement `prop`. This gives the user the possibility to select the premises from which a statement follows. We will define this more exactly in Definition 4.4.

This gives us an intuition for the internal data structure used to represent mathematical texts. Next, we will define correctness of sequences in the following Section 4.2. This gives us a framework to define what sensible statements should look like in Section 4.3.

4.2 Verifying statement sequences

We already introduced the notion of a PROOF for a statement. We will now define a criteria for correctness of statements. This is a rather weak criterion that still allows for meaningless proofs. We will see in the next Section 4.3 in which way proofs need to be constructed such that they are meaningful.

4.2.1 Prover tasks

To verify statement sequences, we need to check if certain formulas imply another formula. In other words: A conjecture needs to follow from a set of axioms. We will use the notion of proof tasks in the following. See Section 6.2 for a discussion of the implementation of the background proving.

Definition 4.2. Proof task.
A proof task consists of a set of axioms, the theory Γ, and a goal formula ϕ. We want to check if $\Gamma \vDash \phi$, i.e., evaluate the function $verify$ as follows:

$$verify(\Gamma, \phi) = \begin{cases} \text{CORRECT} & \text{if an ATP finds a proof for } \Gamma \vDash \psi \\ \text{INCORRECT} & \text{if an ATP finds a proof for } \Gamma \vDash \sim \psi \\ \text{UNKNOWN} & \text{if no ATP could find a derivation for either kind} \end{cases}$$

4.2.2 Correctness of statement sequences

Let us now define which formulas are relevant to prove a conjecture, i.e., what the context Γ is. Intuitively, we consider all statements before and above this formula as the context. Formally:

Definition 4.3. Context
In a sequence of statements $S_1, ...S_n$, the context of a statement S_k is inductively defined as

- $\Gamma(\text{EMPTY}) = \emptyset$,
- $\Gamma(S_k) = \{S_1.\text{GOAL}, ..., S_{k-1}.\text{GOAL}\} \cup \Gamma(S_k.\text{PARENT})$.

Going back to our example in Figure 4.1, the statement `unfold` consists of `assumeContrary`, `assumeP`, `def` and `prop`. The formulas of `case1`, `followR` and `lemma` are those that need to be proven and thus will not be included in the context.

In the context of statement `assumeNotQ`, we will not consider the derived statements in `case1` and have only the context `assumeP`, `def` and `prop`. This represents scoping of

statements: In a hierarchical proof, we will consider some statements only in certain cases or sub proofs.

In order to define correctness for statements that are proved BYSUBCONTEXT, we need to introduce restricted contexts. Intuitively, a restricted context matches all statements on the top level against a white list. Statements on intermediate levels are always included. This allows the user to do manual premise selection.

Definition 4.4. Restricted Context
In a sequence of statements $S_1, ...S_n$, the restricted context of a statement S_k is defined as

$$\Gamma(S_k)_{Id_1,...,Id_m} = \begin{cases} \emptyset \text{ if } \Gamma(S_k).\text{PARENT} = \text{EMPTY and } \Gamma(S_k).\text{ID} \notin \{Id_1, ..., Id_m\} \\ \{S_1.\text{GOAL}, ..., S_{k-1}.\text{GOAL}\} \cup \Gamma(S_k.\text{PARENT})_{Id_1,...,Id_m} \text{ otherwise} \end{cases}$$

Going back to our example in Figure 4.1, the statement `followR2` was annotated with BYSUBCONTEXT `prop`. The intermediate levels will be considered normally, i.e., `assumeNotQ` and `assumeP` will be included in the context. On the top level, only the explicitly stated statements will be included. Thus, `prop` will be included whereas `def` will be ignored.

Equipped with this, we can define correctness of statements and statement sequences. We will consider statements ANNOTATED as correct, the correctness of BYCONTEXT and BYSUBCONTEXT is checked by background provers and the correctness of statements annotated with BYSEQUENCE and BYSPLIT depend on the correctness of their children.

Definition 4.5. Correct statement.
We define correctness of a statement S on the structure of its proof:

- S.PROOF = ASSUMED:
  S is correct
- S.PROOF = BYCONTEXT:
  S is correct iff $verify(\Gamma(S), S.\text{GOAL}) = \text{CORRECT}$
- S.PROOF = BYSUBCONTEXT $Id_1, ..., Id_n$:
  S is correct iff $verify(\Gamma(S)_{Id_1,...,Id_n}, S.\text{GOAL}) = \text{CORRECT}$
- S.PROOF = BYSEQUENCE $S_1, ..., S_n$:
  S is correct iff $S_1, ..., S_n$ is correct
- S.PROOF = BYSPLIT $S_1, ..., S_n$:
  S is correct iff $S_1, ..., S_n$ is correct

A statement sequence $S_1, ..., S_n$ is called correct iff S_i is correct for all $i = 1, ..., n$.

If we go back to our example in Figure 4.1, we will check the correctness of `unfold` by sending the task $verify(\{\sim (Q \to R), P, P \to (R \vee \sim Q), \sim Q \to R\}, \sim (R \vee \sim Q))$ to the background provers. Since this follows directly from the previous statement, the background provers will succeed in finding a derivation. To check the correctness of `followR`, the correctness of `case1` and `case2` are taken into account.

We can produce arbitrary correct statements by annotating them with ASSUMED. In the following, we want to construct a narrower definition in order to characterize correct proofs. As we will see, the construction of statement sequences give us a big enough framework to represent and verify complex mathematical texts.

4.3 Proving with statement sequences

So far, statement sequences do not give us meaningful way to produce a more complex proof for a statement. For example, a statement which is annotated with BYSEQUENCE can have arbitrary statements as children. In the following, we will give a narrower definition of proved statements. This reflects that a proof should not make additional assumptions but follow all steps from the context. We will use the previously introduced Definition 4.3 and Definition 4.4 for contexts and restricted contexts. With that, we can produce meaningful derivations.

4.3.1 Proved statements

Let us first define the concept of a *theory*, which describes the closure of a set of formulas under semantical consequence. In other words: All formulas that already follow from a set of formulas.

Definition 4.6. Theory.
Let Γ be a set of formulas in first-order logic. Then the theory created by Γ is

$$\text{Th}(\Gamma) = \{\phi \mid \Gamma \vDash \phi\}$$

Now we can introduce proved statements. These are the statements that already follow from a context and thus do not extend the theory.

Definition 4.7. Proved statement.
Let S be a statement with S.GOAL = ϕ. We call S proved iff $\phi \in \text{Th}(\Gamma(S))$.

We will denote proved statements bold $\boldsymbol{S}$ in the following.

Statements annotated with ASSUMED are in general not proved – they are the axioms of our theories. Statements that are annotated BYCONTEXT and BYSUBCONTEXT and have been verified obviously are proved as we see in the following two lemmas.

Lemma 4.1.
Let S be a statement such that S.GOAL = P, S.PROOF = BYCONTEXT and S is correct. Then S is proved.

Proof. Since S is correct, the background provers found a proof for $\Gamma(S) \vDash P$. Thus, $P \in \text{Th}(\Gamma(S))$ □

Lemma 4.2.
Let S be a statement such that S.GOAL = P, S.PROOF = BYSUBCONTEXT $Id_1, ..., Id_n$ and S is correct. Then S is proved.

Proof. Since S is correct, the background provers found a proof for $\Gamma(S)_{Id_1,\ldots,Id_n} \vDash P$. Since $\Gamma_{Id_1,\ldots,Id_n} \subseteq \Gamma$, we have in particular $\Gamma \vDash P$. Thus, $P \in \mathrm{Th}(\Gamma(S))$ □

Our proofs will get more complex. At first, we will give an overview of the ELFE language. Then, we will give several ways to prove statements. We will use the definition of proved statements in order to construct sound proofs, i.e., proofs that do not extend the theory. This allows us to produce proved statements annotated with BYSEQUENCE and BYSPLIT.

4.4 Overview of the ELFE language

So far we only talked about internal data structures. Now we introduce the actual ELFE language. The language is pseudo-natural and tries to model how mathematicians write proofs. We present different language constructs and their conversion into statement sequences. We did not define semantics for the language yet and will introduce the language on an exemplary basis.

Let us take a look at the language at the top level. An ELFE text consists of a sequence of commands:

⟨*text*⟩ ::= ⟨*command*⟩*

⟨*command*⟩ ::= ⟨*section*⟩
| ⟨*let*⟩
| ⟨*include*⟩
| ⟨*notation*⟩

To define our grammar, we will use this kind of grammar definition in the following. Terminal symbols are denoted in sans-serif with single quotes like 'this'. Non-terminals are denoted like <*this*>. Optional non-terminals are denoted with <*t*>$^?$, multiple non-terminals with <*t*>*. Multiple non-terminals separated by a comma will be denoted like this: <*t*>, ..., <*t*>.

A <*section*> is a command that introduces statements into the resulting statement sequence. We will take a look at these in Section 4.4.2. We introduce several tactics to prove a statement in Section 4.5. The commands <*let*> and <*include*> are meta-language features and will be dealt with in Section 4.6.1 and Section 4.6.2. But first, we will take a closer look at the internal representation of formulas in Section 4.4.1. There we will also learn about the <*notation*> command which provides a way to write more intuitive predicates.

4.4.1 Formulas

ELFE uses first-order predicate logic to represent mathematics.

⟨*formula*⟩ ::= '('⟨*formula*⟩')'
| ⟨*forall*⟩
| ⟨*exists*⟩
| ⟨*formula*⟩ 'iff' ⟨*formula*⟩
| ⟨*formula*⟩ 'implies' ⟨*formula*⟩

| ⟨*formula*⟩ 'and' ⟨*formula*⟩
| ⟨*formula*⟩ 'or' ⟨*formula*⟩
| 'contradiction'
| 'not' ⟨*formula*⟩
| ⟨*atom*⟩

⟨*atom*⟩ ::= ⟨*id*⟩'('⟨*term*⟩, ..., ⟨*term*⟩')'
| ⟨*var*⟩ 'is' ⟨*id*⟩, ... , ⟨*id*⟩
| ⟨*var*⟩ 'is not' ⟨*id*⟩, ... , ⟨*id*⟩
| ⟨*sugar*⟩

⟨*term*⟩ ::= ⟨*id*⟩'('⟨*term*⟩, ..., ⟨*term*⟩')'
| ⟨*var*⟩

⟨*var*⟩ ::= ⟨*id*⟩

⟨*id*⟩ ::= ⟨*alphaNumeric*⟩*⟨*singleQuotationMark*⟩*

The definition of <*formula*> is equivalent to the classical representation of first-order logic, consider the following exemplary conversion from our language to first-order logic:

'P(x) implies contradiction' $\hookrightarrow$ $P(x) \rightarrow \bot$

The first option of writing an <*atom*> is also close to the classical notation. With terms being either variables or terms itself, it is possible nest formulas into predicates, e.g., 'subset(union(A,B),C)'. The second and third option for writing atoms models predicate assignment in a more natural way, e.g., one can write 'R is relation' for $relation(R)$ and 'R is not relation' for $\neg relation(R)$. The last option allows to write infix atoms with special characters. We will introduce this sugaring in Section 4.4.1.

The precedence follows the intuitive notation: The strongest precedence has 'not', then 'or' and 'and', afterwards 'implies', then 'iff' and finally the quantifiers <*forall*> and <*exists*>. It is always possible to introduce parentheses to group logical connectives or to make a statement more readable.

The definition of <*id*> is used in ELFE whenever a token is required. E.g., predicates and variables are allowed to be an alphanumeric string followed by arbitrary many single quotes. E.g., 'x1' is a valid variable and 'relapp(R,y,y')' is a valid atom.

In order to make quantifications more readable, we allow several ways to write them:

⟨*forall*⟩ ::= 'for all' ⟨*var*⟩. ⟨*formula*⟩
| 'for all' ⟨*var*⟩, ... ,⟨*var*⟩. ⟨*formula*⟩
| 'for all' ⟨*atom*⟩. ⟨*formula*⟩

⟨*exists*⟩ ::= 'exists' ⟨*var*⟩. ⟨*formula*⟩
| 'exists' ⟨*var*⟩, ... ,⟨*var*⟩. ⟨*formula*⟩
| 'exists' ⟨*atom*⟩. ⟨*formula*⟩

The condensed form of a quantifier 'for all x,y,z' can be used to omit several quantifiers. Allowing atoms instead of variables is due to the convention that one often says 'for all P(x). Q(x)' instead of 'for all P(x). P(x) implies Q(x)'. Dual to that, one says 'exists P(x). Q(x)' instead of 'exists P(x). P(x) and Q(x)'. Therefore, the conversion into first-order logic is as follows:

$$\text{'for all P(x). Q(x)'} \quad \hookrightarrow \quad \forall x.P(x) \rightarrow Q(x)$$

$$\text{'exists P(x). Q(x)'} \quad \hookrightarrow \quad \exists x.P(x) \wedge Q(x)$$

Notations

Even though first-order logic allows to model some domains of mathematics quite straightforward, its notation is sometimes not intuitive. The union of two sets A and B is normally expressed as $A \cup B$, and not by $union(A, B)$. In order to allow infix notations and special characters, one can use notations.

⟨*notation*⟩ ::= 'Notation' ⟨*id*⟩: ⟨*sugar*⟩.

⟨*sugar*⟩ ::= ⟨*delimiter*⟩$^?$(⟨*id*⟩⟨*delimiter*⟩)*⟨*id*⟩$^?$

⟨*delimiter*⟩ ::= ⟨*specialCharacter*⟩*

The <*notation*> command allows to introduce a new syntactic sugar. At first, an <*id*> is given. This will be used as the predicate name after converting a sugar. The sugar itself is an alternating sequence of placeholders, which follow the already given rules for a <*id*>, and special characters, which model the characteristics of that sugar. For example, 'R[x,y]' is such a sugar. The tokens 'R', 'x' and 'y' will be treated as placeholders, whereas the other characters are treated as the characteristics of the pattern. E.g., 'f[a,b]' will be matched with the sugar since the brackets and comma are in the same place and other terms are given at the appropriate places. It is in particular possible to put another <*atom*> inside a sugar; consider the example in Text 4.2.

```
Notation square: x^2.
Notation union: A ∪ B.
Notation function : f : A -> B.
Proposition: f: (A ∪ B) -> B and f is injective implies f is surjective.
Proposition: x^2 is positive.
```

TEXT 4.2: Using syntactic sugar

The propositions of Text 4.2 will be transformed into the following formulas:

$$(function(f, union(A, B), B) \wedge injective(f)) \Rightarrow surjective(f)$$

$$positive(squared(x))$$

4.4.2 Top level sections

Now that we have introduced the internal representation of formulas, we will see how one can produce actual statements. On a top level, the following sections are allowed:

⟨*section*⟩ ::= 'Definition' ⟨*id*⟩$^?$: ⟨*formula*⟩.
| 'Proposition' ⟨*id*⟩$^?$: ⟨*formula*⟩.
| 'Lemma' ⟨*id*⟩$^?$: ⟨*formula*⟩. 'Proof:' ⟨*derivation*⟩ 'qed.'

After the section marker the user may give an identifier to refer to this statement afterwards. After a colon he proceeds with a formula in our version of first-order logic.

The tokens 'Definition' and 'Proposition' are evaluated similarly: They allow users to introduce statements without a proof, i.e., annotated with ASSUMED. We introduced both sections to allow the user for more explanatory power: A 'Definition' may be used to introduce new predicates, whereas a 'Proposition' will derive simple conjectures without a proof.

Consider Text 4.3. After introducing the predicate 'subset', an additional proposition is made about when sets are equal. These sections are both introduced as ASSUMED statements in Figure 4.4. Then follows a 'Lemma'. These are sections that require a 'Proof'. We will see in the next Section 4.5 how a proof may look.

```
Definition subset: A ⊆ B iff for all x ∈ A. x ∈ B.

Proposition: A = B iff for all x. x ∈ A iff x ∈ B.

Lemma: A = B iff A ⊆ B and B ⊆ A.
Proof:
    ... .
qed.
```

TEXT 4.3: Example of top level sections

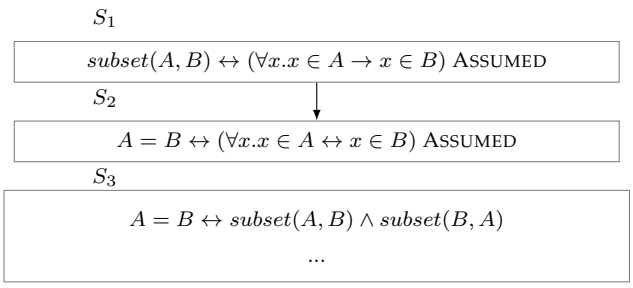


FIGURE 4.4: Exemplary statement sequence for top level sections

4.5 Derivations

In order to prove a 'Lemma', we want to show that it follows from the context – this corresponds to a proven statement as introduced in Definition 4.7. ELFE allows for several proving techniques:

- One or several sub proofs may be constructed which imply the original goal.

- According to the structure of the goal, a proof can be derived in the manner of natural deduction. For example, if we want to prove a goal as $P \to Q$, we may assume P and derive Q from it.
- To give cornerstones to a proof, one can deduce additional statements which add to the context. These cornerstones then guide the user and background provers through the proof.

We will introduce several language constructs in the following that represent these techniques and show that they indeed produce proved statements. However, we cannot show completeness of the derivation system. Since the language features were introduced on the basis of exemplary proofs, new proofs may require additional constructs and lie currently outside of the ELFE system.

Concretely, a derivation looks like this:

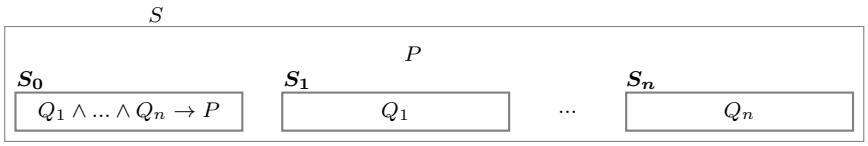
⟨*derivation*⟩ ::= ⟨*subproof*⟩*
| ⟨*case*⟩*
| ⟨*fix*⟩
| ⟨*implies*⟩
| ⟨*take*⟩
| ⟨*then*⟩
| ⟨*finalGoal*⟩

The tokens <*subproof*> and <*case*> introduce ways to split a proof into sub proofs. We will take a look at this in Section 4.5.1. In order to deduce a goal in the manner of natural deduction, we can use the constructs <*fix*> and <*implies*>. We examine this unfolding in Section 4.5.2. The constructs <*take*> and <*then*> denote ways to give additional cornerstones to a proof. This is covered in Section 4.5.4.

4.5.1 Splitting a goal

In order to prove a goal, one will often split it up in several sub goals that are proved separately. If the sub goals indeed imply the original goal, the original goal follows from the context as well. This is reflected by the following lemma:

Lemma 4.3.
Let S be a statement such that S.GOAL = P, S.PROOF = BYSPLIT $\boldsymbol{S_0}, \boldsymbol{S_1}, ..., \boldsymbol{S_n}$, $\boldsymbol{S_0}$.GOAL = $Q_1 \wedge ... \wedge Q_n \to P$, $\boldsymbol{S_i}$.GOAL = Q_i for $i = 1, ..., n$:



Then S is proved.

Proof. We have $\Gamma(S) = \Gamma(S_i)$ for $i = 0, \ldots, n$. With S_i proven for $i = 1, \ldots, n$ we have $\Gamma(S) \vDash Q_i$ for $i = 1, \ldots, n$. With $(\wedge I)$ it follows $\Gamma(S) \vDash Q_1 \wedge \cdots \wedge Q_n$.
With S_0 proven we also have $\Gamma(S) \vDash Q_1 \wedge ... \wedge Q_n \to P$. Thus, we can deduce with $(\to E)$ that $\Gamma(S) \vDash P$ and thus $P \in \mathrm{Th}(\Gamma(S))$. □

There are two ways to construct such a statement within ELFE. Sub proofs are the straightforward implementation. Cases provide a commonly used shortcut to do a case distinction.

Sub proofs

⟨*subproof*⟩ ::= 'Proof' ⟨*formula*⟩: ⟨*derivation*⟩ 'qed.'

Consider the example in Text 4.5. A common method to show the equality of two sets is to show that they are subsets of each other. If we defined sets appropriately, the soundness of this proving method will follow from the background theory.

```
Lemma: A = B.
Proof:
    Proof A ⊆ B:
        ... .
    qed.
    Proof B ⊆ A:
        ... .
    qed.
qed.
```

TEXT 4.5: Example for sub proofs

This text will be transformed into the statement in Figure 4.6. The statement S will be considered proved if S_0, S_1 and S_2 can be proved. As we have seen in Lemma 4.3, this is sound.

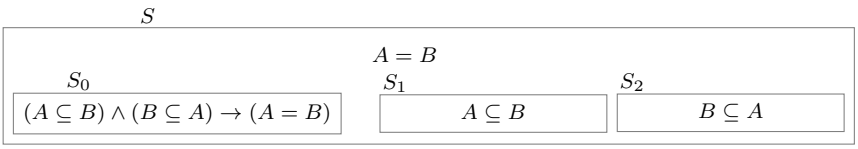


FIGURE 4.6: Conversion of sub proofs into statement

Note that this can also be used to prove goals in the manner of natural deduction: If a goal is $P \vee Q$, the user may just prove P. Consider the statement in Figure 4.7. Since $P \to P \vee Q$, this proof will be accepted.

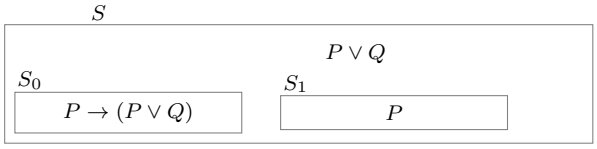


FIGURE 4.7: Natural deduction with sub proofs

We will introduce more constructions to do proofs in the manner of natural deduction in Section 4.5.2.

Case distinctions

With case distinctions, we make different assumptions and derive the goal separately from each assumption.

⟨*case*⟩ ::= 'Case' ⟨*formula*⟩: ⟨*derivation*⟩ 'qed.'

A proof with a case distinction can be found in Text 4.12.

```
Lemma: for all x. squared(x) is positive.
Proof:
    Case x is positive:
        ... .
    qed.
    Case x is negative:
        ... .
    qed.
qed.
```

TEXT 4.8: Example for case distinctions

This will be transformed into the statement in Figure 4.9. Again, Lemma 4.3 shows that this proves S if S_0, S_1 and S_2 can be proved. This will only be the case if we have corresponding premises previously in the text.

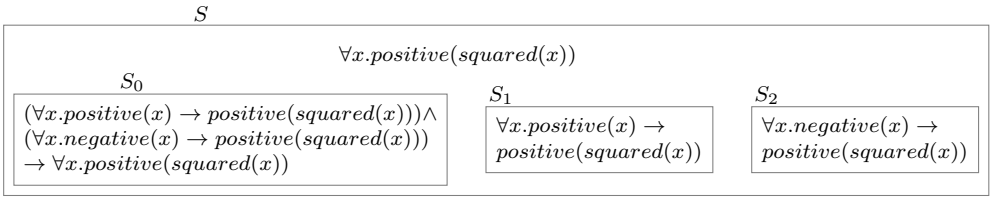


FIGURE 4.9: Exemplary case distinction as statement

4.5.2 Unfolding goals

In the following we will introduce further methods to do proofs in the style of natural deduction. E.g., if a user wants to prove a goal $P \to Q$, he will often assume P and follow Q from it.

Note that ELFE is not a complete natural deduction system and not all deduction rules are implemented.

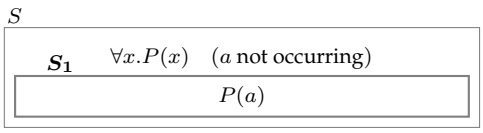
∀ introduction

If a user wants to prove an universally quantified statement, he will often fix a particular element and prove the goal for this element. Important is that no further assumptions are made about this element. This corresponds to the following rule with the constant a as the fixed element:

$$(\forall I) : \frac{P(a)}{\forall x.P(x)} \quad \text{with } a \text{ not occurring in } P(x)$$

This natural deduction rule is implemented as follows:

Lemma 4.4.
Let S be a statement such that S.GOAL = $\forall x.P(x)$ and a not occurring in S.GOAL, S.PROOF = BYSEQUENCE $\boldsymbol{S_1}$, $\boldsymbol{S_1}$.GOAL = $P(a)$:



Then S is proved.

Proof. Since S_1 is proved and $\Gamma(S) = \Gamma(S_1)$, we have $\Gamma(S) \vDash P(a)$.
With $(\forall I)$ it follows that $\Gamma(S) \vDash \forall x.P(x)$ since a does not occur in $P(x)$. Thus, $\forall x.P(x) \in \text{Th}(\Gamma(S))$. □

This is implemented in ELFE with the 'Fix' construction:

⟨*fix*⟩ ::= 'Fix' ⟨*var*⟩. ⟨*derivation*⟩

Consider the following Text 4.10.

```
Lemma: for all x. squared(x) is positive.
Proof:
    Fix x.
    ...
qed.
```

TEXT 4.10: Example for fixing an element

Text 4.10 will be transformed into the statement in Figure 4.11.

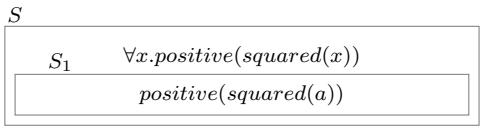


FIGURE 4.11: Fixing an element as a statement

As we have seen in in Lemma 4.4, S is proved if S_1 can be proved.

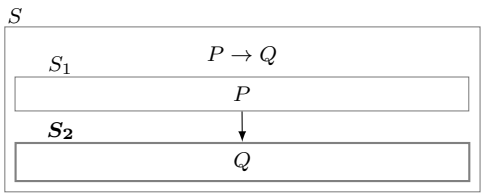
→ introduction

In order to prove a goal of the form $P \to Q$, a user will often assume P and follow Q from it. This corresponds to the natural deduction rule:

$$\to I : \frac{P \vdash Q}{P \to Q}$$

The equivalent to this rule in ELFE is as follows.

Lemma 4.5.
Let S be a statement such that S.GOAL = $P \Rightarrow Q$, S.PROOF = BYSEQUENCE $S_1, \boldsymbol{S_2}$, S_1.GOAL = P, $\boldsymbol{S_2}$.GOAL = Q:



Then S is proved.

Proof. We have $\Gamma(S_2) = \Gamma(S) \cup P$. Since S_2 is proven, $\Gamma(S) \cup P \vDash Q$. With $(\rightarrow I)$ it follows $\Gamma(S) \vDash P \rightarrow Q$. Thus, $P \rightarrow Q \in \text{Th}(\Gamma(S))$. □

Such a statement can be constructed in ELFE with this construction:

⟨*implies*⟩ ::= 'Assume' ⟨*formula*⟩. ⟨*derivation*⟩ 'Hence' ⟨*formula*⟩.

```
Notation minus: -x.

Lemma: for all x. x is positive implies -x is negative.
Proof:
    Assume x is positive.
        ...
    Hence -x is negative.
qed.
```

TEXT 4.12: Case distinctions with ELFE

Note that ELFE assumes newly introduced variables in an 'Assume' statement to be universally quantified but fixed. Thus, we can omit 'Fix x.'. Consider the resulting statement in Figure 4.13. At first, x is fixed to a. Then, we unfold the implication. If S_3 can be proved, it follows with Lemma 4.5 that S_1 is proved. Then, S is also proved with Lemma 4.4.

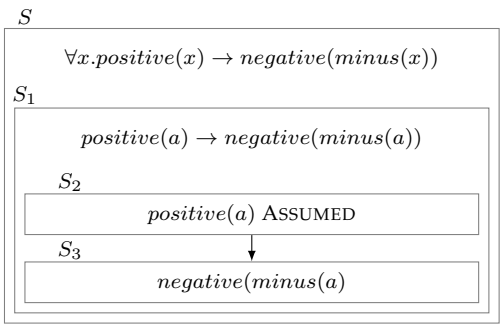


FIGURE 4.13: Assume and hence as statement

4.5.3 Inferring goals

The constructions presented in Section 4.5.2 can also be used for proving alternative goals. The parser algorithm will check if the formulas after 'Assume' and 'Hence' match the goal formula. If so, the goal will be unfolded; if not, the parser algorithm will construct the alternative goal and check if it indeed implies the original goal. In the following we want to give an example of this inferring, see Section 6.1 for an implementation of the algorithm.

Consider Text 4.14. In order to show that '$(A^C)^C$' is a subset of 'A' (which is true in the standard interpretation of sets since the sets are equal), a user will fix a particular element in '$(A^C)^C$' and show that it is as well in 'A'. Note that we left out that 'A' is a set here to ease readability. You can find the complete proof in Appendix A.

```
Lemma: ((A^C)^C) ⊆ A.
Proof:
    Assume x ∈ ((A^C)^C).
    ...
    Hence x ∈ A.
qed.
```

TEXT 4.14: Inferring a goal

The system will discover that the user proves an alternative goal and constructs it depending on the proof structure given by the user. This results in the statement in Figure 4.15. The prover has to check if the alternative goal indeed implies the original goal. This is the case if we defined '^{C}' and '$\subseteq$' appropriately. With Lemma 4.3, this construction is sound.

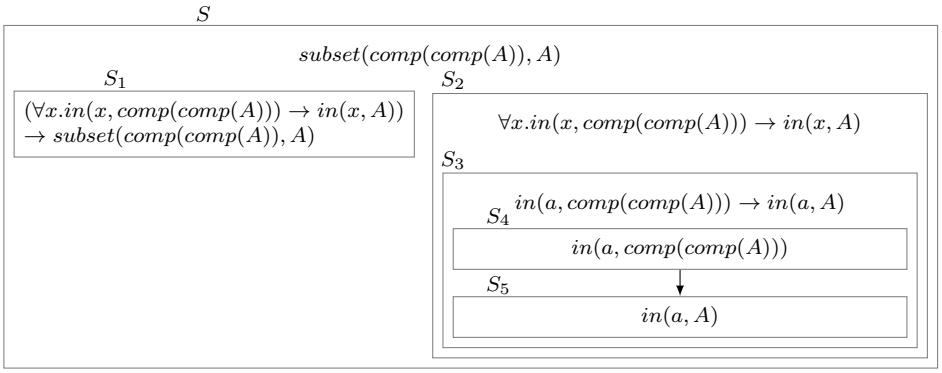


FIGURE 4.15: Inferring an alternative goal

Note that the same statement could be constructed by using sub proofs. The user could explicitly state that he proves an alternative goal as in Text 4.16. However, the implemented solution allows for shorter and more intuitive proofs. It is obvious to a user that the definition of a subset allows for this proving method.

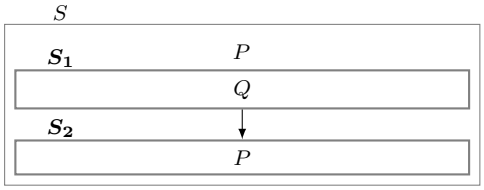
```
Lemma: ((A^C)^C) ⊆ A.
Proof:
    Proof x ∈ (A^C)^C) implies x ∈ A:
    ...
    qed.
qed.
```

TEXT 4.16: Explicitly proving an alternative goal

4.5.4 Extending the context

As a third option (besides splitting a goal into sub goals and proving it in the manner of natural deduction), a user may want to give cornerstones to a proof without changing the actual goal. This may help the background provers in finding a proof for the final goal and make a proof more readable for a human. Formally, this is expressed with the following lemma:

Lemma 4.6.
Let S be a statement such that S.GOAL = P, S.PROOF = BYSEQUENCE $\boldsymbol{S_1}$, $\boldsymbol{S_2}$, $\boldsymbol{S_1}$.GOAL = Q, $\boldsymbol{S_2}$.GOAL = P:



Then S is proved.

Proof. Since S_1 is proved, we have $\Gamma(S_1) \vDash Q$. Because of $\Gamma(S) = \Gamma(S_1)$ already $\Gamma(S) \vDash Q$. Thus, Q does not extend $\mathrm{Th}(\Gamma(S))$. Hence, with S_2 proved we have $\Gamma(S_2) \vDash P$ and it follows that already $\Gamma(S) \vDash P$. □

This lemma can be used in ELFE by using 'Then' or 'Take':

⟨*then*⟩ ::= 'Then' ⟨*formula*⟩ ⟨*by*⟩$^?$.

⟨*take*⟩ ::= 'Take' ⟨*formula*⟩ ⟨*by*⟩$^?$.

⟨*by*⟩ ::= ⟨*id*⟩, ... , ⟨*id*⟩

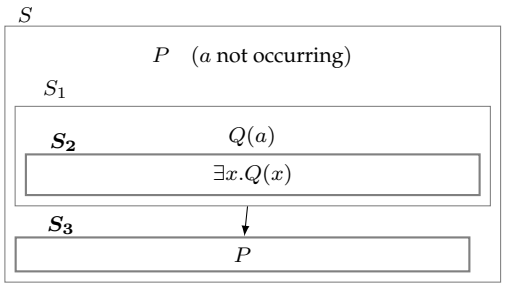
'Then' simply creates a new statement that is annotated BYCONTEXT and then added to the context of the original goal. The token <*by*> gives the possibility to prove a statement BYSUBCONTEXT by limiting the context to certain premises as seen in Definition 4.4.

'Take' corresponds to the following proof technique: If an element exists, it is fixed and then used afterwards. However, no further assumptions can be made about the fixed element. In the final conclusion, the element is not allowed to occur. This corresponds to the following natural deduction rule:

$$(\exists E): \frac{\exists x.Q(x) \quad Q(a) \vdash P}{P} \quad P \text{ does not contain } a$$

In ELFE, this rule can be used as an extension of Lemma 4.6:

Lemma 4.7.
Let S be a statement such that S.GOAL = P (a not occurring in P),
S.PROOF = BYSEQUENCE S_1 $\boldsymbol{S_3}$,
S_1.GOAL = $Q(a)$, S_1.PROOF = BYSEQUENCE $\boldsymbol{S_2}$,
$\boldsymbol{S_2}$.GOAL = $\exists x.Q(x)$,
$\boldsymbol{S_3}$.GOAL = P:



Then S is proved.

Proof. Since $\Gamma(S) = \Gamma(S_2)$, we have $\Gamma(S) \vDash \exists x.Q(x)$. With Lemma 4.5, we also have $Q(a) \vdash P$. Since a not occurring in P, it follows with $(\exists E)$ that $\Gamma(S) \vDash P$ and thus $P \in \text{Th}(\Gamma(S))$ □

Consider the example in Text 4.17. A relation R which is transitive, symmetric and bound (each element relates to at least another element), is in particular reflexive. In order to prove this, we unfold the definition of reflexivity, i.e., fix a particular element and show that the relation relates it to itself. In order to do this, we first fix a particular x. Since R is bound, there is a y such that $R[x, y]$. With the symmetry of R we know that $R[y, x]$, the transitivity finally gives us $R[x, x]$. The complete proof is in Appendix A.

```
Lemma: R is transitive, symmetric, bound implies R is reflexive.
Proof:
    Proof for all x. R[x,x]:
        Fix x.
        Take y such that R[x,y].
        Then R[y,x].
        Then R[x,x].
    qed.
qed.
```

TEXT 4.17: Giving cornerstones to a proof

This text will be transformed into the statement sequence in Figure 4.18. We omitted the outer implication $transitive(R) \wedge symmetric(R) \wedge bound(R) \rightarrow reflexive(R)$ to ease readability. The derivation of statement S reflects that we proved the alternative goal $\forall x.relapp(R, x, x)$. It is checked in statement S_1 whether this alternative

goal indeed implies the original, i.e., reflexivity needs to be defined adequately. The statement S_2 contains the actual derivation. First, we fix a particular element a in statement S_3. Thus, we only need to show that $R[a, a]$ for this fixed a. To prove this, we first retrieve the b that a is related to. We further extend the context with the fact that then also $R[b, a]$. This finally leads us to $R[a, a]$. We will see how this final proof obligation is treated in the next Section 4.5.5.

We can apply Lemma 4.7 since the constant b is not occurring in the goal of S_3. Even if we use it in S_7 when proving $relapp(R, b, a)$, the constant cannot be used outside of S_3.

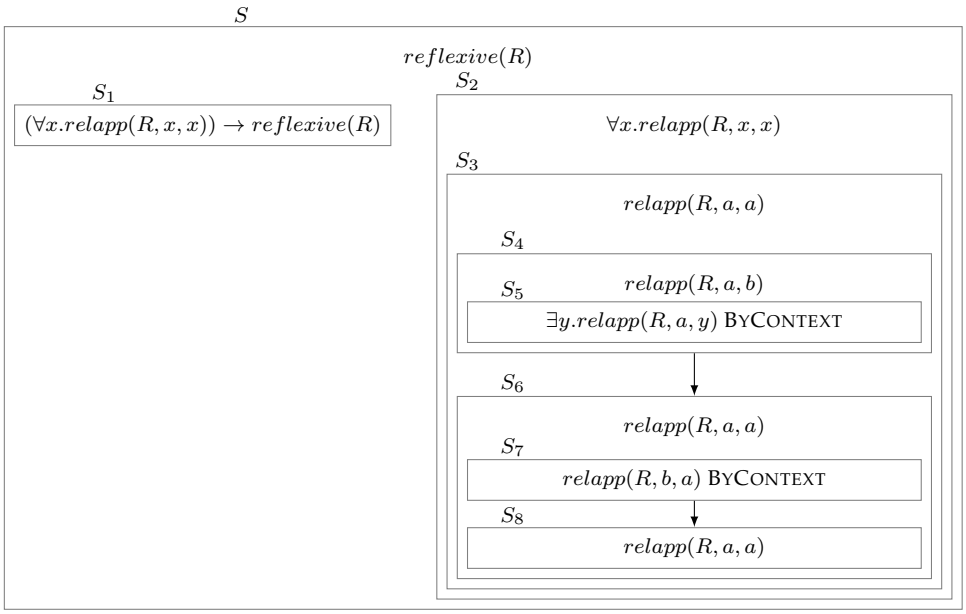


FIGURE 4.18: Extending the context with cornerstones

4.5.5 Proving the final goal

If the user does not give any further derivation, the remaining goal is given to the background provers by creating a statement annotated with BYCONTEXT. This is sound with Lemma 4.1. This corresponds to the last rule of <*derivation*>:

⟨*finalGoal*⟩ ::= ⟨*emtpy*⟩

Going back to our example in Figure 4.18, the goal of S_8 will be given to the background provers. If they find a proof for $relapp(R, a, a)$, the whole proof will be accepted.

4.6 Meta level constructs

4.6.1 Let construction

In order to make ELFE texts more concise and readable, we introduced meta variables. In a longer text, one will often reuse the same variable for a certain object, e.g.,

A for a set. So far, it would be necessary to write in every statement 'for all A. A is set implies ...'. The <*let*> construction allows to assign predicates to variables:

⟨*let*⟩ ::= 'Let' ⟨*var*⟩ 'be' ⟨*id*⟩, ..., ⟨*id*⟩.
| 'Let' ⟨*var*⟩, ... , ⟨*var*⟩ 'be' ⟨*id*⟩.
| 'Let' ⟨*atom*⟩.

Whenever a variable introduced with <*let*> is used in a following statement, we will universally quantify it and assume it has the specified predicate. Consider Text 4.19 which makes a statement about symmetric relations. This text is equivalent to the verbose version in Text 4.20.

```
Let R be relation.
Definition symmetry: R is symmetric iff for all x,y. R[x,y] implies R[y,x].
```

TEXT 4.19: Example for let construction

```
Definition symmetry: for all R. relation(R) implies (R is symmetric iff for all x,y.
R[x,y] implies R[y,x]).
```

TEXT 4.20: Text without meta variables

Note that if a lemma contains a meta variable, we automatically fix and assume it in the proving sequence. E.g., if we extend Text 4.19 by a lemma containing the variable 'R', it is not necessary to write 'Fix R. Assume R is relation. ...' in the proof.

4.6.2 Inclusions

In order to modularize and reuse mathematical texts, we introduced the <*include*> command. The parser looks in the `library` sub folder of the project for a corresponding `.elfe` file and inserts it into the original text.

⟨*include*⟩ ::= 'Include' ⟨*library*⟩.

⟨*library*⟩ ::= 'relations' | 'sets' | 'functions'

Currently, these three libraries have been created. The user can add more by putting `.elfe` files in the library sub folder.

```
Include relations.
```

TEXT 4.21: Example for using the library

The scope of meta variables introduced with 'Let' are limited to each file. This is so that the user does not have to remember all meta variables used in background libraries. Syntax sugars introduced with 'Notation' have a global scope since these are often crucial bits of the formalized library.

Chapter 5

The ELFE system

We recall that a main objective of this work was to develop an interactive proving system that has a low entry barrier. In order to this, a library was developed for basic domains of mathematics which is introduced in Section 5.1. Afterwards, we will take a look at the command line interface of the system in Section 5.2 and of the web interface in Section 5.3.

5.1 Library

Formalizing different areas in first-order logic is not always straightforward. For example, functions and relations need to be defined indirectly since we cannot quantify over predicates and functions. However, with adequate predicates and by using syntactic sugaring we can prepare libraries that make proving within these domains intuitive. In the following, we will present a library created for proving lemmas about relations in Section 5.1.1, sets in Section 5.1.2 and functions in Section 5.1.3.

5.1.1 Relations

We define relations not directly by using predicates – otherwise we need second order features to state properties of relations. Instead, we use the predicate `'relapp'` which has the relation as the first argument followed by its elements. It holds if the relation contains the tuple of the elements. Afterwards, we introduce basic properties of binary relations such as symmetry and transitivity. Additionally, we introduce the inverse and complement of a relation as well as basic operations on relations such as the union and intersection of two relations.

Notation relapp: R[x,y].

Let R,S be relation.

Definition nonempty: R is nonempty iff exists x,y. R[x,y].
Definition empty: R is empty iff not R is nonempty.
Definition symmetry: R is symmetric iff for all x,y. R[x,y] implies R[y,x].
Definition total: R is total iff for all x,y. R[x,y] or R[y,x].
Definition boundness: R is bound iff for all x. exists y. R[x,y].
Definition transitivity: R is transitive iff for all x,y,z. R[x,y] and R[y,z] implies R[x,z].
Definition reflexivity: R is reflexive iff for all x. R[x,x].

Definition equality: R = S iff (for all x,y. R[x,y] iff S[x,y]).

Notation subrelation: $R \subseteq S$.
Definition subrelation: $R \subseteq S$ iff for all x,y. R[x,y] implies S[x,y].

Notation inverse: R^{-1}.
Definition relationInverse: R^{-1} is relation and for all x,y. (R^{-1})[y,x] iff R[x,y].

Notation comp: R^C.
Definition relationComplement: R^C is relation and for all x,y. (R^C)[x,y] iff not R[x,y].

Notation relunion: $R \cup S$.
Definition relationUnion: $R \cup S$ is relation and
for all x,y. $(R \cup S)$[x,y] iff R[x,y] or S[x,y].

Notation reint: $R \cap S$.
Definition relationIntersection: $R \cap S$ is relation and
for all x,y. $(R \cap S)$[x,y] iff R[x,y] and S[x,y].

TEXT 5.1: The relations library

With the library we can write simple proofs for relations, see Text 5.2 for an exemplary proof. The fact that the union of a relation and its inverse '$R \cup (R^{-1})$' is symmetric follows directly from the definition of an inverse relation. In this example, we make a more extensive derivation: If we take an arbitrary tuple in '$R \cup (R^{-1})$', we show by case analysis that also the reversed tuple is in '$R \cup (R^{-1})$'.

Include relations.

Let R be relation.

Lemma: $R \cup (R^{-1})$ is symmetric.
Proof:
    Assume $(R \cup (R^{-1}))[x,y]$.
    Then $R[x,y]$ or $(R^{-1})[x,y]$.
    Case $R[x,y]$:
        Then $(R^{-1})[y,x]$.
    qed.
    Case $(R^{-1})[x,y]$:
        Then $R[y,x]$.
    qed.
    Hence $(R \cup (R^{-1}))[y,x]$.
qed.

TEXT 5.2: Example for using the relations library

5.1.2 Sets

Defining sets in first-order logic is straightforward. The definitions of 'subset', 'union', 'intersection' and 'complement' are direct realizations of the normal definitions in Section 2.4.2. The proposition 'subsetEquality' follows directly from the context, however, we manually state it to give a shortcut for proof search of the background provers.

Let A,B be set.

Notation in: $a \in A$.

Definition equality: $A = B$ iff for all x. $x \in A$ iff $x \in B$.

Definition nonempty: A is nonempty iff exists x. $x \in A$.
Definition empty: A is empty iff A is not nonempty.

Notation subset: $A \subseteq B$.
Definition subset: $A \subseteq B$ iff for all $x \in A$. $x \in B$.

Proposition subsetEquality: $A = B$ iff $A \subseteq B$ and $B \subseteq A$.

Notation union: $A \cup B$.
Definition union: $A \cup B$ is set and for all x. $x \in (A \cup B)$ iff $x \in A$ or $x \in B$.

Notation intersection: $A \cap B$.
Definition intersection: $A \cap B$ is set and for all x. $x \in (A \cap B)$ iff $x \in A$ and $x \in B$.

Notation comp: A^C.
Definition comp: A^C is set and for all x. $x \in (A^C)$ iff not $x \in A$.

Notation powerset: $A\wp$.
Definition powerset: $A\wp$ is set and $B \in (A\wp)$ iff $B \subseteq A$.

TEXT 5.3: The sets library

With this we can make basic proofs about sets such as the proof in Text 5.4. The complement of an union of two sets is equal to the intersection of the complements of each set. One can comprehend this directly with a Venn diagram. Formally, the proof can be done via the logical definitions of sets. This underlines the strong correspondence between set theory and first-order logic.

```
Include sets.

Let A,B be set.

Lemma: ((A ∪ B)^C) = ((A^C) ∩ (B^C)).
Proof:
    Proof ((A ∪ B)^C) ⊆ ((A^C) ∩ (B^C)):
        Assume x ∈ ((A ∪ B)^C).
        Then not x ∈ (A ∪ B).
        Then not x ∈ A and not x ∈ B.
        Then x ∈ (A^C) and x ∈ (B^C).
        Hence x ∈ ((A^C) ∩ (B^C)).
    qed.
    Proof ((A^C) ∩ (B^C)) ⊆ ((A ∪ B)^C):
        Assume x ∈ ((A^C) ∩ (B^C)).
        Then x ∈ (A^C) and x ∈ (B^C).
        Then not x ∈ A and not x ∈ B.
        Then not x ∈ (A ∪ B).
        Hence x ∈ ((A ∪ B)^C).
    qed.
qed.
```

TEXT 5.4: Example for using the set library

5.1.3 Functions

To define functions, we interpret them as relations: A function $f : A \to B$ is a binary relation which relates every element of A to exactly one element of B. The first conjunction of the 'function' definition in Text 5.5 expresses the totality criteria, the second conjunction the uniqueness. As an additional premise we propose that a function maps two equal elements to the same element in 'functionClosure'. This already follows from the definition of functions since an element of A maps to exactly one element of B. However, adding this conclusion simplifies proof search for the background provers.

We introduce the additional function 'funApp(f,x)' which represents the element that 'f' maps 'x' on. As a syntactic sugar we use curly brackets to underline that these functions in the text are distinct from functions of first-order logic. With 'funApp' we create the proposition 'funEqualitiy' which allows for directly writing '(f{x}) = (f{y}'').

The definitions of 'injective', 'surjective' and 'bijective' functions as well as the inverse function are straightforward realizations of Section 2.4.3.

In order to define function composition, we introduce the notation 'g∘f'. The definition then states that a composition of two functions is also a function and the transitive mapping of it. However, we need to introduce the additional proposition 'compositionClosure' which does not follow from the context. The proposition expresses that $f(x) = f(y)$ implies that also $g(f(x)) = g(f(y))$. Since we are working with first-order logic, we cannot retrieve the objects created by $f(x)$ respectively $f(y)$ (the proposition derived in 'funEquality' just expresses equality of two objects and does not require the actual objects). Thus, we cannot apply function g to $f(x)$ (respectively $f(y)$) and need this additional proposition.

```
Include sets.
Include relations.

Let A,B,C be set.

Notation function: f: A -> B.

Definition function: for all f.
    f: A -> B iff for all x ∈ A. exists y ∈ B.
        relapp(f,x,y) and
        (for all y2 ∈ B. y = y2 or not relapp(f,x,y2)).

Let f: A -> B.

Proposition functionClosure:
for all x1 ∈ A. for all x2 ∈ A. x1 = x2 implies exists y ∈ B. f[x1,y] and f[x2,y].

Definition injective: f is injective iff
    for all x1 ∈ A. for all x2 ∈ A. for all y ∈ B. f[x1,y] and f[x2,y] implies x1 = x2.

Definition surjective: f is surjective iff
    for all y ∈ B. exists x ∈ A. f[x,y].

Definition bijective: f is bijective iff f is injective and f is surjective.

Notation inverse: f^-1.
Definition inverse: (f^-1): B -> A and
    (for all x ∈ A. for all y ∈ B. f[x,y] implies (f^-1)[y,x]).

Let I be set.
Let i: I -> I.

Definition identity: i is identity iff for all x ∈ I. i[x,x].

Notation funApp: f{x}.
Proposition funEquality: for all x1 ∈ A. for all x2 ∈ A. (f{x1}) = (f{x2})
    iff exists y ∈ B. (f[x1,y] and f[x2,y]).

Let g: B -> C.

Notation composition: g∘f.
Definition composition: (g∘f): A -> C and
    (for all x ∈ A. for all y ∈ B. for all z ∈ C.
        ((f[x,y] and g[y,z]) implies (g∘f)[x,z])).

Proposition compositionClosure:
    for all c ∈ A. for all d ∈ A. for all e ∈ B. f[c,e] and f[d,e] implies
        exists m ∈ C. (g∘f)[c,m] and (g∘f)[d,m].
```

TEXT 5.5: The functions library

Since first-order logic has limited capabilities for talking about functions, the functions library is not always intuitive. However, given the discussed propositions and sugars, some proofs about functions can be done quite straightforwardly. Consider Text 5.6 which proofs that if a composition of two functions is injective, in particular the firstly applied function is injective. First, we unfold the implication of the lemma. Then we use the definition of injectivity to prove an additional implication:

$f(x) = f(x') \rightarrow x = x'$. This can be proven by the fact that the composition is already injective.

```
Include functions.

Let A,B,C be set.

Let f: A -> B.
Let g: B -> C.

Lemma: g∘f is injective implies f is injective.
Proof:
    Assume g∘f is injective.
    Assume x1 ∈ A and x2 ∈ A and (f{x1}) = (f{x2}).
    Then ((g∘f){x1}) = ((g∘f){x2}).
    Hence x1 = x2.
    Hence f is injective.
qed.
```

TEXT 5.6: Example of proof with functions library

5.2 Command line interface

ELFE can be used directly from the command line. The user either may give the text to verify via the standard input stream or a file containing the text as a command line argument:

```
./elfe theorem.elfe
cat theorem.elfe | ./elfe
```

Consider the example in Text 5.7 which makes a trans observation about the transitivity of equality.

```
Let A,B,C be element.

Lemma trans: A = B and B = C implies A = C.
Proof:
    Assume A = B and B = C.
    Hence A = C.
qed.
```

TEXT 5.7: Simple ELFE text

After executing `elfe`, the program runs through two stages as we see in Figure 5.8: First, the text is parsed. The constructed statement sequence is presented to the user. If the user gave an identifier, this is used as the identifier of the statement. Otherwise, an incrementing identifier is assigned.

Afterwards, the correctness check of the statement sequence is shown to the user. In order to check a statement annotated with BYSEQUENCE, all children are considered.

ASSUMED statements are just added to the context. The statement that actually requires proof work is passed to the background provers. In this example, E PROVER was the first to prove the conjecture. Note that all variables are prefixed with `c`. This is due to the fact that we prove universally quantified statements by fixing an arbitrary element, i.e., proving it for a constant instead of a variable. Thus, `A`, `B` and `C` are fixed constants.

Afterwards, the overall result of the verification is given to the user. Only if all statements are correct, the result is `CORRECT`. The program concludes with some benchmarking statistics.

```
--------------------------PARSING-------------------------
trans: ! [VA] : ((element(VA)) => (! [VB] : ((element(VB))
    => (! [VC] : ((element(VC)) => (((VA=VB) & (VB=VC)) =>
    (VA=VC))))))) -- Prove by sequence:
|  s1: (element(cA)) & ((element(cB)) & (element(cC))) -- Assumed
|  s2: (cA=cB) & (cB=cC) -- Assumed
|  s3: cA=cC -- ByContext
-------------------------VERIFYING------------------------
Check  trans: ! [VA] : ((element(VA)) => (! [VB] :
    ((element(VB)) => (! [VC] : ((element(VC)) => (((VA=VB)
    & (VB=VC)) => (VA=VC)))))))
Assume s1: (element(cA)) & ((element(cB)) & (element(cC)))
Assume s2: (cA=cB) & (cB=cC)
Prove  s3: cA=cC
PROVED by E Prover
--------------------------RESULT--------------------------
CORRECT
-------------------------STATISTICS-----------------------
Parsing time: 908.7 µs
Verifying time: 20.88 ms
Total: 21.79 ms
```

FIGURE 5.8: Exemplary run of `./elfe`

5.3 Web interface

The ELFE system can also be accessed via a web interface. As we see in Figure 5.9, a user can enter a proof in a text area. Above the text area are buttons to enter special characters used in the background libraries. Below the text area is a field for returning status information of the proof. This field will be filled after initiating the verification process with a click on "VERIFY".

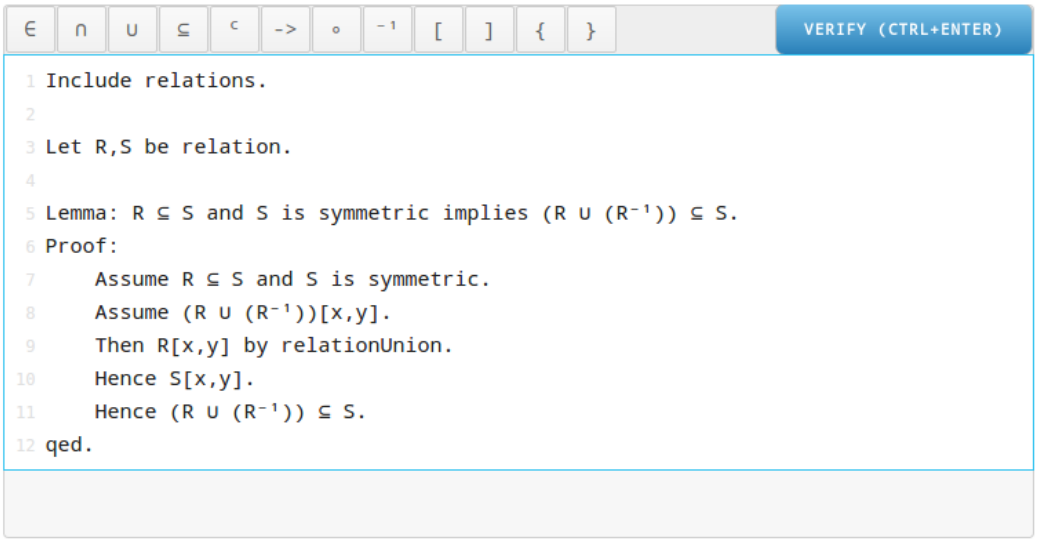


FIGURE 5.9: The web interface of ELFE

If the user made a syntax error as in Figure 5.10, the exact line and column of the error is prompted. Since we expect that the user wants to close the outer implication, we can suggest that he uses the correct language construct 'Hence'.

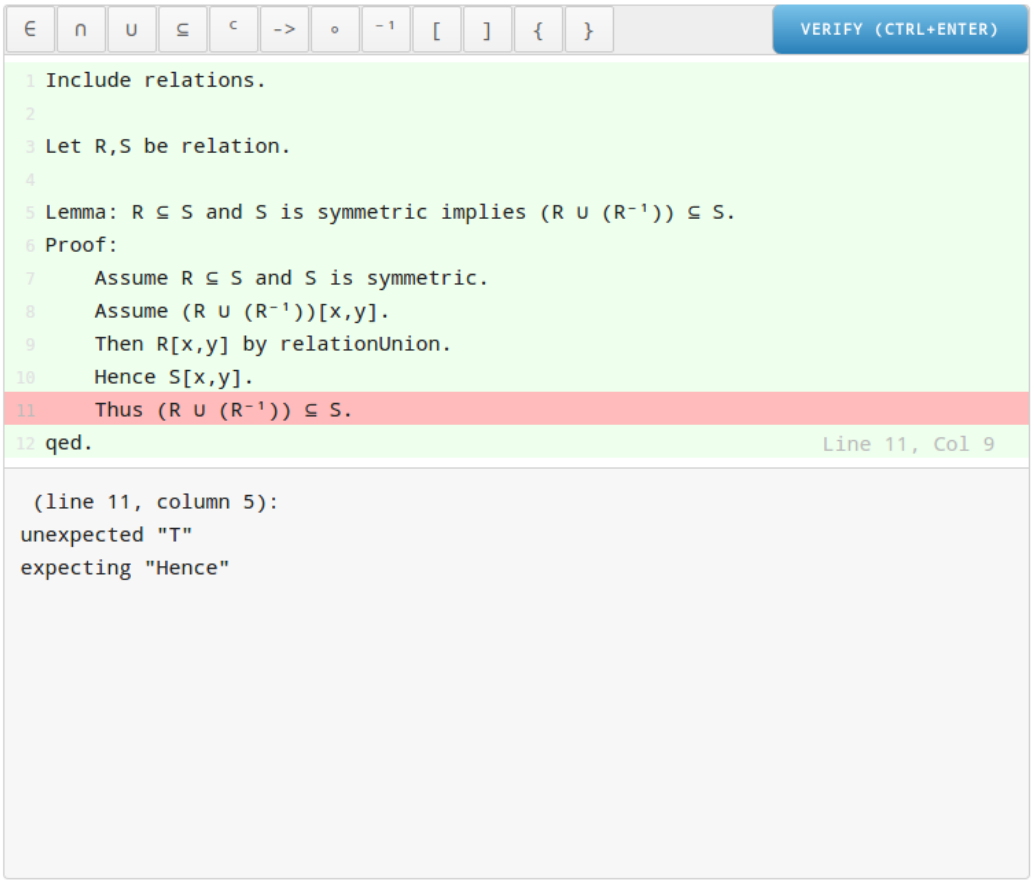


FIGURE 5.10: Parsing error

If the user made a wrong derivation, the system returns a countermodel if possible. In this example, BEAGLE found a countermodel for the derivation step made in line 9. Note that all variables in the countermodel are prefixed with c since we unfolded the universal quantifications and we see the raw forms of the predicates without any syntax sugaring.

BEAGLE suggests that there are relations R and S that meet the requirements of the lemma. Also, there are x and y that are in $R \cup (R^{-1})$ and particularly in R^{-1}, but not in R. This indicates that we misunderstood the union of relations: If a tuple is in the union of two relations, it is not necessarily in both relations.

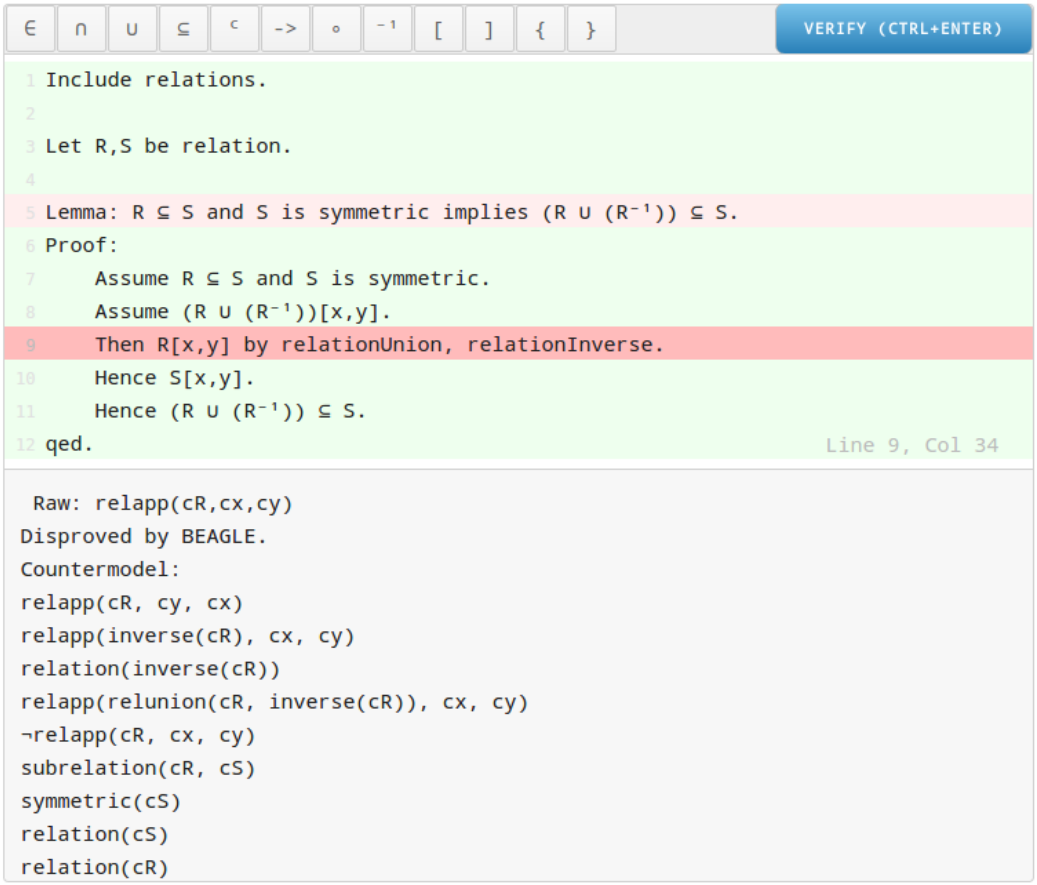


FIGURE 5.11: Countermodel for wrong proof

We can correct the mistake in the previous proof by making a case distinction as demonstrated in Figure 5.12. The text is now completely verified. By putting the cursor in different lines, we can inspect the internal representations of the formulas, e.g., in line 10 we can see the goal of the case distinction.

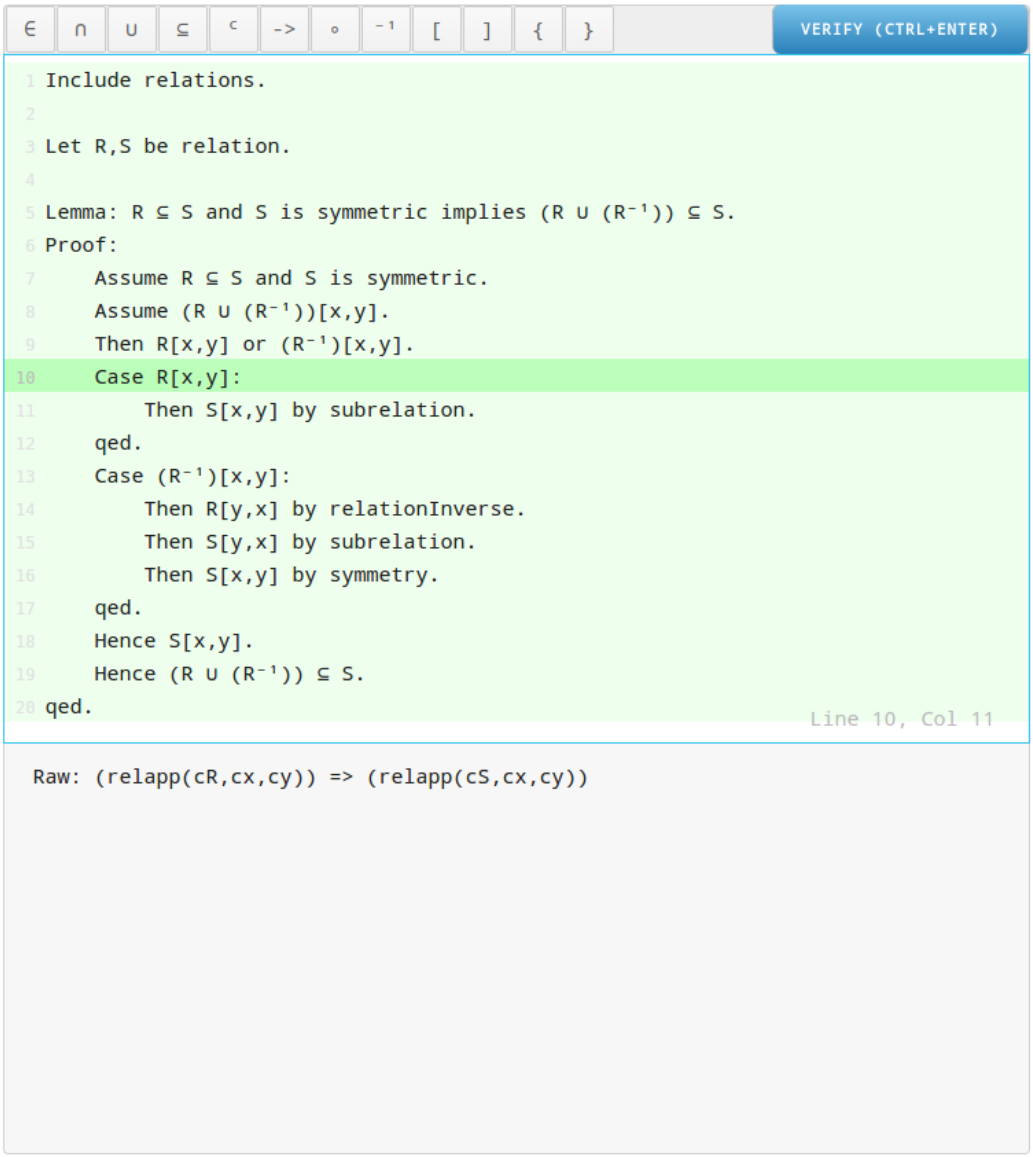


FIGURE 5.12: A verified proof

Chapter 6

Implementation of ELFE

The full implementation of ELFE consists of over 1000 lines of Haskell code and can be found online [1]. In the following we only want to take a look at crucial algorithms of the system.

We will use a notation close to the Haskell programming language. Cryptic function names will be replaced with more intuitive ones. We represent lists comma separated $l_1, ..., l_n$ and the empty list as $\emptyset$. A tuple of two elements x, y (respectively two types `A,B`) is denoted $x \times y$ (respectively `A` $\times$ `B`). Functions are annotated with their types where `[A]` is the type list of `A`. We also use the `do` notation which is syntactic sugar for programming with monads. The reader does not need further knowledge about monads except that parts enclosed by `do` are evaluated sequentially and the keyword `<-` assigns a value to a variable.

6.1 Parsing derivations

The parser is the largest part of the source code since all language constructs need to be parsed and the parser state administered. Most of it is straightforward, we will only take a look at the part that creates a proof for a lemma. The algorithm implements the language definition in Section 4.5. We omit the parts that create the abstract syntax tree (in particular language markers such as `'Proof:'`, `'Assume'`, `'qed.'` etc.) and focus on the structure of the algorithm.

The function `derivation` is very close to the language definition of <*derivation*>. All possible proving methods are tried to be applied successively. This is expressed by the choice function `<|>`. The formula to be proven will be referred as ϕ in the following, c_n are currently fixed variables, i.e., variables that are considered as constants. For example, to prove an universally quantified goal, a fixed but arbitrary element will be chosen in `unfold`. The user will call it the same element in his proof, but we need to tell the background provers that it is a constant. Thus, we keep track of fixed variables.

```
derive :: Formula -> [Variable] -> [Statement]
derive φ c_n = subproofs φ c_n
           <|> cases φ c_n
           <|> unfold φ c_n
           <|> infer φ c_n
           <|> extendContext φ c_n
           <|> finalProof φ c_n
```

The function subproofs constructs sub proofs as introduced in Section 4.5.1. First, the different sub goals and their proofs are saved in $S_1, ..., S_n$. This is done with the function proof. The call of formula corresponds to parsing the sub goal specified by the user. Then, proof recursively calls derive for the sub goal.

Back in subproofs, the statement S_0 contains the soundness criteria, i.e., checks if the sub goals indeed imply the original goal. The function vars2Cons replaces all fixed variables given in c_n with constants.

```
subproofs :: Formula -> [Variable] -> [Statement]
subproofs φ c_n = do
    S_1, ..., S_n <- many (proof c_n)
    S_0 <- Statement (vars2Cons (⋀ S_1, ..., S_n → φ) c_n) ByContext
    return Statement φ (BySplit S_0, ..., S_n)

proof :: [Variable] -> Statement
proof c_n = do
    ψ <- formula
    S_0, ..., S_n <- derive ψ
    return Statement ψ (BySequence S_0, ..., S_n)
```

To prove a formula in the manner of natural deduction as in Section 4.5.2, the function unfold performs pattern matching on the structure of the goal. A quantified formula may be derived for a fixed constant instead of a variable. Thus, the variable x is added to c_n as discussed above. Analogous to that, unfolding of an implication results in assuming the left hand side and deriving the right hand side of the implication. The unfoldings are of course only applied if the user gives the language markers 'Fix' respectively 'Assume' and 'Hence', we omit the markers here for readability.

```
unfold :: Formula -> [Variable] -> [Statement]
unfold (∀x.ψ) c_n = derive ψ (c_n, x)
unfold (ψ → ψ') c_n = do
    S_1 <- Statement (vars2Cons ψ c_n) Assumed
    S_2, ..., S_n <- derive ψ' c_n
    return S_1, ..., S_n
```

In order to infer if a user proves an alternative goal as introduced in Section 4.5.3, the infer function tries to build the derived formula ψ. The function lookAhead prevents the parser from actually consuming any input. In particular, inferImplies parses the given derivation in order to get to the conclusion of the assumption. The derivation is discarded, so that it can be built again by calling derive in infer and saving the derivation in $S_2, ..., S_n$. The statement S_0 checks soundness of the alternative goal, i.e., if the alternative goal ψ indeed implies the original goal ϕ.

```
infer :: Formula -> [Variable] -> [Statement]
infer φ c_n = do
    ψ <- lookAhead (inferImplies c_n)
    S_2, ..., S_n <- derive ψ c_n
    S_1 <- Statement (vars2Cons ψ c_n) (BySequence S_2, ..., S_n)
    S_0 <- Statement (vars2Cons (ψ → φ) c_n ByContext)
    return Statement φ (BySplit S_0, S_1)

inferImplies :: [Variable] -> Formula
inferImplies c_n = do
    ψ <- fof
    _ <- derive Top c_n
    ψ' <- fof
    return ψ → ψ'
```

As a last proving method, the user may extend the context by giving cornerstones to a proof as introduced in Section 4.5.4. Accordingly, the function `extendContext` calls `then` and `take` and adds their statements in the context of the actual derivation. Both functions return a tuple of a statement S_0 and constants c'_n. This is due to the fact that `take` fixes a variable x. Thus, it needs to be added to c'_n in the following derivation of ψ. Note that outside of the derivation, x is not fixed. Thus, `take` is a sound implementation of Lemma 4.7.

The function `then` simply creates a statement that is checked by the background provers. We omitted the implementation of <*by*> here to ease readability.

```
extendContext :: Formula -> [Variable] -> [Statement]
extendContext φ c_n = do
    S_0 × c'_n <- then c_n <|> take c_n
    S_1, ...S_n <- derive φ c'_n
    return Statement (vars2Cons φ c_n) (BySequence (S_0, ...S_n))

take :: [Variable] -> Statement × [Variable]
take c_n = do
    x <- variable
    ψ <- formula
    S_1 <- Statement (∃x.(vars2Cons ψ c_n)) ByContext)
    return (Statement (vars2Cons ψ (c_n, x)) (BySequence S_1) × x, c_n )

then :: [Variable] -> Statement × [Variable]
then c_n = do
    ψ <- formula
    return Statement (vars2Cons ψ c_n) ByContext
```

If the user gives no additional proving method, the remaining goal is given to the provers via `finalProof`.

```
finalProof φ c_n = Statement (vars2Cons φ c_n) ByContext
```

6.2 Verifying proof obligations

To check the correctness of statements annotated with BYCONTEXT and BYSUBCONTEXT, several background provers are invoked by calling `verify` with the goal formula and the context. All background provers are called concurrently and are terminated as soon as the first prover finds a result. Some of the background provers provide a countermodel if an obligation is incorrect. This is depicted here by the second element of the tuple returned by `runProcess`. If no prover finds a proof or countermodel, the `timeout` thread will trigger termination of all threads. We assume that the background provers work correctly and trust their results.

We use the semaphore `done` to synchronize termination. Additionally, the channel `result` is used to communicate the result. Constants from the configuration as `TIMEOUT` and `PROVERS` are denoted uppercase. In the actual implementation, `createTask` creates a file containing the TPTP task as introduced in Section 2.3.1.

```
verify :: Formula -> Context -> Result
verify φ Γ = do
    done <- newSemaphore
    result <- newChannel
    task <- createTask φ Γ
    threads <- createThreads (prove task prover result done) PROVERS
    timeoutThread <- createThread (timeout result done)
    readSemaphore done
    result <- readChannel result
    killThreads timeoutThread, threads
    return result

prove :: Task -> Prover -> Channel Result -> Semaphore Bool
prove task prover result done = do
    res × countermodel <- runProcess prover task
    if res == pos
        writeChannel result Correct
        putSemaphore done True
    else if res == neg
        writeChannel result Incorrect countermodel
        putSemaphore done True

timeout :: Channel Result -> Semaphore Bool
timeout result done = do
    threadDelay TIMEOUT
    writeChannel result Unknown
    putSemaphore done True
```

6.3 Verifying a statement sequence

With the algorithm introduced in Section 6.2, one can check if a formula follows from a context. With that, we can check the correctness of statements and sequences as

given in Definition 4.5. As we recall, statements annotated by ASSUMED are considered correct. Statements annotated with BYCONTEXT and BYSUBCONTEXT will be sent to the background provers. To check correctness of statements annotated with BYSEQUENCE and BYSPLIT, the correctness of all of their children will be taken into account.

```
verifyStat :: Statement -> Context -> Result
verifyStat (Statement φ Assumed)                    Γ = Correct
verifyStat (Statement φ ByContext)                  Γ = verify φ Γ
verifyStat (Statement φ (BySubContext Id1,...,Idn)) Γ = verify φ Γ_{Id1,...,Idn}
verifyStat (Statement φ (BySequence S1,...,Sn))     Γ = verifySeq S1,...,Sn Γ
verifyStat (Statement φ (BySplit S1,...,Sn))     Γ = verifySplit S1,...,Sn Γ
```

As soon as one of the statements in the derivation of a statement proven BYSPLIT or BYSEQUENCE could not be proven by the background provers, the statement will be considered `Incorrect`. This is a simplification of the implemented algorithm where all partial results will be saved in an appropriate data structure. We omit this part here. Once a statement is checked, it adds to the context in BYSEQUENCE, while the contexts for the children in BYSPLIT are distinct. This corresponds to Definition 4.3 where we introduced contexts as only containing statements before and above the current statement.

```
verifySeq :: [Statement] -> Context -> Result
verifySeq ∅ Γ = Correct
verifySeq S1,...,Sn Γ = do
    status <- verifyStat S1 Γ
    if status /= Correct
        return Incorrect
    else
        verifySeq S2,...,Sn (Γ, S1)

verifySplit :: [Statement] -> Context -> Result
verifySplit ∅ Γ = Correct
verifySplit S1,...,Sn Γ = do
    status <- verifyStat S1 Γ
    if status /= Correct
        return Incorrect
    else
        verifySplit S2,...,Sn Γ
```

Chapter 7

Evaluation of ELFE

ELFE is still in a prototypical state. It was tested by students of Imperial College London. In Section 7.2, we will take a look at their evaluation and suggestions. Since I already used the system thoroughly, I will present my experiences in Section 7.1.

7.1 Limits of the system

Since first-order logic is an intuitive way to write down proofs in set theory and relations, proofs in these domains could be written down easily. Working with the functions library was more complex. Especially functions like 'funApp' which were introduced to make a proof more readable for humans increases the difficulty for the background provers. Thus, it is often hard to assess if a proof itself is wrong or only takes a long time to prove. Debugging a failing proof is still difficult with the user interface provided by ELFE. In most cases, I had to look at the TPTP tasks given to the background provers and manually delete and change the given premises to find out where I went wrong.

BEAGLE was able to provide countermodels to a wrong proof only in a few cases. Restricting the context of a derivation step increased the success rate significantly. Since I am familiar with the operating principle of ELFE, I was then able to understand the countermodel. For new users it is certainly difficult to relate a countermodel to the entered text.

The 'Let' construction was useful to shorten definitions and proofs. However, this construction hides what is going on under the hood. This sometimes can be confusing. The notations have turned out to be a very powerful construct to ease the readability of proofs. New notations can be introduced easily and make a proof look quite intuitive.

As we see in Figure 7.1, most of the proofs could be verified quite fast. The parser has a longer runtime if many syntactic sugars are introduced. The performance of the parser certainly can be improved. The verification time took in particular long if there were many premises, e.g., if we used the functions library.

Name of the proof	Parsing	Verifying	Total
Complement of a complement of a set	23.9 ms	357.0 ms	380.9 ms
Injective composition	276.1 ms	9912.0 ms	10190.0 ms
Composition of function and its inverse	426.0 ms	661.7 ms	1088.0 ms
Transitive, symmetric and bound relation	28.9 ms	186.4 ms	215.4 ms
Union of relation and its inverse	20.7 ms	592.8 ms	613.5 ms
Set complement and union distribution	40.5 ms	1527.0 ms	1568.0 ms
Set union and intersection distribution	32.5 ms	4672.0 ms	4705.0 ms
Subrelation and symmetry	23.1 ms	557.8 ms	580.9 ms
Inverse and complement of a relation	42.3 ms	671.8 ms	714.1 ms
Cantor's theorem	321.3 ms	4041.0 ms	4362.0 ms
Knaster-Tarski theorem	4 500.0 ms	31 540.0 ms	36 040.0 ms

FIGURE 7.1: Benchmarking the implemented proofs

Probably the systems biggest limitation is the fact that it maps into first-order logic. Some domains like set theory are fairly well axiomatizable within ELFE, while some properties like well-foundedness are not expressible at all.

Another problem that occurred was that the background provers were too clever. They sometimes find intermediate steps that are not at all obvious for a human reader. This is in particular problematic with proofs by contradiction. If the background provers find the inconsistency caused by the assumption, all derivations a user may make are trivially also true, even though they do not make sense in the proof.

7.2 User feedback

The system was tested with 12 students of Computing and Electrical Engineering. At first, they were given the proof in Text 7.2. All were able to identify the proof pattern and complete the proof. Later on, they were given more complex proofs. Some of them were able to complete them. Once they gained an understanding of the proof, most of them were able to formalize it in the system.

```
Include sets.
Let A be set.
Let x be element.
Lemma: ((A^C)^C) = A.
Proof:
    Proof ((A^C)^C) ⊆ A:
        Assume x ∈ ((A^C)^C).
        Then not x ∈ (A^C).
        Hence x ∈ A.
    qed.
    Proof A ⊆ ((A^C)^C):
    qed.
qed.
```

TEXT 7.2: To completed proof by students

After they tried the system, they were given the following statements and had to indicate with 1 (strongly agree) to 6 (strongly disagree) their agreement with the statements.

- *I enjoy writing mathematical proofs.*
  Mean: 3.3 – Median: 3,5
- *I find writing mathematical proofs difficult.*
  Mean: 2.6 – Median: 2
- *I think computers can be of use in learning how to write mathematical proofs.*
  Mean: 2.3 – Median: 2
- *I enjoyed writing mathematical proofs in the* ELFE *system.*
  Mean: 2.5 – Median: 2
- *I found the feedback of the system helpful.*
  Mean: 2.6 – Median: 2
- *I would like to know how* ELFE *and interactive theorem proving works.*
  Mean: 1.8 – Median: 1

As we see, the testers were not especially keen on writing mathematical proofs. Writing proofs inside the system made it a bit more enjoyable. The system seems to have succeeded in waking interest for interactive theorem proving.

In text form, they could also write down what they liked about the system and what should be improved. It was highlighted that the language was "simple and clear" and did not "get in the way of the proof". They liked the "very understandable and simple UI" and their reactiveness. As improvements for the user interface they proposed autocompletion features of the proofs and syntax highlighting. The given raw translations of the mathematical text were not easy to understand. One user also pointed out that the background provers are sometimes too clever – thus, a text is accepted even if crucial cornerstones of a proof are missing. He would like to have a criteria on when a proof is "complete" for human and not only for a computer.

Chapter 8

Related work

In the following we will give an overview of current approaches to interactive theorem proving. In Section 8.1, we will take a look at mathematical text verifiers like the SYSTEM FOR AUTOMATED DEDUCTION, which heavily influences this project. In Section 8.2, we will present two of the most popular interactive theorem provers, ISABELLE and COQ. Finally, we will take a look at a new development in Section 8.3, automated theorem provers for typed higher-order logic.

8.1 Mathematical text verifier

In the following, we will present two projects aimed for verifying mathematical texts: The SYSTEM FOR AUTOMATED DEDUCTION (SAD) in Section 8.1.1 and NAPROCHE in Section 8.1.2. Strictly speaking, these are also interactive theorem provers since they build a bridge between automatic theorem proving and human mathematicians. However, the explicit goal of these projects was to verify natural-sounding mathematical proofs. The technical part of the verification process should be hidden from the users.

8.1.1 SYSTEM FOR AUTOMATED DEDUCTION

The SAD was developed at the University Paris and the Taras Shevchenko National University of Kyiv. It continues the project "Algoritm Ochevidnosti" (algorithm of obviousness) which was initiated by the soviet researcher Victor Glushkov in the 1960s [27]. His purpose was to develop a tool that shortens long but "obvious" proofs to users. These omitted parts should be verified by automated theorem provers. The early stages of such a machine should already be used for teaching students since "to understand a proof means to be able to explain it to a machine that is operating with a relatively unsophisticated algorithm". [13, p.111]

SAD uses the input language FORTHEL. It is possible to express complex mathematical statements in a quite natural way. FORTHEL texts are converted to an ordered set of first-order formulas. The structure of the initial text is preserved such that necessary proof tasks can be defined. These tasks are then given to an ATP. The internal reasoner may simplify tasks and omit trivial statements. Afterwards, the verification status of the text is given to the user. For each proof task, the result of the used ATP is returned. This allows to inspect possible sources of failing tasks, but requires knowledge of how the background provers work. [28]

Consider the example in Text 8.1. This FORTHEL text formalizes the lemma proved by ELFE in Text 4.17. We see strong similarities since our system was based on FORTHEL. In the first line, it is introduced that singular and plural tokens both describe the same elements. Then, the predicates 'element' and 'relation' are introduced to the parser. These predicates hold for all elements – in contrast to the 'Let' construction as introduced in Section 4.6.1 which only types certain elements. However, it is still necessary to identify x, y, z as elements and R as relations so that the parser will recognize these tokens. The 'Signature' command is similar to our 'Notation' command from Section 4.4.1 and introduces syntactic sugar to the parser.

To prove the lemma, SAD detects that we want to unfold the definition 'DefRef'. The syntax is not as fixed as in ELFE where 'Then' and 'Hence' have distinct meanings in different proving techniques. In SAD, they can be used interchangeably. However, the parser still recognizes that the user wants to show an implication. The proof then is analogous to our proof.

```
[element/-s] [relation/-s]

Signature ElmSort. An element is a notion.
Signature RelSort. A relation is a notion.

Let x,y,z denote elements.
Let R denote relations.

Signature RelApp. R[x,y] is an atom.

Definition DefSym. R is symmetric iff for all x,y : R[x,y] => R[y,x].
Definition DefBou. R is bound iff for all x : exists y : R[x, y].
Definition DefTrans. R is transitive iff for all x,y,z : (R[x,y] or R[y,z]) => R[x,z].
Definition DefRef. R is reflexive iff for all x : R[x, x].

Proposition.
    Let R be a transitive symmetric bound relation. Then R is reflexive.
Proof.
    Let x be an element.
    Take y such that R[x,y].
    Then R[y,x].
    Then R[x,x].
qed.
```

TEXT 8.1: FORTHEL text about relations

8.1.2 NAPROCHE

The NAPROCHE system was a joined project between mathematicians at the University of Bonn and linguists at the University of Duisburg-Essen. Its central goal was to develop a controlled natural language (CNL) which checks semi-formal mathematical texts. The input are texts in a LATEX style language, consisting of mathematical formulas embedded in a controlled natural language. The semantics of PRS have

been researched extensively, however, the project is not continued and has no working version online. [17]

To extract the semantics of a CNL text, NAPROCHE adapts a concept from computational linguistics: Proof Representation Structures (PRS) enrich the linguistic concept of Discourse Representation Structures in such a way that they can represent mathematical statements and their relations. Consider the example in Figure 8.2. The PRS describes a `proof_1` which starts with the observation that there is no element in the empty set. The inner PRS with the identifier `0` links with `Drefs` to another discourse referent `1`, i.e., the outer PRS. The mathematical referent after `Mrefs` is the actual statements of this referent, i.e., the first-order formula that there is no set in the empty set. In order to verify the PRS, one has to check the conditions in `Conds`, i.e., that there is indeed an empty set. The user can then proceed with the rest of the proof in `consec_2`.

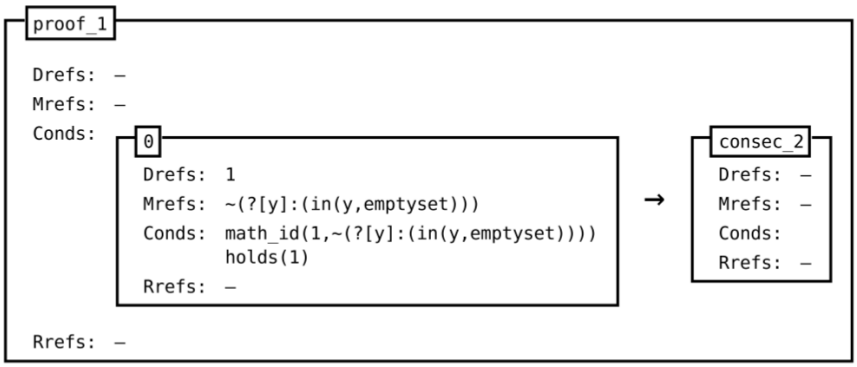


FIGURE 8.2: Proof Representation Structure [8]

Even though our construction of statement sequences was driven by thinking in data-structures and less linguistically, it was certainly influenced by NAPROCHE.

8.2 Interactive theorem prover

The classical approach to interactive theorem proving integrates a human user strongly in the technical verification process. We will present the popular provers ISABELLE in Section 8.2.1 and COQ in Section 8.2.2 here.

We compare our system with ISABELLE and COQ in the following with respect to a proof of Cantor's theorem. The theorem states that there exists no surjective function f from a set A to its powerset $\mathcal{P}(A)$. It uses a diagonalization argument, i.e, constructs the subset M of $\mathcal{P}(A)$ with $M := \{x \in A \mid x \notin f(x)\}$, which leads to a contradiction.

Proving Cantor's theorem in ELFE as shown in Text 8.3 is not straightforward. We experimented with offering a possibility to write set comprehensions, however, this requires to transform all predicates used as set conditions to terms. Instead, we had to introduce the diagonalization of A as a separate definition. Apart from that definition, the proof is quite intuitive: First, we make a case distinction on whether 'A' is empty. If it is, the powerset still contains the empty set and thus, it is not possible

to construct a surjective function.

To prove the theorem if 'A' is not empty, we save the diagonalization of 'A' in 'M'. Since 'M' is in the powerset of 'A' and we assumed 'f' to be surjective, we can obtain an 'a' that maps to 'M'. This leads to the contradiction that 'a' is both in 'M' and not in 'M'. To make this contradiction obvious, we made a case distinction.

```
Include functions.

Let A be set.

Definition: for all f. f: A -> (A℘) implies diagonalized(A) is set and (for all x.
    x ∈ diagonalized(A) iff (x ∈ A and (for all y ∈ (A℘). f[x,y] implies not x ∈ y))).


Lemma: for all f. f: A -> (A℘) implies f is not surjective.
Proof:
    Assume exists f. f: A -> (A℘) and f is surjective.
    Case A is empty:
        Then A℘ is nonempty.
        Then contradiction.
    qed.
    Case A is nonempty:
        Take M such that M = diagonalized(A).
        Then M ∈ (A℘).
        Take a such that a ∈ A and f[a,M].
        Case a ∈ M:
            Then not a ∈ M.
            Then contradiction.
        qed.
        Case not a ∈ M:
            Then a ∈ M.
            Then contradiction.
        qed.
    qed.
    Hence contradiction.
qed.
```

TEXT 8.3: ELFE proof of Cantor's theorem

8.2.1 ISABELLE

ISABELLE is a joined project of the Cambridge University and the TU Munich. It supports polymorphic higher-order logic, augmented with axiomatic type classes. It is designed for interactive reasoning in a variety of formal theories. At present it provides useful proof procedures for Constructive Type Theory , various first-order logics, Zermelo-Fraenkel set theory, and higher-order logic. [22]

Canthor's theorem can be proved in ISABELLE straightforwardly as we see in Text 8.4. Similar to our proof in ELFE, we make a proof by contradiction. The diagonalization of 'A' can be defined directly since ISABELLE works with higher-order logic.

Analogously to our proof in ELFE, we can obtain an 'a' that maps to 'M' and conclude the contradiction. We have to explicitly tell which automated theorem proving technique should be used to derive these conclusions. The technique 'auto' works just with logical simplifications, i.e., term rewriting. The tactic 'blast' calls a tableau prover.

```
theory Cantor
    imports Main
begin

theorem Cantor: "∄f :: 'a ⇒ 'a set. ∀A. ∃x. A = f x"
proof
    assume "∃f :: 'a ⇒ 'a set. ∀A. ∃x. A = f x"
    then obtain f :: "'a ⇒ 'a set" where *: "∀A. ∃x. A = f x" ..
    let ?M = "x. x ∉ f x"
    from * obtain a where "?M = f a" by auto
    moreover have "a ∈ ?M ⟷ a ∉ f a" by auto
    ultimately show False by blast
qed
```

TEXT 8.4: Cantors theorem with ISABELLE

SLEDGEHAMMER

Isabelle has several built-in proof techniques which need to be called explicitly. In 2007 the extension SLEDGEHAMMER was developed. SLEDGEHAMMER calls in parallel several ATP like E PROVER, SPASS and VAMPIRE. Since higher-order logic is not in general reducible to first-order logic, this requires many unsound derivations. Thus, ISABELLE will only extract which of the premises were used by the ATP and then try to reconstruct the proof with its own proving techniques. [21]

SLEDGEHAMMER requires no further configuration and is called by mouse-click. In a recent study, 34% of nontrivial goals contained in representative ISABELLE texts could be proved by SLEDGEHAMMER. With this extension, ISABELLE allows also beginners to prove challenging theorems. The creators note that SLEDGEHAMMER was not designed as a tool to teach ISABELLE. Instead, it was focused primarily on experienced users. However, they realized that it changed the way ISABELLE is taught. Beginners do not have to learn about low level proving tactics and how they work – it is not straightforward to get an intuition which tactics may be suited for which problem without deeper knowledge of automated theorem proving. Instead, they can focus on the proof from a higher level. Additionally, SLEDGEHAMMER finds which lemmas are relevant to a proof. This dispenses the necessity to memorize lemmas of background libraries. [21] In fact, all proving tactics used in Text 8.4 can be found quickly by using SLEDGEHAMMER.

8.2.2 COQ

COQ is an interactive theorem prover initially developed 1984 at INRIA. It is based on the Curry–Howard correspondence which relates types to classical logic. COQ

was used in proving the four color theorem. [14]

Text 8.5 shows the proof of Cantor's theorem in COQ. Since we have types at our disposal, we can easily define surjective functions. In the first line of the proof, we simply assume that there exists a 'f' that meets the assumptions, i.e., is surjective. The diagonalization is then saved within the function 'g'. Afterwards, we introduce the additional assumption 'B' which fixes a 'x' from the image of 'f'. Then we introduce the assumption 'C' which states that 'g x' and 'f x x' are equal. This is proven by rewriting 'B'. Finally, we unfold the definition of 'g' which leads to the contradiction 'f x x <-> not f x x '.

```
Set Implicit Arguments.
Unset Strict Implict.
Require Import
Require Import List.
Import ListNotations.

Definition surjective (X Y : Type) (f : X -> Y) : Prop := forall y, exist x, f x = y.

Theorem Cantor X : not exists f : X -> X -> Prop, surjective f.
Proof.
    intros [f A].
    pose (g := fun x => not f x x).
    destruct (A g) as [x B].
    assert (C:g x <-> f x x).
    {
        rewrite B. tauto.
    }
    unfold g in C. tauto.
Qed.
```

TEXT 8.5: Cantors theorem with COQ

8.3 Higher order automatic theorem prover

The TPTP syntax is a widely accepted standard and made it possible to easily implement ATP in interactive theorem provers. Since 2008 the new syntax version THF has been developed. Its goal introduce a language standard for higher-order logic based on Church's simple type theory. [5]

Consider the exemplary THF text in Figure 8.6 which proves the associativity of the union of sets. From the two basic types `$i` and `$o` a user can construct other types with the type constructor >. In this example, elements of a set have the type `$i` and sets the type `$i > $o`. Function application can be done with `@`. In the conjecture, associativity is proven for all sets `A`, `B` and `C`.

```
thf(in_type,type,(in: $i > ( $i > $o ) > $o )).
thf(union_type,type,(union: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).
thf(in,axiom,((in = (^ [X: $i,S: ($i > $o)] : (S @ X))))).
thf(union,axiom,((
    union = (^ [S1: ($i > $o),S2: ($i > $o),U: $i] :
        ((in @ U @ S1) | (in @ U @ S2)))))
).
thf(union_distribution,conjecture,(
    ! [A: ($i > $o),B: ($i > $o),C: ($i > $o)] :
        ((union @ (union @ A @ B) @ C) = (union @ A @ (union @ B @ C))))
).
```

FIGURE 8.6: Exemplary THF file [5]

The THF syntax is already implemented in several automated theorem provers. LEO for example is currently developed at the Free University of Berlin. It is a self-contained higher-order theorem prover based on ordered paramodulation and superposition. The system attempts to remove higher-order features from the problem so that it can be solved efficiently by first-order provers, but also has a own higher-order reasoning calculus. LEO has been integrated in ISABELLE. [6]

Chapter 9

Conclusion

In the following, we will first summarise our work in Section 9.1. There are numerous ways to extend and improve ELFE in the future. We will take a look at these in Section 9.2.

9.1 Deliverables

In this work, we developed statement sequences for representing mathematical proofs. This data structure acts as a powerful intermediate language between mathematical texts and ATP. The web interface provides an intuitive way to interact with the system. Together with the background library, this allows for quickly writing proofs.

The big goal of this work was to detach the users of interactive theorem provers from the technicalities of ATP. While this is certainly not suitable for all use cases, it is desirable if we want to use computers in teaching mathematics. This work provides a proof of concept that this is feasible and sensible.

9.2 Future work

Given the limited time frame of this work, there are numerous ways to extend the system. In the following, we will first take a look at iterative improvements and extensions to the system in Section 9.2.1 and then give an outlook on a conceptually revised version of the tool in Section 9.2.2.

9.2.1 Improvements and extensions

The language constructs presented here were the result of formalizing several exemplary proofs. If one formalizes more proofs, he will probably feel the need for additional proving methods. If one can map the proving methods soundly into statement sequences, this should be easy to implement. Students often prove problems in discrete mathematics with induction. This is not directly realisable in ELFE since well-foundedness cannot be expressed in first-order logic. An interesting approach could be to transform the relevant predicates and terms into an TRS. Termination of TRS is extensively researched and tools such as APROVE [11] could be queried in the background to show the soundness of proofs by induction.

So far, arithmetic proofs could be formalized in ELFE. However, it is tedious to define the standard interpretation of arithmetic. The background prover Z3 and BEAGLE already provide background theories for arithmetic. Evidently, it would be good to

use these features. In order to do this, additional parsing of integers and arithmetic operations needs to be implemented.

The countermodels found by BEAGLE are currently presented in raw form to the user. Another extension could aim to link this countermodel more directly to the mathematical text to ease understanding. Additionally, the different automated theorem provers provide other hints that could be of use for user. The used input formulas could be given to point out which premises were relevant to a proof. The derivation sequence as given by E PROVER or Z3 could be processed such that it is possible for a user to retrace the proofs. If one does not understand why a proof works, this could introduce intermediary steps in the proof.

Some mathematical statements cannot be expressed in first-order logic directly, but in higher-order logic that is reducible to first-order logic. Set comprehensions and properties of predicates like symmetry could be introduced with additional language constructs and internally transformed into first-order logic.

As the background theories grow, we probably need more features like namespacing. To increase the performance of the background provers, internal premise selection could be implemented.

For the user interface, features like autocompletion of proof structures and syntax highlighting could ease the access to the system.

9.2.2 ELFE with higher-order ATP

As we introduced in Section 8.3, the last years have seen interesting advances in automated proving of higher-order logic. Since first-order has certain limitations, it might be good to utilize these provers. THF provides a comfortable interface to the provers.

Higher order logic allows for expressing many more domains in a natural way. A strong type system allows to detect insensible mathematical texts before giving them to background provers. Thus, it would be interesting to extend ELFE by a type system. This extension requires a rewrite of the core of the system. It would be to examine if statement sequence still are a sensible representation of the proof obligations.

With that, the system may become as powerful as ISABELLE or COQ. In contrast to these provers, the technical part of ATP could be abstracted away. Theorem provers then could become a valuable tool in teaching mathematics.

Bibliography

[1] *ELFE - Interactive Theorem Proving for students*, `https://github.com/maxdore/elfe`, Accessed: 2017-06-18.

[2] *SPASS History*, `http://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/history/`, Accessed: 2017-06-18.

[3] L. Bachmair and H. Ganzinger, *Rewrite-based equational theorem proving with selection and simplification*, Journal of Logic and Computation, 4(3):217–247, 1994.

[4] P. Baumgartner, J. Bax and U. Waldmann, *Beagle – A Hierarchic Superposition Theorem Prover*, Proc. 25th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, 9195:367–377, 2015.

[5] C. Benzmüller, F. Rabe and G. Sutcliffe, *THF0 – The Core of the TPTP Language for Higher-Order Logic*, Proc. Automated Reasoning: 4th International Joint Conference, Lecture Notes in Artificial Intelligence, 5195:491–506, 2008.

[6] C. Benzmüller, N. Sultana, L. Paulson, F. Theiß, *The Higher-Order Prover Leo-II* Journal of Automated Reasoning, 55(4):389–404, 2015.

[7] C. Chang and R. Char-Tung Lee, *Symbolic Logic and Mechanical Theorem Proving*, Elsevier, 1973.

[8] M. Cramer, *Mathematisch-logische Aspekte von Beweisreprasentationsstrukturen*(In German), 2009.

[9] M. Fitting, *First-Order Logic and Automated Theorem Proving*, 2nd Edition, Springer, 1996.

[10] G. Gentzen, *Untersuchungen über das logische Schließen*(in German), Mathematische Zeitschrift, 39:176–210, 1935.

[11] J. Giesl, P. Schneider-Kamp and R. Thiemann, *AProVE 1.2: Automatic termination proofs in the dependency pair framework*, Proc. Automated Reasoning: Third International Joint Conference, Lecture Notes in Computer Science, 4130:281–286, 2006.

[12] J. Giesl, *Termersetzungssysteme*(in German), 2011.

[13] V. Glushkov, *Problems in the Theory of Automata and Artificial Intelligence*, Journal of Cybernetics, 1(1):97–113, 1971.

[14] G. Gonthier, *Formal proof–the four-color theorem*, Notices of the American Mathematical Society, 55(11):1382–1393, 2008.

[15] E. Grädel, *Mathematische Logik*(in German), 2016.

[16] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, 2009.

[17] D. Kühlwein, M. Cramer, P. Koepke and B. Schröder, *The Naproche system*, Proc. 18th Symposium of Intelligent Computer Mathematics, Lecture Notes in Artificial Intelligence, 6824:180–195, 2009.

[18] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, 2nd edition, Springer, 2008.

[19] L. de Moura, N. Bjørner, *Z3: An Efficient SMT Solver*, Lecture Notes in Computer Science, 4963:337–340, 2008.

[20] A. Newell and H. Simon, *The logic theory machine–A complex information processing system*, Information Theory, 2(3):61–79, 1956.

[21] L. Paulson and J. Blanchette, *Three years of experience with Sledgehammer: A Practical Link Between Automatic and Interactive Theorem Provers*, Proc. 8th International Workshop on the Implementation of Logics, 2015.

[22] L. Paulson, *Isabelle: The next 700 theorem provers*, Logic and computer science, 31:361–386, 1990.

[23] A. Riazanov and A. Voronkov, *The design and implementation of VAMPIRE*, AI Communications, 15(2,3):91–110, 2002.

[24] G. Robinson and L. Wos, *Paramodulation and Theorem-Proving in First-Order Theories with Equality*, Automation of Reasoning, 2:298–313, 1983.

[25] G. Sutcliffe, *The TPTP Problem Library and Associated Infrastructure*, Journal of Automated Reasoning, 43(4):337–362, 2009.

[26] S. Schulz, *E - a brainiac theorem prover*, AI Communications, 15(2,3):111–126, 2002.

[27] K. Vershinin and A. Paskevich, *ForTheL — the language of formal theories*, IJ Information Theories and Applications, 7:121–127, 2000.

[28] K. Verchinine, A. Lyaletski and A. Paskevich, *System for Automated Deduction(SAD): A Tool for Proof Verification*, Proc. 21st International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, 4603:398–403, 2007.

[29] C. Weidenbach, *Combining Superposition, Sorts and Splitting*, Handbook of Automated Reasoning, Elsevier, 1965–2013, 2001.

Appendix A

Running proofs

Complement of a complement of a set

Include sets.

Let A be set.

Lemma: $((A^C)^C) = A$.
Proof:
    Proof $((A^C)^C) \subseteq A$:
        Assume $x \in ((A^C)^C)$.
        Then not $x \in (A^C)$.
        Hence $x \in A$.
    qed.
    Proof $A \subseteq ((A^C)^C)$:
        Assume $x \in A$.
        Then not $x \in (A^C)$.
        Hence $x \in ((A^C)^C)$.
    qed.
qed.

Injective composition

Include functions.

Let A,B,C be set.

Let f: A -> B.
Let g: B -> C.

Lemma: $g \circ f$ is injective implies f is injective.
Proof:
    Assume $g \circ f$ is injective.
    Assume $x1 \in A$ and $x2 \in A$ and $(f\{x1\}) = (f\{x2\})$.
    Then $((g \circ f)\{x1\}) = ((g \circ f)\{x2\})$.
    Hence $x1 = x2$.
    Hence f is injective.
qed.

Composition of function and its inverse

Include functions.

Let A,B be set.
Let f: A -> B.

Lemma: f is bijective implies $(f^{-1})\circ f$ is identity.
Proof:
    Assume f is bijective.
    Assume $x \in A$.
    Take y such that $y \in B$ and f[x,y] by function.
    Then $(f^{-1})[y,x]$ by inverse.
    Then $((f^{-1})\circ f)[x,x]$ by inverse, composition.
    Hence $((f^{-1})\circ f)[x,x]$.
    Hence $(f^{-1})\circ f$ is identity.
qed.

Transitive, symmetric and bound relation

Include relations.

Let R be relation.

Lemma: transitive(R) and symmetric(R) and bound(R) implies R is reflexive.
Proof:
    Assume transitive(R) and symmetric(R) and bound(R).
    Proof for all x. R[x,x]:
        Fix x.
        Take y such that R[x,y] by boundness.
        Then R[y,x] by symmetry.
        Then R[x,x].
    qed.
    Hence R is reflexive.
qed.

Union of relation and its inverse

Include relations.

Let R be relation.

Lemma: $R \cup (R^{-1})$ is symmetric.
Proof:
    Assume $(R \cup (R^{-1}))[x,y]$.
    Then $R[x,y]$ or $(R^{-1})[x,y]$.
    Case $R[x,y]$:
        Then $(R^{-1})[y,x]$.
    qed.
    Case $(R^{-1})[x,y]$:
        Then $R[y,x]$.
    qed.
    Hence $(R \cup (R^{-1}))[y,x]$.
qed.

Set complement and union distribution

Include sets.

Let A,B be set.

Lemma: $((A \cup B)^C) = ((A^C) \cap (B^C))$.
Proof:
    Proof $((A \cup B)^C) \subseteq ((A^C) \cap (B^C))$:
        Assume $x \in ((A \cup B)^C)$.
        Then not $x \in (A \cup B)$.
        Then not $x \in A$ and not $x \in B$.
        Then $x \in (A^C)$ and $x \in (B^C)$.
        Hence $x \in ((A^C) \cap (B^C))$.
    qed.
    Proof $((A^C) \cap (B^C)) \subseteq ((A \cup B)^C)$:
        Assume $x \in ((A^C) \cap (B^C))$.
        Then $x \in (A^C)$ and $x \in (B^C)$.
        Then not $x \in A$ and not $x \in B$.
        Then not $x \in (A \cup B)$.
        Hence $x \in ((A \cup B)^C)$.
    qed.
qed.

Set union and intersection distribution

Include sets.

Let A,B,C be set.

Lemma: $(A \cap (B \cup C)) = ((A \cap B) \cup (A \cap C))$.
Proof:
    Proof $(A \cap (B \cup C)) \subseteq ((A \cap B) \cup (A \cap C))$:
        Assume $x \in (A \cap (B \cup C))$.
        Then $x \in A$ and $x \in (B \cup C)$.
        Then $x \in B$ or $x \in C$.
        Then $x \in (A \cap B)$ or $x \in (A \cap C)$.
        Hence $x \in ((A \cap B) \cup (A \cap C))$.
    qed.
    Proof $((A \cap B) \cup (A \cap C)) \subseteq (A \cap (B \cup C))$:
        Assume $x \in ((A \cap B) \cup (A \cap C))$.
        Then $x \in (A \cap B)$ or $x \in (A \cap C)$.
        Then ($x \in A$ and $x \in B$) or ($x \in A$ and $x \in C$).
        Then $x \in A$ and ($x \in B$ or $x \in C$).
        Hence $x \in (A \cap (B \cup C))$.
    qed.
qed.

Subrelation and symmetry

Include relations.

Let R,S be relation.

Lemma: $R \subseteq S$ and S is symmetric implies $(R \cup (R^{-1})) \subseteq S$.
Proof:
    Assume $R \subseteq S$ and S is symmetric.
    Assume $(R \cup (R^{-1}))[x,y]$.
    Then $R[x,y]$ or $(R^{-1})[x,y]$.
    Case $R[x,y]$:
        Then $S[x,y]$ by subrelation.
    qed.
    Case $(R^{-1})[x,y]$:
        Then $R[y,x]$ by relationInverse.
        Then $S[y,x]$ by subrelation.
        Then $S[x,y]$ by symmetry.
    qed.
    Hence $S[x,y]$.
    Hence $(R \cup (R^{-1})) \subseteq S$.
qed.

Inverse and complement of a relation

Include relations.

Let R be relation.

Lemma: $((R^{-1})^C) = ((R^C)^{-1})$.
Proof:
    Proof $(((R^{-1})^C)[x,y]) \subseteq (((R^C)^{-1})[x,y])$:
        Assume $((R^{-1})^C)[x,y]$.
        Then not $(R^{-1})[x,y]$.
        Then not $R[y,x]$.
        Then $(R^C)[y,x]$.
        Hence $((R^C)^{-1})[x,y]$.
    qed.
    Proof $(((R^C)^{-1})[x,y]) \subseteq (((R^{-1})^C)[x,y])$:
        Assume $((R^C)^{-1})[x,y]$.
        Then $(R^C)[y,x]$.
        Then not $R[y,x]$.
        Then not $(R^{-1})[y,x]$.
        Hence $((R^{-1})^C)[x,y]$.
    qed.
qed.

Cantor's theorem

```
Include functions.

Let A be set.

Definition: for all f. f: A -> (A℘) implies diagonalized(A) is set and (for all x.
    x ∈ diagonalized(A) iff (x ∈ A and (for all y ∈ (A℘). f[x,y] implies not x ∈ y))).


Lemma: for all f. f: A -> (A℘) implies f is not surjective.
Proof:
    Assume exists f. f: A -> (A℘) and f is surjective.
    Case A is empty:
        Then A℘ is nonempty.
        Then contradiction.
    qed.
    Case A is nonempty:
        Take M such that M = diagonalized(A).
        Then M ∈ (A℘).
        Take a such that a ∈ A and f[a,M].
        Case a ∈ M:
            Then not a ∈ M.
            Then contradiction.
        qed.
        Case not a ∈ M:
            Then a ∈ M.
            Then contradiction.
        qed.
    qed.
    Hence contradiction.
qed.
```

Knaster–Tarski theorem

Include functions.

Let S,T,U be set.

Notation leq: x <= y.
Proposition: x <= x.
Proposition: x <= y and y <= x implies x = y.
Proposition: x <= y and y <= z implies x <= z.

Definition DefLB: for all u. lowerBound(u,S,T) iff S ⊆ T and u ∈ T and
(for all v ∈ S. u <= v).
Definition DefUB: for all u. upperBound(u,S,T) iff S ⊆ T and u ∈ T and
(for all v ∈ S. v <= u).
Definition infimum: for all u. infimum(u,S,T) iff S ⊆ T and u ∈ T and lowerBound(u,S,T) and
(for all lowerBound(v,S,T). v <= u).
Definition DefSup: for all u. supremum(u,S,T) iff S ⊆ T and u ∈ T and
lowerBound(u,S,T) and (for all lowerBound(v,S,T). u <= v).

Proposition InfUnique: S ⊆ T and infimum(u,S,T) and infimum(v,S,T) implies
u = v.
Proposition SupUnique: S ⊆ T and supremum(u,S,T) and supremum(v,S,T) implies
u = v.

Definition DefCLat: T is completeLattice iff for all S ⊆ T.
(exists u. infimum(u,S,T)) and (exists v. supremum(v,S,T)).

Let f: U -> U.

Definition SetFix: fixPoints(T) is set and (for all x ∈ T. x ∈ fixPoints(T) iff f[x,x]).
Definition DefMonot: f is monotone iff for all x1 ∈ T. for all x2 ∈ T. x1 <= x2 implies (for all y1 ∈ U. for all y2 ∈ U. f[x1,y1] and f[x2,y2] implies y2 <= y2).

Definition upperBounds: f is monotone implies upperBounds(U) is set and (for all x. x ∈ upperBounds(U) iff x ∈ U and upperBound(x,T,U) and (for all y. f[x,y] implies y <= x)).
Definition lowerBounds: f is monotone implies lowerBounds(U) is set and (for all x. x ∈ lowerBounds(U) iff x ∈ U and lowerBound(x,T,U) and (for all y. f[x,y] implies x <= y)).

Knaster–Tarski theorem (continued)

Lemma Tarski:
    U is completeLattice and f is monotone implies fixPoints(U) is completeLattice.
Proof:
    Assume U is completeLattice and f is monotone.
    Assume M is set and M $\subseteq$ fixPoints(U).

    Proof exists u. supremum(u,M,fixPoints(U)):
        Take P such that P = upperBounds(U).
        Take p such that infimum(p,P,U).
        Take y such that f[p,y].
        Then lowerBound(y,P,U) and upperBound(y,M,U).
        Then p = y and supremum(p,M,fixPoints(U)).
    qed.

    Proof exists v. infimum(v,M,fixPoints(U)):
        Take Q such that Q = lowerBounds(U).
        Take q such that supremum(q,Q,U).
        Take y such that f[p,y].
        Then upperBound(y,Q,U) and lowerBound(y,M,U).
        Then q = y and infimum(q,M,fixPoints(U)).
    qed.

    Hence exists u. infimum(u,M,fixPoints(U)) and
        (exists v. supremum(v,M,fixPoints(U))).
    Hence fixPoints(U) is completeLattice.
qed.

Appendix B

Tutorial

The ELFE system allows for verifying mathematical proofs. In this tutorial, we will take a look at its language features.

Formulas

In order to make mathematical statements, we use a language that is similar to first-order logic. Concretely, we have the following syntax constructs at our disposal:

for all ***var*****.** ***formula***	– an universally quantified statement
exists ***var*****.** ***formula***	– an existentially quantified statement
formula **iff** ***formula***	– if and only if
formula **implies** ***formula***	– an implication
formula **and** ***formula***	– both should be true
formula **or** ***formula***	– either one is true
not ***formula***	– a negation
contradiction	– corresponds to a $\bot$ in first-order logic

We can use alphanumeric strings as variables. In order to introduce mathematical properties, we will use atoms inside a formula. We can construct atoms as follows:

var **is** ***predicate***
var **is not** ***predicate***
term **=** ***term***
predicate(term, ... , term)

For example, we may say 'R is symmetric' or 'symmetric(R)'. Terms can be variables or more complex – for example, the union of two sets A and B. This can be written down similar with 'union(A,B)'. Alternatively, we can use syntactic sugar: $A \cup B$. The libraries contain several different sugars. Note that we need to insert brackets when nesting sugars: '$((A \cup B)^C) = ((A^C) \cap (B^C))$'.

Top level commands

On the top level, we have six possible commands. The following three allow us to introduce mathematical statements:

Definition ***formula*****.**	– define a statement
Proposition ***formula*****.**	– derive a statement without proof
Lemma ***formula*****. Proof:** ***proof*** **Qed.**	– make a proved statement

Definitions and propositions can be used to introduce facts to a proof. Lemmas are the interesting part since here we can test our proofs – we will see in the next section how to write a proof.

Besides these three statements, we can use the following statements to shorten our proofs:

Include *file*. – includes a file of the library
Let *var* be *predicate*. – defines a meta-variable
Notation *predicate*: *pattern*. – introduces a notation

The inclusion command allows us to use background libraries. With the second command we can assign a property to a variable for the whole text. For example, if we define several properties about sets, we will write in the beginning 'Let A be set' and use 'A' afterwards. The last command allows us to introduce own syntactic sugars.

```
Include relations.
Notation disjunion: A Ů B.
Let A,B be relation.
Definition disjunionDef: (A Ů B)[x,y] iff A[x,y] or B[x,y] and not (A[x,y] and B[x,y]).
Proposition: A Ů B is relation.
Lemma disjunionAssociative: A Ů B = B Ů A.
Proof:
    ...
qed.
```

TEXT B.1: Exemplary use of top sections

Proof structures

There are three ways to prove a goal within ELFE.

Splitting the goal

In order to simplify a goal, we will often make several proofs that imply the original goal:

Proof *formula*: *proof* Qed. – make several sub proofs
Case *formula*: *proof* Qed. – make case distinctions

The sub proofs can be completely unconnected – important is that the background theory proves that all sub proofs together imply the original goal. Within a case distinction, the specified formula is assumed and we want to derive the original goal.

Unfolding the structure of the goal

Often, we will want to make assumptions in a proof.

Assume *formula*. – proof an implication
proof

Hence *formula*.

In order to prove an universally quantified statement, we can fix a particular element:

Fix *var*. – fix an universally quantified variable

Deriving intermediary steps

We can derive cornerstones for a proof with the following statement:

Then *formula*. – derive a cornerstone

We will often want to retrieve a specific element that we use afterwards in our proof. This can be done with this construction:

Take *var* such that *formula*. – fix an existing variable

```
...
Lemma disjunionAssociative: A ⋓ B = B ⋓ A.
Proof:
    Assume (A ⋓ B)[x,y].
    Case A[x,y]:
        Then not B[x,y].
        Then (B ⋓ A)[x,y].
    qed.
    Case B[x,y]:
        Then not A[x,y].
        Then (B ⋓ A)[x,y].
    qed.
    Hence (B ⋓ A)[x,y].
qed.
```

TEXT B.2: Exemplary proof