The ELFE System
Verifying Mathematical Proofs of Undergraduate Students

Maximilian Doré[1] and Krysia Broda[2]
[1]*Department of Computing, RWTH Aachen University, Germany*
[2]*Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, U.K.*

Keywords: Didactics of Mathematics, Mathematical Reasoning, Proof Checking, Formal Mathematics.

Abstract: ELFE is an interactive system for teaching basic proof methods in discrete mathematics. The user inputs a mathematical text written in fair English which is converted to a special data-structure of first-order formulas. Certain proof obligations implied by this intermediate representation are checked by automated theorem provers which try to either prove the obligations or find countermodels if an obligation is wrong. The result of the verification process is then returned to the user. ELFE is implemented in HASKELL and can be accessed via a reactive web interface or from the command line. Background libraries for sets, relations and functions have been developed. It has been tested by students in the beginning of their mathematical studies.

1 INTRODUCTION

The Soviet researcher Victor Glushkov formulated in 1971 that "to understand a proof means to be able to explain it to a machine that is operating with a relatively unsophisticated algorithm" (Glushkov, 1971, p. 111). Remarkably, teaching mathematics in university is still a mostly analogous endeavour. In order to understand mathematical reasoning, students practice writing proofs on paper and wait for the feedback of instructors to improve their understanding. Immediate feedback would greatly increase the learning curve – it is often difficult to see when a proof is complete or what steps are missing.

Such feedback could be provided by machines. And indeed, many attempts have been made to formalize mathematics. Most prominently, the interactive theorem provers ISABELLE and COQ are advanced systems; for instance COQ was used in proving the Four-color-theorem (Gonthier, 2008). However, mathematical beginners are overwhelmed by the capabilities of such systems since using them requires a deep understanding of workings of automated theorem provers (ATP).

The goal of this work is to provide users with a system that gives feedback on proofs entered in a fairly natural Mathematical language. Thereby the users are detached from the technicalities of automated theorem provers. The ELFE system provides a proof of concept that this is feasible and sensible. In the past years, several attempts have been made to create a proof verifier which accepts mathematical texts written in fair English, one of which SYSTEM FOR AUTOMATED DEDUCTION (SAD) (Verchinine et al., 2007) was most influential for our work. The SAD provides an intuitive input language, called FORTHEL. However, the user still has to dig into the automated verification process to understand why a proof does not work. The ELFE system in contrast processes the output of background provers and tries to give countermodels to wrong proofs.

```
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 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.
```

Figure 1: Exemplary ELFE text.

Consider the exemplary proof in Figure 1 which is in fact a valid ELFE text. After including a back-

Doré, M. and Broda, K.
The ELFE System.
DOI: 10.5220/0006681000150026
In *Proceedings of the 10th International Conference on Computer Supported Education (CSEDU 2018)*, pages 15-26
ISBN: 978-989-758-291-2

ground library and introducing specific sets A, B and C and functions f and g, a lemma is proposed that if the composition of f and g is injective, so the firstly applied f must be injective. This lemma is proven by the reasoning that if f maps two elements x and x' to the same element, the composition of f and g must map them to the same elements. Since this composition is injective, it follows that x and x' are the same elements and f is thus injective. Note that (g∘f){x} denotes the function application of g∘f which is put in brackets to specify the precedence of the symbols. We will learn in the following how the text is verified.

The remainder of the paper is structured as follows. We first give a brief overview of the implementation in Section 2 and web interface in Section 3. Next we introduce the Elfe language and proof structures and justify the correctness of the formalisation in Section 4. Finally we evaluate our work in Section 5 and compare it with popular current theorem provers in Section 6 before concluding with a short discussion in Section 7.

An instance of the system can be found online [1].

2 IMPLEMENTATION

The ELFE system can be accessed through a web interface or a command-line interface (CLI) as shown in Figure 2. The web interface provides an intuitive way of accessing the systems output, while the CLI offers more debugging functionality. We will take a closer look at the the web interface in Section 3.

After the text is entered via one of its interfaces, it will be parsed into an intermediary representation in first-order logic. This proof representation is presented in Section 4.2. The Verifier takes the intermediary proof representation and checks it for correctness by calling several ATPs in parallel. If a proof obligation is wrong, the Verifier tries to extract a countermodel from the background provers. The result of this verification process is then returned to the user via the chosen interface.

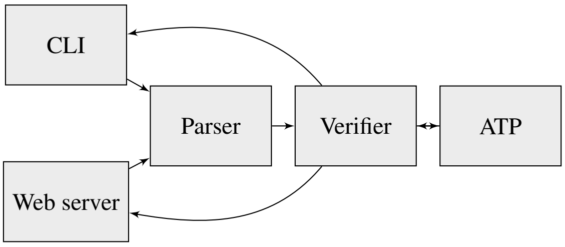


Figure 2: Architecture of the ELFE system.

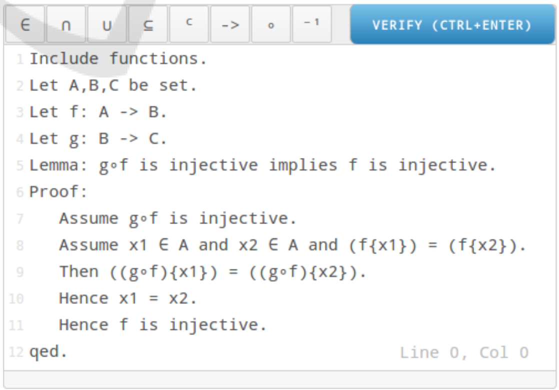
The system is implemented in HASKELL, its source code can be found online[2]. In order to parse a text, a parser combinator is constructed with the library PARSEC[3] . The framework SCOTTY[4] is used to provide a backend for the web interface. The reactive frontend is implemented with the JAVASCRIPT framework VUEJS[5] .

In order to send proof obligations to the background provers, the syntax standard TPTP (Sutcliffe, 2009) is used. Since the used ATP can be easily configured, nearly all current systems can be interfaced.

So far, we have used the provers E PROVER (Schulz, 2002), SPASS (Weidenbach et al., 2002) and VAMPIRE (Riazanov and Voronkov, 2002) due to their performance at the CADE System Competitions (Sutcliffe, 2016). Additionally, we used the provers Z3 (De Moura and Bjørner, 2008) and BEAGLE (Baumgartner et al., 2015) which do theorem proving modulo background theories. Even though we did not fully utilize, for instance, their arithmetic proving facilities, it turned out efficient to call several provers in parallel. E.g., E PROVER turned out to be fast in proving lemmas with equality while BEAGLE gave useful countermodels for wrong proof obligations.

3 WEB INTERFACE

The front-end of the web interface shown in Figure 3 consists of a simple text field in which the user can enter his proof. Above the input, several special characters can be entered by mouse click besides a button that initiates the verification process.



```
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.
```

Figure 3: The web interface of ELFE.

After the verification process has finished, colours

[1] https://elfe-prover.org

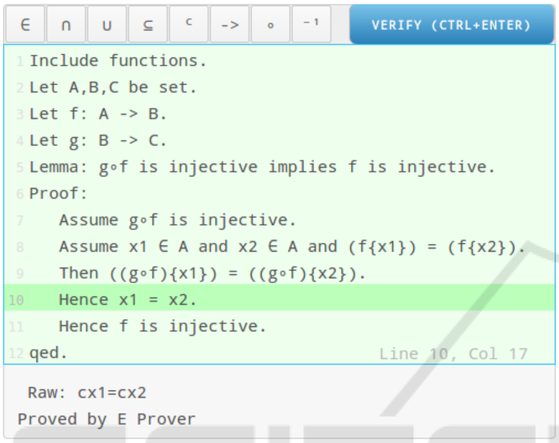
[2] https://github.com/maxdore/elfe

[3] https://hackage.haskell.org/package/parsec

[4] https://hackage.haskell.org/package/scotty

[5] https://vuejs.org/

indicate the status of each text line as depicted in Figure 4. Since all text is green, the text was considered correct. The user can inspect the verification process by clicking in specific lines, more information about the verification is then given in the box below the text field. In our example, we learn the TPTP representation of the proof obligation x1 = x2 and that it was proved by E PROVER. Note that the variables are prefixed with c in the raw version since they are considered constants at this point in the proof. The reason for this will be explained in Section 4.3.

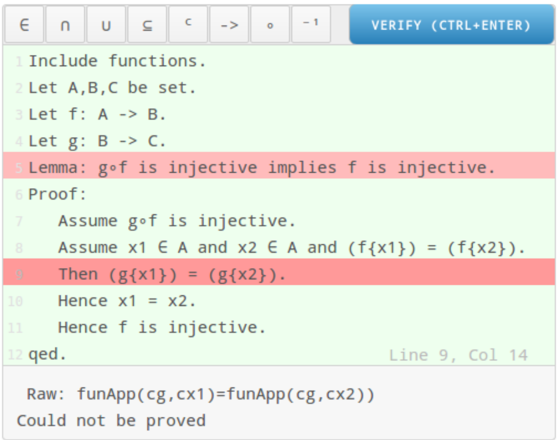


```
1  Include functions.
2  Let A,B,C be set.
3  Let f: A -> B.
4  Let g: B -> C.
5  Lemma: g∘f is injective implies f is injective.
6  Proof:
7     Assume g∘f is injective.
8     Assume x1 ∈ A and x2 ∈ A and (f{x1}) = (f{x2}).
9     Then ((g∘f){x1}) = ((g∘f){x2}).
10    Hence x1 = x2.
11    Hence f is injective.
12 qed.                                   Line 10, Col 17
```

```
 Raw: cx1=cx2
Proved by E Prover
```

Figure 4: Verified correct ELFE text.

If the user enters an incorrect proof, as in Figure 5, red colours indicate that the verification process failed. In the example in line 9 we wrongly concluded that g must have mapped x and x' to the same elements, which does not always hold. The background provers could not prove this, but also did not find a countermodel to the obligation.

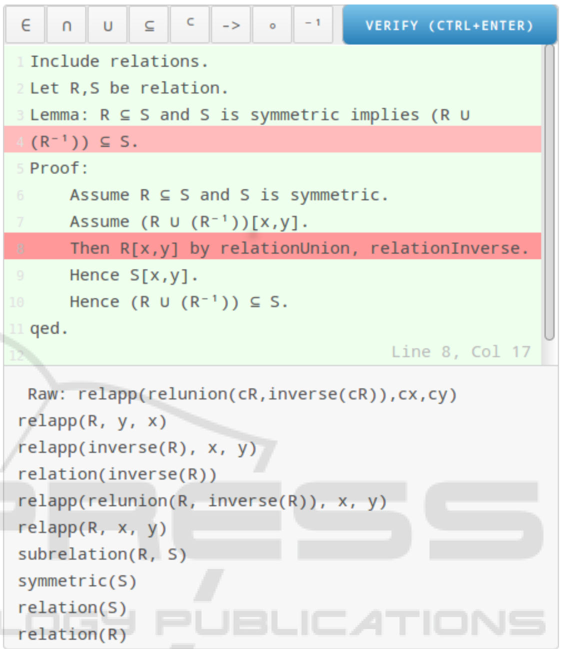


```
1  Include functions.
2  Let A,B,C be set.
3  Let f: A -> B.
4  Let g: B -> C.
5  Lemma: g∘f is injective implies f is injective.
6  Proof:
7     Assume g∘f is injective.
8     Assume x1 ∈ A and x2 ∈ A and (f{x1}) = (f{x2}).
9     Then (g{x1}) = (g{x2}).
10    Hence x1 = x2.
11    Hence f is injective.
12 qed.                                   Line 9, Col 14
```

```
 Raw: funApp(cg,cx1)=funApp(cg,cx2))
Could not be proved
```

Figure 5: An unsound ELFE text.

In the proof in Figure 6, a countermodel could be found for a wrong conclusion. The lemma states that if a relation R is included in S and S is symmetric, the inverse of R must be included in S as well. While the statement is in general correct, the proof is too imprecise and misses a case distinction. The countermodel now tells us that if x and y are in the union of R and its inverse, they might be in the inverse of R but not in R itself. Thus, the conclusion in line 8 does not in general hold. The correct version of this proof can be found in the Appendix.



```
1  Include relations.
2  Let R,S be relation.
3  Lemma: R ⊆ S and S is symmetric implies (R ∪
4  (R⁻¹)) ⊆ S.
5  Proof:
6     Assume R ⊆ S and S is symmetric.
7     Assume (R ∪ (R⁻¹))[x,y].
8     Then R[x,y] by relationUnion, relationInverse.
9     Hence S[x,y].
10    Hence (R ∪ (R⁻¹)) ⊆ S.
11 qed.
12                                        Line 8, Col 17
```

```
 Raw: relapp(relunion(cR,inverse(cR)),cx,cy)
relapp(R, y, x)
relapp(inverse(R), x, y)
relation(inverse(R))
relapp(relunion(R, inverse(R)), x, y)
relapp(R, x, y)
subrelation(R, S)
symmetric(S)
relation(S)
relation(R)
```

Figure 6: Countermodel for a wrong ELFE text.

4 ELFE LANGUAGE

The input language for ELFE is mathematical texts written in a subset of natural mathematical language. We will not introduce the whole feature set in this paper and only examine the exemplary proof of Figure 1 in the following. Other language constructs like case distinctions or sub proofs, which make a text less monolithic, are presented in (Doré, 2017).

In order to verify an ELFE text, we transform it into a special data-structure which implies certain proof obligations. Since this internal proof representation uses first-order logic, we will first introduce how to transform the ELFE language into first-order logic. This preprocessing will be presented in Section 4.1. Keywords like Then and Hence have special meanings in an ELFE proof and are used to structure a mathematical proof. This structure is captured in an intermediate proof representation which is introduced in Section 4.2. The intermediate proof representation implies certain obligations which need to be checked by the background provers. What these are will be explained in Section 4.3.

4.1 From ELFE to First-order Logic

First-order logic is used to encode mathematical statements. Most transformations are straightforward from ELFE to first-order logic, e.g., P implies f is injective is transformed to $P \rightarrow injective(f)$. In order to make an ELFE text more legible, three commands introduce meta-language features.

```
Include sets, 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.
     f[x,y] and
     (for all y' ∈ B. y = y' or not f[x,y']).

Let f: A → B.

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

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])).
```

Figure 7: Excerpt of the functions library.

The command Include can be used to include the axioms of a background theory. E.g., in our example in Figure 1 we include the functions library with Include functions. The user can easily create his own background theory since these are written in the ELFE language as well. You can find an excerpt of the functions library in Figure 7.

The command Notation is used to introduce syntactic sugars. One can write an arbitrary pattern of Unicode characters to define such a pattern, e.g., Notation function: f: A → B. The alphabetical parts of the pattern, i.e., f, A and B are treated as placeholders for arbitrary terms. Thus, all terms of the form

*: * → *

with * being arbitrary terms are subsequently considered instances of the predicate function. For example, g: B → C will be transformed internally to the first-order formula $function(g,B,C)$. Similarly, the notation for composition is defined as g∘f. Consider the version of our proof in raw first-order logic in Figure 8, where the first line of our exemplary ELFE proof Assume g∘f is injective is transformed into Assume $injective(composition(g,f))$. Note that notations can be used both for term and predicate symbols.

```
Lemma: $\forall set(A), set(B), set(C), function(f,A,B),$
$function(g,B,C).\ \ injective(composition(g,f)) \rightarrow$
$injective(f)$.
Proof:
  Assume $injective(composition(g,f))$.
  Assume $funApp(f,x) = funApp(f,x')$
            $\wedge in(x,A) \wedge in(x',A)$.
  Then $funApp(composition(g,f),x)$
            $= funApp(composition(g,f),x')$.
  Hence $x = x'$.
  Hence $injective(f)$.
qed.
```

Figure 8: The injectivity proof without syntactic sugar.

The command Let binds a predicate symbol to a variable, effectively assigning a type to a symbol. By writing Let A,B, C be set, we ensure that in all following statements A, B and C have the predicate symbol set. Consider Figure 8 which shows the injectivity proof after removing meta-level language features. A, B and C are introduced universally quantified as sets in the lemma.

4.2 Statement Sequences

So far, we have only seen how single mathematical statements are transformed into first-order formulas. In order to capture the structure of a proof, we propose a special kind of data-structure, so-called statement sequences. Intuitively, a statement holds a first-order formula with an identifier and a proof. A proof can consist of other statements in order to represent complex proof objects.

Definition 1. Statement Sequences.
A statement S is a tuple ID × GOAL × 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 $S_1, ..., S_n$

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

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

Consider the example in Figure 9. We will depict a statement visually in the following as a box with its ID in the upper-left corner. The GOAL of a statement is written in the header of a statement, the PROOF below. A PROOF can take different forms to capture complex proof structures. The axioms of a text however are simply annotated by ASSUMED. E.g., the statements S_{fun} and S_{inj} depict the statements resulting from the definitions in Figure 7. In the functions library, numerous additional definitions are made which are omitted here. Statements annotated with ASSUMED will be depicted green in the following. Below the axioms, the statement S of the lemma of our text in Figure 1 follows. In order to prove this statement, we need more advanced proof structures which will be introduced in the next Section 4.3. The statement is depicted red and with a dashed border to indicate that its proof is not complete.

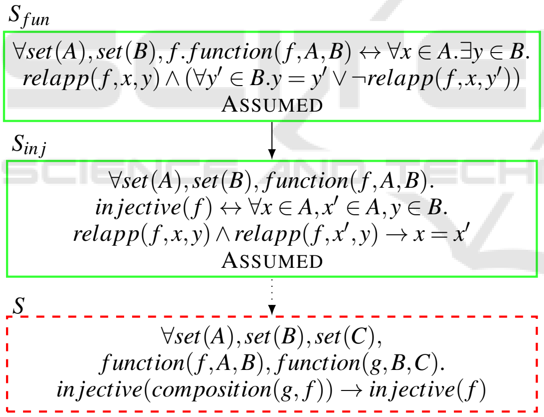

S_{fun}
$\forall set(A), set(B), f.function(f,A,B) \leftrightarrow \forall x \in A.\exists y \in B.$
$relapp(f,x,y) \wedge (\forall y' \in B.y = y' \vee \neg relapp(f,x,y'))$
ASSUMED
S_{inj}
$\forall set(A), set(B), function(f,A,B).$
$injective(f) \leftrightarrow \forall x \in A, x' \in A, y \in B.$
$relapp(f,x,y) \wedge relapp(f,x',y) \rightarrow x = x'$
ASSUMED
S
$\forall set(A), set(B), set(C),$
$function(f,A,B), function(g,B,C).$
$injective(composition(g,f)) \rightarrow injective(f)$

Figure 9: Exemplary statement sequence.

To give an overview of the other types of PROOF: A proof BYSEQUENCE and BYSPLIT makes it possible to nest more complex derivation sequences. A statement annotated with BYCONTEXT will be checked by the background provers. BYSUBCONTEXT is a special case of this proof type which allows for restricting the context of the statement.

4.3 Proved Statements

Since we want to verify that a text is sound, we need to introduce a soundness criteria for statements. Axioms of a text are considered correct, but the lemma needs a more subtle criteria.

First we will define which axioms are considered relevant to a statement. Intuitively, the context of a statement in a statement sequence are all statements "above" it.

Definition 2. Context of a Statement.
Let $S_1,\ldots S_n$ be a statement sequence. The context of a statement S_k is inductively defined as

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

For example, in Figure 9, the context of statement S consists of the respective goals of S_{fun} and S_{inj} (as well as other definitions of the library which are omitted here). With that, we can define an appropriate soundness criteria for statements.

Definition 3. Proved Statement.
Let S be a statement with S.GOAL = ϕ.
We call S proved iff $\Gamma(S) \vDash \phi$.

In other words, a statement is considered proved if it already followed from the theory created by its context. The statements S_{fun} and S_{inj} in Figure 9 are not proved since they build up the axioms of our theory. The statement S however should follow from these axioms, i.e., should be a proved statement. In order to show that S is proved, we will create a more complex proof object in the following such that correctness of the proof object implies that S followed from its context.

We start by unfolding the outer implication of the lemma $\forall set(A), set(B), set(C), function(f,A,B), function(g,B,C).\ \ injective(composition(g,f)) \rightarrow injective(f)$. More specifically, we fix specific sets A, B and C and functions f, g. As we see in Figure 10, this is captured in our data-structure by introducing another statement S_1 such that the proof of S is BYSEQUENCE S_1. We represent proofs BYSEQUENCE by putting the proof inside the statement to prove. The difference between S and S_1 is that we removed the quantifiers and replaced the variables with constants (depicted in blue and with an overline $\bar{A}$ in case the color does not show up).

In order to prove the new goal of S_1, we do a so-called unfolding of the implication. The left hand side is put in the statement S_2 and annotated with ASSUMED such that it is in the context of S_3, which holds the right hand side of the implication.

The whole reduction from S to S_3 is done automatically by the system. It detects if meta-variables are contained in the goal and injects the proof automatically.

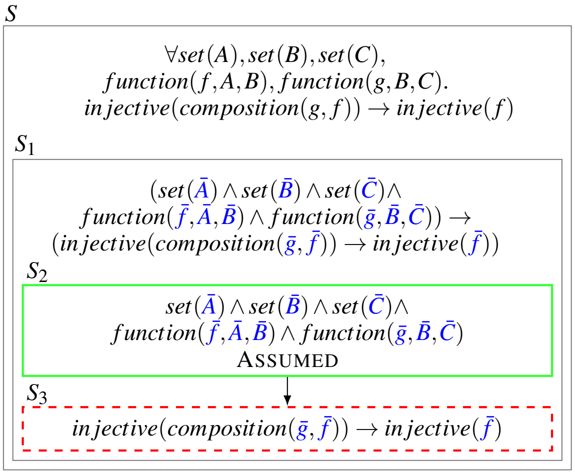

S
$\forall set(A), set(B), set(C),$
$function(f,A,B), function(g,B,C).$
$injective(composition(g,f)) \to injective(f)$
S_1
$(set(\bar{A}) \wedge set(\bar{B}) \wedge set(\bar{C}) \wedge$
$function(\bar{f},\bar{A},\bar{B}) \wedge function(\bar{g},\bar{B},\bar{C})) \to$
$(injective(composition(\bar{g},\bar{f})) \to injective(\bar{f}))$
S_2
$set(\bar{A}) \wedge set(\bar{B}) \wedge set(\bar{C}) \wedge$
$function(\bar{f},\bar{A},\bar{B}) \wedge function(\bar{g},\bar{B},\bar{C})$
ASSUMED
S_3
$injective(composition(\bar{g},\bar{f})) \to injective(\bar{f})$

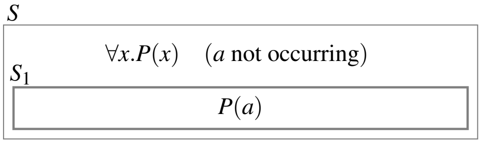
Figure 10: Unfolding meta-variables.

With this, we have reduced the problem of showing that S is proved to showing that S_3 is proved. In order to convince us that S_3 is proved indeed implies that S is proved, we will first see that it is sound to fix an universally quantified variable to a constant. This can be done by natural deduction which has been shown to be sound (Fitting, 1990). Concretely, our construction is analogous to the following deduction rule:

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

We use this deduction rule in showing the soundness of our construction in Lemma 1.

Lemma 1. $\forall$ **Introduction.**
Let S be a statement such that S.GOAL $= \forall x.P(x)$ and a not occurring in S.GOAL, S.PROOF $=$ BYSEQUENCE S_1, S_1.GOAL $= P(a)$ and S_1 is proved:



Then S is proved.

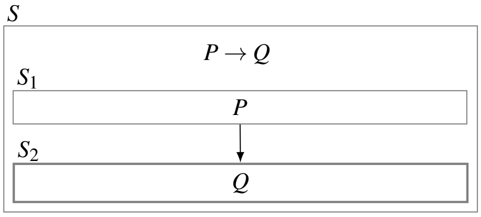
Proof. Since S_1 is proved and $\Gamma(S) = \Gamma(S_1)$, we have $\Gamma(S) \vDash P(a)$. With $(\forall I)$ it follows that $\Gamma(S) \vDash \forall x.P(x)$ since a does not occur in $P(x)$. □

Next we have to show that it is sound to assume the left hand side of an implication and deduce the right hand side. Again, this is analogous to a natural deduction rule:

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

This rule is used in the soundness proof in Lemma 2.

Lemma 2. $\to$ **Introduction.**
Let S be a statement such that S.GOAL $= P \to Q$, S.PROOF $=$ BYSEQUENCE S_1, S_2, S_1.GOAL $= P$, S_2.GOAL $= Q$ and S_2 is proved:



Then S is proved.

Proof. We have $\Gamma(S_2) = \Gamma(S) \cup \{P\}$. Since S_2 is proved, $\Gamma(S) \cup P \vDash Q$. With $(\to I)$ it follows $\Gamma(S) \vDash P \to Q$. □

Now we will construct the proof of S_3 as shown in Figure 11. In the proof text in Figure 1, we explicitly wrote Assume $injective(composition(g,f))$. [...] Hence $injective(f)$. Analogous to the unfolding of the implication of S_1 in Figure 10, we assume the left hand side and now have to prove the right hand side. Again, this is sound as proved in Lemma 2.

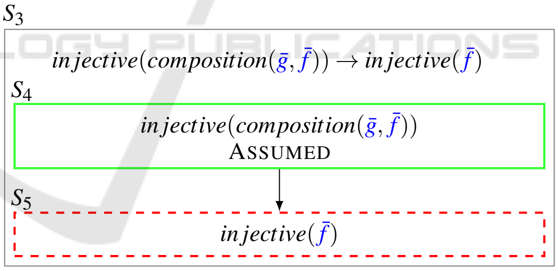

S_3
$injective(composition(\bar{g},\bar{f})) \to injective(\bar{f})$
S_4
$injective(composition(\bar{g},\bar{f}))$
ASSUMED
S_5
$injective(\bar{f})$

Figure 11: Unfolding an implication.

Now, we have to prove that $injective(f)$ holds. In order to do that, the proof in Figure 1 uses the definition of injectivity: Assume $funApp(f,x) = funApp(f,x') \wedge in(x,A) \wedge in(x',A)$. [...] Hence $x = x'$. In other words, we prove an alternative goal. In order to retain a sound construction, we have to show two things: First, that the alternative goal indeed implies the original goal and second, that the alternative goal holds. This is represented in Figure 12 by putting two statements S_6 and S_7 below the goal of S_5. This depicts that the PROOF of S_5 is BYSPLIT S_6, S_7. Note that a proof BYSPLIT leads to a division of contexts, i.e., the derived goal of S_6 will not be put into the context of S_7. Thus, the proof BYSPLIT allows for a finer scoping of statements.

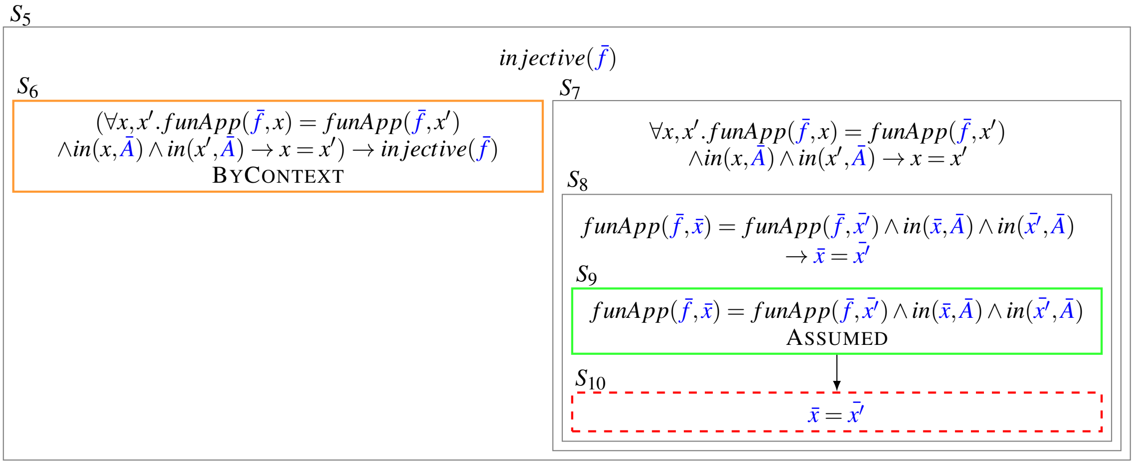

S_5
$injective(\bar{f})$
S_6
$(\forall x,x'.funApp(\bar{f},x) = funApp(\bar{f},x') \wedge in(x,\bar{A}) \wedge in(x',\bar{A}) \rightarrow x = x') \rightarrow injective(\bar{f})$
BYCONTEXT
S_7
$\forall x,x'.funApp(\bar{f},x) = funApp(\bar{f},x') \wedge in(x,\bar{A}) \wedge in(x',\bar{A}) \rightarrow x = x'$
S_8
$funApp(\bar{f},\bar{x}) = funApp(\bar{f},\bar{x}') \wedge in(\bar{x},\bar{A}) \wedge in(\bar{x}',\bar{A}) \rightarrow \bar{x} = \bar{x}'$
S_9
$funApp(\bar{f},\bar{x}) = funApp(\bar{f},\bar{x}') \wedge in(\bar{x},\bar{A}) \wedge in(\bar{x}',\bar{A})$
ASSUMED
S_{10}
$\bar{x} = \bar{x}'$

Figure 12: Proving an alternative goal.

The statement S_6 contains the soundness check. Its proof is BYCONTEXT which means that it will be sent to the background provers. If some ATP finds a proof, a statement annotated with BYCONTEXT is considered proved. This is here the case if our definition of injectivity indeed allows us to prove this alternative goal. We will depict statements proved BYCONTEXT in orange in the following.

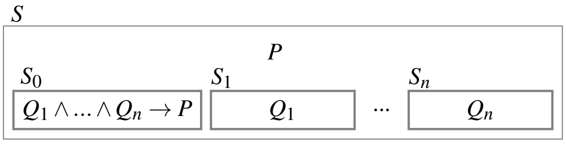
Statement S_7 contains the proof of the alternative goal. Again, the universally quantified variables x and x' are fixed to constants. Afterwards, the implication is unfolded. As seen in Lemma 1 and Lemma 2, this is sound.

To convince us that this construction is sound, we have to use two additional natural deduction rules:

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

These rules will be used in the proof of Lemma 3 which is an abstract case of our approach in Figure 12.

Lemma 3. Splitting a Goal.
Let S be a statement such that S.GOAL = P, S.PROOF = BYSPLIT $S_0, S_1, ..., S_n$, S_0.GOAL = $Q_1 \wedge ... \wedge Q_n \rightarrow P$, S_i.GOAL = Q_i and S_i is proved for $i = 1, ..., n$:



Then S is proved.

Proof. We have $\Gamma(S) = \Gamma(S_i)$ for $i = 0, \dots, n$. With S_i proved for $i = 1, \dots, n$ we have $\Gamma(S) \vDash Q_i$ for $i = 1, \dots, n$. With $(\wedge I)$ it follows $\Gamma(S) \vDash Q_1 \wedge \cdots \wedge Q_n$. With S_0 proved we also have $\Gamma(S) \vDash Q_1 \wedge ... \wedge Q_n \rightarrow P$. Thus, we can deduce with $(\rightarrow E)$ that $\Gamma(S) \vDash P$. □

The remaining bit to prove is the goal of S_{10}, i.e., that $x = x'$ follows from the context. However, in the text in Figure 1 the next derivation step is Then $funApp(composition(g, f), x) = funApp(composition(g, f), x')$.. This statement does not change the overall goal we want to proof, but gives a cornerstone to how one can derive the goal. As depicted in Figure 13, this additional finding will first be verified by annotating statement S_{11} with BYCONTEXT. Afterwards, the actual goal is proved. Since the user gave no additional proving methods, we send the final goal $x = x'$ to the background provers as well.

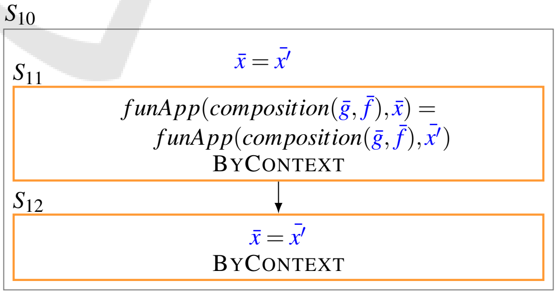
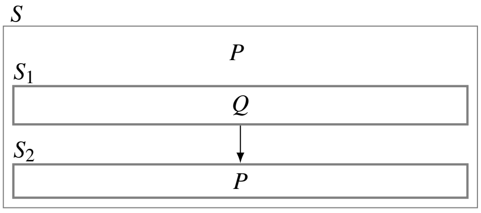

S_{10}
$\bar{x} = \bar{x}'$
S_{11}
$funApp(composition(\bar{g},\bar{f}),\bar{x}) = funApp(composition(\bar{g},\bar{f}),\bar{x}')$
BYCONTEXT
S_{12}
$\bar{x} = \bar{x}'$
BYCONTEXT

Figure 13: Giving a cornerstone to a proof.

If S_{11} can be derived by the background provers already, the theory created by the context of S_{12} is not extended by adding S_{11}. This is formally reflected in Lemma 4.

Lemma 4. Deriving a Cornerstone.
Let S be a statement such that S.GOAL = P, S.PROOF = BYSEQUENCE S_1, S_2, S_1.GOAL = Q, S_2.GOAL = P and S_1 is proved:



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 $\Gamma(S) \vDash Q$. Hence, with S_2 proved we have $\Gamma(S_2) \vDash P$ and it follows that already $\Gamma(S) \vDash P$. □

This completes our construction of the internal proof representation of the lemma in Figure 1. Three statements S_6, S_{11} and S_{12} are annotated BYCONTEXT and will be sent to the background provers. If each of these three statements can be derived from their respective contexts, we can conclude that the original goal of S already followed from its context. The proof of the lemma is then considered sound.

5 EVALUATION

The tool was tested by students in the beginning of their mathematical studies. In Section 5.1, we will take a look at their evaluation and suggestions. We also formalized some more advanced theorems in the system, e.g., Cantor's theorem and the Knaster-Tarski theorem, and will discuss our experiences as well as the system's inherent limitations in Section 5.2.

5.1 User Feedback

The system was tested with 12 undergraduates of Computing, Mathematics and Electrical Engineering at Imperial College London, of which none had prior experience with interactive theorem provers. Due to the limited time frame we were not able to evaluate the system further.

At first, the students were given the proof sketch shown in Figure 14. An intuition about the proof was given in natural language, i.e., it was explained we want to prove that the complement of the complement of a set is the set itself. All students were able to identify the proof pattern, i.e., that we show set inclusion in both directions. This is a very common proof procedure to show equality of two sets. When writing the remaining bit of the proof, the students successfully resolved syntactic errors by inspecting the parsing errors and all completed the proof. The syntactic characteristics of ELFE, e.g., that Then and Hence have distinct meanings, did not pose an obstacle since they only had to copy the structure of the first sub proof. However, only two students were able to figure out the meanings of these language features, i.e., that Hence closes an implication whereas Then is for giving cornerstones to a proof. This suggests that using ELFE requires an introduction to the different language features and users cannot start writing proofs right away.

Later on, the testers were given more complex proof sketches. Students who were in general comfortable with mathematical reasoning were able to complete the proofs. The other students had problems grasping the idea of the proof and did not start to write a proof 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.
```

Figure 14: Proof to be completed in the evaluation.

After the students 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

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 its 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 humans and not only for a computer.

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

5.2 Limits of the Current 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. Some additional lemmas and function symbols which were introduced to make a proof more readable for humans increase the difficulty for the background provers. If the background provers take too long in proof search, it is 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, the raw proof obligations given to the background provers were more helpful in finding bugs by manually deleting and changing the given premises. This is due to constructions like Let which shorten a proof, but also hide what is going on inside the system.

The Notation command has 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.

BEAGLE was able to provide countermodels to a wrong proof only if the number of premises was limited. Restricting the context of a derivation step increased the success rate significantly. However, for new users it is certainly difficult to relate a countermodel to the entered text since it is given in the raw TPTP format.

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 cleverness is particularly 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.

Writing larger proof texts in straightforward domains as set theory can be easily done in ELFE. However, some properties like well-foundedness are not expressible at all in first-order logic, so it might be expedient for future versions to use higher-order logic at the core of statement sequences.

6 RELATED WORK

In Section 6.1, we will take a look at mathematical text verifiers like the SYSTEM FOR AUTOMATED DEDUCTION, which heavily influenced this project. In Section 6.2, we will compare ELFE to the popular interactive theorem provers ISABELLE and COQ.

6.1 Mathematical Text Verifier

In the following, we will present two projects aimed for verifying mathematical texts: The SYSTEM FOR AUTOMATED DEDUCTION (SAD) in Section 6.1.1 and NAPROCHE in Section 6.1.2.

6.1.1 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. His goal was to develop a tool that shortens long but "obvious" proofs to users. These omitted parts should be verified by automated theorem provers. (Verchinine and Paskevich, 2000)

SAD uses the input language FORTHEL which allows for expressing mathematical statements intuitively. 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. (Verchinine et al., 2007)

Currently, it is not possible to work with functions in SAD due to the lack of background libraries. Thus, we could not implement the injectivity proof of Figure 1 in SAD.

6.1.2 Naproche

The Naproche system was a joint 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. (Cramer et al., 2009)

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. The semantics of PRS have been researched extensively; however, the project is not continued and has no working version available.

6.2 Interactive Theorem Prover

The classical approach to interactive theorem proving integrates a human user strongly in the technical verification process. We will briefly introduce the popular provers Isabelle in Section 6.2.1 and Coq in Section 6.2.2 with their respective formalization of the injectivity proof in Figure 1.

6.2.1 Isabelle

Isabelle is a joint project of Cambridge University and the Technical University Munich. It supports polymorphic higher-order logic, augmented with axiomatic type classes. At present it provides useful proof procedures for Constructive Type Theory, various first-order logics, Zermelo-Fraenkel set theory and higher-order logic. (Nipkow et al., 2002)

Consider the injectivity proof written in Isabelle in Figure 15. The predicate inj_on f A expresses that function f is injective on the domain A. The proof structure is close to the one used in Elfe: We introduce arbitrary x and x' which f maps to the same element and conclude that they must have been the same. One has to specify the automated proof tactics and used premises: In our example, the derivations are made by term rewriting using definitions comp_def and inj_on_def from the background library.

In comparison to Elfe, the user is therefore more involved in the automated verification process. Since 2007, Isabelle offers the extension Sledgehammer. By calling several ATP, Sledgehammer tries to determine which premises are important to a goal. It then tries to reconstruct the automated proofs with methods implemented in Isabelle. In fact, the mechanical prove methods needed in Figure 15 can be found by invoking Sledgehammer.

```
theory InjectiveComposition
  imports Fun
begin

lemma:
  assumes "inj_on (g ∘ f) A"
  shows "inj_on f A"
proof
  fix x x'
  assume "x ∈ A" and "x' ∈ A"
  moreover assume "f x = f x'"
  then have "(g ∘ f) x = (g ∘ f) x'"
      by (auto simp: comp_def)
  ultimately show "x = x'" using assms
      by (auto simp: inj_on_def)
qed
```

Figure 15: Proof in Isabelle.

In a recent study, 34% of nontrivial goals contained in representative Isabelle texts could be proved by Sledgehammer. With this extension, Isabelle allows beginners to prove challenging theorems. The creators note that Sledgehammer was not designed as a tool to teach Isabelle since it focused primarily on experienced users. However, it changed the way Isabelle is taught. Beginners do not have to learn about low level proving tactics and how they work but can focus on the proof from a higher level. (Paulson and Blanchette, 2010)

```
Require Import Basics.
Definition injective {A B} (f : A → B) :=
  forall x y : A, f x = f y → x = y.
Theorem c_inj (A B C:Type) (f:A→B) (g:B→C):
  (injective (compose g f)) → injective f.
Proof.
  intuition.
  intros x x'.
  pose (f x = f x').
  intuition.
  assert (g (f x) = g (f x')).
  { elim H0. rewrite H0. trivial. }
  auto.
Qed.
```

Figure 16: Proof in Coq.

6.2.2 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. In order to prove a proposition, one has to construct a term with the type corresponding to the proposition.

Consider the injectivity proof implemented in Figure 17. Again, the idea of the proof is to show that f x = f x' implies x = x'. However, we have to explicitly apply rewrite techniques to make the derivation steps. The tactic intuition says that we can assume a left hand side of an implication and then prove the right hand side. Afterwards, we want to make sure that we can just apply g on both sides. We have to rewrite both sides of H0, which stands for f x = f x', in order to get to our assertion. The final goal x = x' is then derived by applying the rewrite technique auto.

As we see, the translation process of mathematical texts to functional programs requires a good understanding of type theory and is not suitable for mathematical beginners.

Consequently, the most prominent current interactive theorem provers are of a deeply technical natural. They are thought of as programming languages that happen to prove theorems, and not digitisations of mathematical language.

7 DISCUSSION

This paper presented ELFE, a system that checks proofs in discrete mathematics. Entered texts are transformed to statement sequences, a special datastructure of first-order formulas. Remaining proof obligations are then checked by background provers. Statement sequences are a powerful intermediate proof representation which can hold manifold proof techniques. The clear soundness criteria allows for extending the proof techniques easily.

Students who tested the system liked especially that they got immediate feedback on their proof work. The implemented background libraries allow for an easy start. Once a user becomes familiar with the tool, he can easily construct his own background libraries. Certainly, more evaluation of the system in pedagogical environments is necessary. It will be particularly interesting to examine how teachers can incorporate the system in their courses.

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.

In addition to giving countermodels for wrong proofs, one could utilize more features of the background provers. Many provers return in-depth information about the proof of a conjecture. This information could be useful for users in order to understand why a proof works or fails. The challenge is to present the technical output of the background provers via an intuitive interface. In order to do proofs with arithmetic, it might be useful to utilize already implemented arithmetic capabilities of background provers such as Z3 and BEAGLE. Expert users presumably prefer systems with deep insight into the technical verification process, but an abstraction is necessary if we want to use computers in teaching mathematics.

The biggest structural limitation of ELFE is that it internally uses first-order logic. E.g., with the current capabilities it is not straightforward to implement proofs by induction. The recent years have seen interesting advances in automated theorem proving of typed higher-order logic. A new standard for typed higher-order-logic has been added to TPTP which is used by several provers like LEO-II (Benzmüller et al., 2015) and SATALLAX (Brown, 2012). A next version of ELFE could use this development in order to provide a more powerful way of expressing mathematics. This requires to introduce a meaningful type system for ELFE.

REFERENCES

Parsec: Monadic parser combinators. https://hackage.haskell.org/package/parsec. Accessed: 2017-01-03.

Scotty: Haskell web framework. https://hackage.haskell.org/package/scotty. Accessed: 2017-01-03.

Vuejs: The progressive javascript framework. https://vuejs.org/. Accessed: 2017-01-03.

Baumgartner, P., Bax, J., and Waldmann, U. (2015). BEAGLE–a hierarchic superposition theorem prover. In *Proc. CADE-25*, pages 367–377.

Benzmüller, C., Sultana, N., Paulson, L. C., and Theiß, F. (2015). The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404.

Brown, C. E. (2012). SATALLAX: An automatic higher-order prover. In *Proc. IJCAR 2012*, pages 111–117.

Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., and Veldman, J. (2009). The NAPROCHE project–controlled natural language proof checking of mathematical texts. In *Proc. CNL 2009*, volume 5972, pages 170–186.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340.

Doré, M. (2017). ELFE – An interactive theorem prover for undergraduate students. Bachelor thesis.

Fitting, M. (1990). *First-order Logic and Automated Theorem Proving*. Springer, 2nd edition.

Glushkov, V. M. (1971). Problems in the theory of automata and artificial intelligence. *Journal of Cybernetics*, 1(1):97–113.

Gonthier, G. (2008). Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393.

Nipkow, T., Wenzel, M., and Paulson, L. C. (2002). *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer.

Paulson, L. C. and Blanchette, J. C. (2010). Three years of experience with SLEDGEHAMMER, a practical link between automatic and interactive theorem provers. In *Proc. IJCAR 2010*, pages 1–10.

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

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

Sutcliffe, G. (2009). The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362.

Sutcliffe, G. (2016). The CADE ATP System Competition. *AI Magazine*, 37(2):99–101.

Verchinine, K., Lyaletski, A., and Paskevich, A. (2007). SYSTEM FOR AUTOMATED DEDUCTION (SAD): a tool for proof verification. In *Proc. CADE-21*, pages 398–403.

Verchinine, K. and Paskevich, A. (2000). FORTHEL–the language of formal theories. *International Journal of Information Theories and Applications*, 7(3):120–126.

Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobald, C., and Topić, D. (2002). SPASS version 2.0. In *Proc. CADE-18*, pages 45–79.

APPENDIX

```
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.
```

Figure 17: Correct ELFE proof about relations.