SEMI-AUTOMATED CONSTRUCTION OF PROOF SCHEMATA

Gela Chankvetadze[1], Lia Kurtanidze[2], Mikheil Rukhaia[1]

[1]I. Vekua Institute of Applied Mathematics,
I. Javakhishvili Tbilisi State University,
University Str. 2, 0186 Tbilisi, Georgia.
[2]Faculty of Informatics, Mathematics and Natural Sciences,
Georgian University,
I. Chavchavadze Ave. 53A, 0179 Tbilisi, Georgia.

(Received: 12.08.16; accepted: 11,12.16)

Abstract

In this paper we present a goal-directed proof-search algorithm for formula schemata, which is based on a sequent calculus. Usually, sequent calculus inference rules can be applied freely, producing a redundant search space. The standard approaches are extended to formula schemata to get rid of redundancy in a proof-search. A formula schema is a finite representation of an infinite sequence of first-order formulas, thus complete automation of the process is not feasible. Still, there are some (not so trivial) subclasses, where the process can be fully automated.

Key words and phrases: formula schemata, sequent calculus, proof-search.
AMS subject classification: 68T15, 03F03.

1 Introduction

Proof theory takes its roots from G. Gentzen, when he introduced a sequent calculus, **L***ogische* **K***alkül*, for first-order logic [18]. Since then, proofs are heavily used in computer science, in particular, program and hardware verification. This gave rise to theorem proving, a new branch of mathematical logic. There are various theorem proving techniques, like resolution, tableaux, etc. It is well known that first-order logic is undecidable, therefore all complete proof-search procedures are non-terminating.

The concept of *term schematization* was introduced in [6] to avoid non-termination in symbolic computation procedures and to give finite descriptions of infinite derivations. Later, *formula schemata* for propositional logic was developed [1] to deal with schematic problems (graph coloring, digital circuits, etc.) in a more uniform way. The language is strong enough that the satisfiability problem is already not decidable for propositional schemata [3]. Although, there exist some decidable classes of propositional formula schemata and a tableaux prover (RegSTAB) is implemented [2] for a subclass, called *regular schemata*.

In [9, 17] the language of formula schemata was extended to language of *first-order schemata*, which allows us to specify an (infinite) set of first-order formulas

by a finite term. A sequent calculus, called LKS, was defined for such a language, which can be considered as an alternative to a sequent calculus with the induction rule.

The aim of this paper is to define an efficient proof-search procedure, that will, for a given first-order formula schema, obtain its proof schema. It is well known that naive proof-search in sequent calculus leads to a redundant search space. The reason is that inference rules can be applied nondeterministically. To avoid such nondeterminism, J.Y.Girard introduced a concept of *polarity* and based on it, defined a *focused sequent calculus* LC [11]. In [14], LC was adapted to proof-search and a sequent calculus LKF was obtained. Later, in [5], the technique of focusing was used in inductive theorem proving as well.

The idea of focusing lies on the classification of logical connectives into *asynchronous* and *synchronous* polarities. These polarities do not affect the provability of formulas, but the shape of proofs and proof-construction steps. Asynchronous rules are usually invertible and can be applied in any order. In contrast, if a synchronous formula is chosen for decomposition, a synchronous phase begins and all synchronous subformulas should be decomposed until the axioms or only asynchronous subformulas are reached. Unlike Gentzen's sequent calculus, the proof-construction steps become deterministic using this technique.

In this paper we do not define a focused sequent calculus, but consider an invertible version of LKS. Nevertheless, our proof-search procedure is based on ideas of focusing to narrow the search space and uses other standard and well-known approaches to avoid backtracking [4].

The rest of the paper is organized in the following way: Section 2 is devoted to basic definitions such as a language of first-order formula schemata and its sequent calculus. Section 3 and Section 4 describe main contribution of the paper – the proof-search algorithm and its implementation, respectively. Finally, we conclude the paper in Section 5.

2 Proof Schemata

We define a *schematic first-order language*, following [9], that is an extension of the language described in [1, 3] to first-order logic. It allows us to specify an (infinite) set of first-order formulas by a finite term.

We consider two sorts ω, to represent the natural numbers, and ι, to represent an arbitrary first-order domain. Our language consists of countable sets of *variables* of both sorts, and sorted n-ary function and predicate symbols partitioned into *constant function/predicate symbols* and *defined function/predicate symbols*. While the first is used to define usual first-order terms and predicates, the latter allows recursion on terms and formulas.

Terms are built from variables and constant function symbols as usual. We assume the predefined constant functions zero $0\colon \omega$ and successor $s\colon \omega \to \omega$ to be present. By $V(t)$ we denote a variable set of a term t, and by $\bar{\cdot}$ we denote the sequence of terms of appropriate sort.

For every defined function symbol f, we assume that its sort is $\omega \times \tau_1 \times \cdots \times \tau_n \to \tau$ (with $n \geq 0$ and $\tau ::= \omega \mid \iota \mid \tau \to \tau$). Then we define a *term schema*

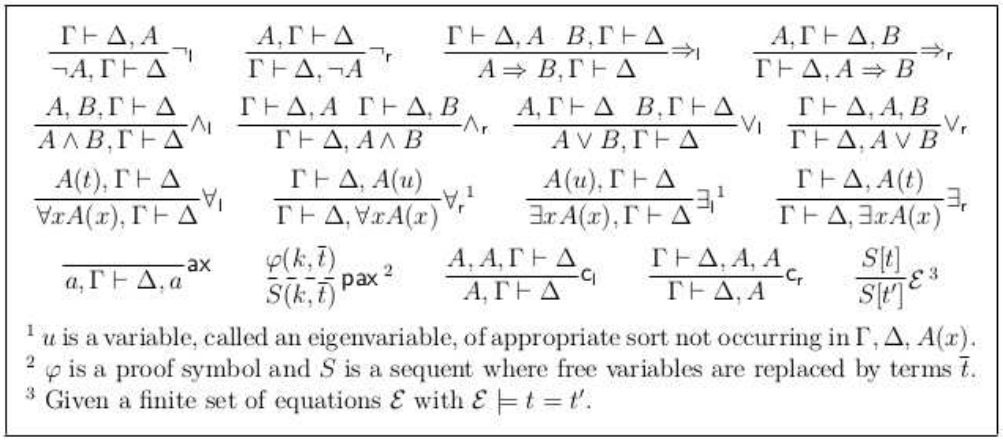

Figure 2.1: The sequent calculus LKS.

$f(y, \overline{x})$ by the two rewrite rules

$$f(0, \overline{x}) \to t_0 \qquad f(s(y), \overline{x}) \to t[f(y, \overline{x})]$$

where $V(t_0) \subseteq \{x_1, \ldots, x_n\}$ and $V(t[f(y, \overline{x})]) \subseteq \{y, x_1, \ldots, x_n\}$, and t_0, t are terms not containing f. If a defined function symbol g occurs in t_0 or t then $g \prec f$. We assume that these rewrite rules are primitive recursive, i.e. that $\prec$ is irreflexive.

We write $t \twoheadrightarrow t'$ to denote that an expression t rewrites to an expression t' in arbitrarily many steps.

Example 2.1 *Assume $g\colon \omega \times \iota \to \iota$ is a defined function symbol, $f\colon \iota \to \iota$ is a constant function symbol, and $k\colon \omega, x\colon \iota$ are variables. The following rewrite rules for g*

$$g(0, x) \to x \qquad g(s(k), x) \to f(g(k, x))$$

define a term schema $g(k, x)$. For every natural number n, $g(n, x)$ rewrites to $f^n(x)$, e.g.

$$g(s(s(s(0))), x) \twoheadrightarrow f(f(f(x))).$$

Formulas are built inductively from atomic formulas using the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$ and $\exists$ as usual. The notions of interpretation, satisfiability and validity of formulas are defined in the usual classical sense.

Analogously to defined function symbols, we assume that rewrite rules are given for defined predicate symbols as well and that they have an irreflexive order $\prec$ for the latter, to build *formula schemata*.

A variable occurrence in a formula is called *bound* if it is in the scope of $\forall$ or $\exists$ connectives, otherwise it is called *free*. In our setting, it is important to clarify how to interpret multiple occurrences of the same bound variable. An occurrence of a bound variable x is associated to the deepest quantifier that binds x.

Example 2.2 *Assume $P\colon \omega$ is a defined predicate symbol, $Q\colon \omega \times \iota$ is a constant predicate symbol, and $k\colon \omega, x\colon \iota$ are variables. The following rewrite rules for P*

$$P(0) \to \forall x Q(0,x) \qquad P(s(k)) \to \exists x(Q(k,x) \wedge P(k)). \tag{2.1}$$

define a formula schema $P(k)$. Then, e.g.

$$P(s(s(0))) \twoheadrightarrow \exists x(Q(s(s(0)),x) \wedge \exists x(Q(s(0),x) \wedge \forall x Q(0,x)))$$

which is equivalent to (by renaming of bound variables)

$$\exists x_2(Q(s(s(0)),x_2) \wedge \exists x_1(Q(s(0),x_1) \wedge \forall x_0 Q(0,x_0))).$$

Proposition 2.3 *Let A be a formula. Then every rewrite sequence starting at A terminates, and A has a unique normal form.*

Proof. Trivial, since all definitions are primitive recursive. □

Sequents are expressions of the form $\Gamma \vdash \Delta$, where Γ and Δ are multisets of formula schemata. Sequents are denoted by $S(\overline{x})$, were $\overline{x}$ indicates free variables occurring in S.

The sequent calculus LKS is given in Figure 2.1, where *proof axioms* (pax), which are called *proof links* in [9, 17], may appear only at the leaves of a proof. A proof axiom has a similar meaning as induction hypothesis: it is assumed that a proof φ at step k proves a sequent $S(k)$.

The latter leads to a notion of *proof schema*: a tuple Ψ of LKS proof pairs for $\varphi_1, \ldots, \varphi_n$ proof symbols, where each pair corresponds to the base and recursive cases of inductive definition. The proof symbols in a proof schema must be properly ordered in a sense that if $i > j$, φ_i must not contain a proof axiom referring to φ_j. We also say that the end-sequent of φ_1 is the *end-sequent* of Ψ. For a formal definition of proof schemata we refer an interested reader to [9, 17].

Example 2.4 *Let $g(k,x)$ be a term schema defined in Example 2.1. Then a proof schema of the sequent $P(a), \forall x(\neg P(x) \vee P(f(x))) \vdash P(g(k,x))$ is given in Figure 2.3 (the base case) and Figure 2.4 (the recursive case)* [1].

According to the above definitions, it is easy to see that proof schemata naturally represent infinite sequences of first-order proofs.

Proposition 2.5 *The sequent calculus LKS is sound.*

Proof. Although the calculus is not exactly the same, the proof is similar to the ones given in [9, 17]. □

[1] The proof schema is obtained by the schematic prover, thus contains redundant structural rules. Details are explained in Section 4.

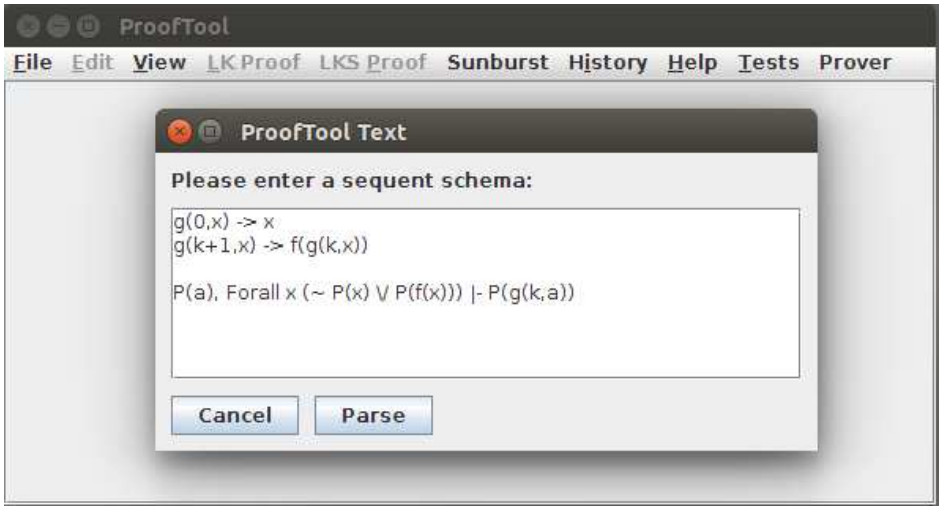

Figure 2.2: A sample input for the algorithm.

3 The Algorithm

Construction of a proof schema of a sequent is divided into two tasks: first the LKS-proof for base case and then the LKS-proof for recursive case must be constructed. The main difference between these two lies on the usage of proof axioms, i.e. induction hypothesis, which are obsolete in the base case.

To handle quantifiers, we use similar method described in [4]. This means that the choice for the weak quantifier instance term is postponed until it is obtained via unification. It works in the following way: on a decomposition step of a *weakly quantified formula* [2], we keep its original version as well (to avoid backtracking) and replace the quantified variable with an eigenvariable, until the proper term is obtained via unification. The substitution is applied to the whole proof skeleton, to replace every occurrence of the eigenvariable with the proper term.

Note that all propositional rules in LKS are invertible, thus they can be applied freely. The priority is given to unary inference rules, since the binary rules duplicate the context. Therefore, unary rules are applied when applicable and the application of binary rules is postponed as far as possible.

In the proof-search of recursive case, rewriting of defined function and predicate symbols must be done in a way that each symbol is rewritten only one step down (e.g. going from $k+1$ to k). After such rewriting, if no other rules are applicable on a sequent, the proof axiom should be introduced or algorithm must terminate with "no proof found". If the sequent contains a match instance of the end-sequent modulo k parameter, then a proof axiom to itself must be made; otherwise a proof axiom referring to a new proof symbol must be created and new proof-search for this sequent must be scheduled.

It is easy to see that this algorithm is not complete. In fact, there is no

[2] A formula, having a weak quantifier as an outermost connective.

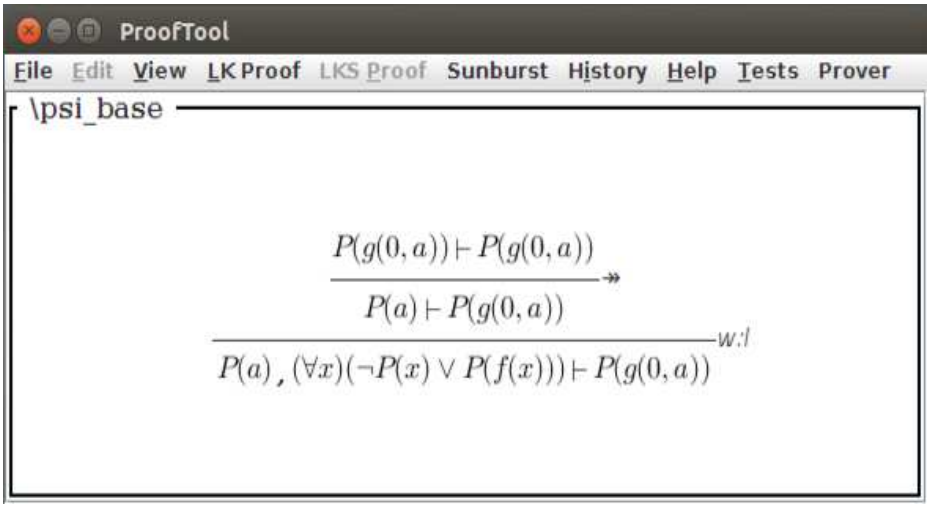

Figure 2.3: A proof for the base case is found.

complete algorithm exists, because the unsatisfiability of formula schemata is not semi-decidable even for propositional schemata [3].

4 Implementation

The algorithm is implemented under the GAPT [3] framework, which is written in the programming language Scala [16]. GAPT [10] provides data-structures, algorithms and user interfaces for analyzing and transforming formal proofs. The framework is very general and implements the basic data structures for simply-typed lambda calculus, for sequent and resolution proofs as well as expansion proofs [13]. Various theorem provers have already been integrated into this framework [7, 12]. In parallel, we have developed a Graphical User Interface called ProofTool which can be used both as a pure visualization tool (with the features like zooming, scrolling, searching, etc.) and as a proof manipulator (allowing to call GAPT's proof transformations such as cut-elimination, regularization, skolemization, etc. [4]). Details about ProofTool and how it displays formulas, sequents and proofs can be found in [8, 15].

GAPT provides several input/output formats, including one for first-order formula schemata. The format is intuitive and easy to use. An example of a sequent in this format is given in Figure 2.2. We have the following assumptions: constant function symbols are denoted by `f, f1,...`, and constant predicate symbols by `P, P1,...`. Defined function and predicate symbols are represented by `g, g1,...`, and `Q, Q1,...`, respectively. The logical connectives are represented using $\sim$, `\/`, `/\`, `=>`, `Forall`, `Exists`, and `|-` represents the sequent sign. For more detailed

[3] General Architecture for Proof Theory, `http://www.logic.at/gapt`

[4] In official release of version 2.0 this is not the case any more.

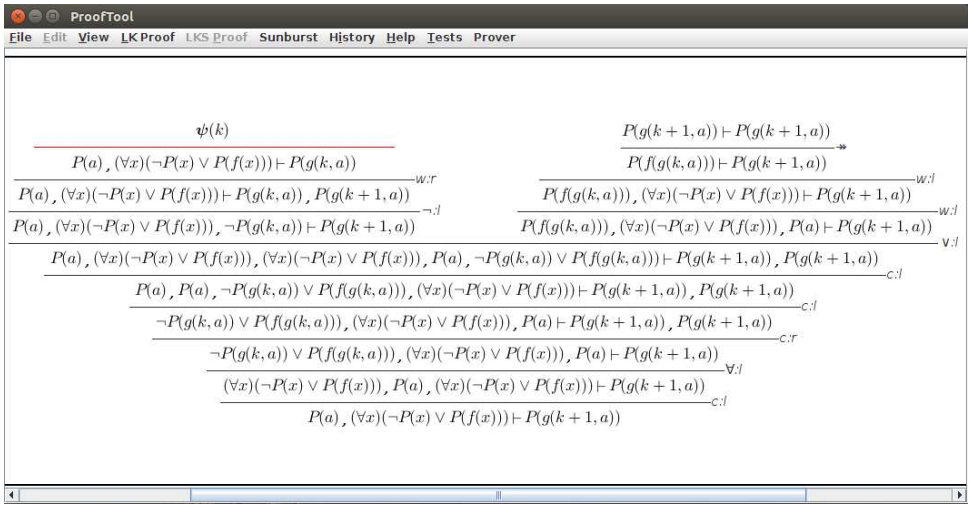

Figure 2.4: A proof for the recursive case is found.

description of the format we refer an interested reader to [8].

GAPT system implements a non-invertible version of LK, but there are so called *macro rules*, which simulate invertible rules using additional weakening and contraction rules. Thus the actual output of the schematic prover contains some redundant structural rules. There exists a function in GAPT removing such redundancies from proofs, but at the time of this writing it significantly affects performance of the prover.

The algorithm is under implementation. Currently the prover succeeds to find only "simple" proof schemata. Under the term "simple" we mean a proof schema, which consists only of one proof pair. This means that the LKS-proof of recursive case contains proof axioms referring to itself only. The following example illustrates the problem.

Example 4.1 *Consider a simple modification of the sequent from Example 2.4:*

$$P(a), \forall x(\neg P(x) \vee P(f(x))) \vdash P(g(k,x)) \wedge P(a).$$

Our prover produces a trivial proof for the base case, but fails on the recursive case. According to the algorithm, it first decomposes right-hand side formula, producing the following (structural rules are omitted):

At this point, the prover tries to further decompose the left-hand side of the derivation, thus failing to prove the statement. Instead, a proper behaviour would be to place a new proof axiom and schedule a proof-search for $P(a), \forall x(\neg P(x) \vee P(f(x))) \vdash P(g(k,x))$. *Our future work is concentrated on this problem.*

Finally, we would like to mention that the prover is integrated in ProofTool. The Prover>Start menu item opens a window, where user can type a sequent schema to be proved (see Figure 2.2). After clicking on the Parse button, the text

input is parsed and the sequent schema is passed to the algorithm, which tries to prove the base case first. If the proof is found, it appears on the screen (see Figure 2.3), and the prover continues to prove the recursive case. If the proof of the recursive case is found, it is also displayed (see Figure 2.4). Otherwise an error message appears, describing the problem.

5 Conclusions

We presented a proof-search algorithm for first-order formula schemata. Its implementation is an ongoing work, so far succeeding to obtain only "simple" proof schemata. The main problem in implementation is to decide when to close a branch with a proof axiom referring to a new proof symbol.

For the future, we plan to continue development in two directions, that complement each other:

1. Add interactivity to the prover via ProofTool. A user will have a possibility to decide when to close a branch with a proof axiom, decide for a proper substitution term, etc., on a point-and-click basis.
2. Make the prover more clever, by designing a reasonable algorithm, making a proper decision without user interaction. Note that naive algorithm can lead to placing a new proof axiom after every decomposition step. Although it is a legal approach, it is completely inefficient.

Acknowledgment. The work was supported by Shota Rustaveli National Science Foundation project no. FR/51/4-120/13.

References

1. Aravantinos V, Caferra R, Peltier N. A Schemata Calculus for Propositional Logic. *In Tableaux'09*, volume 5607 of *LNCS*, pages 32–46, 2009.
2. Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. RegSTAB: A SAT-Solver for Propositional Iterated Schemata. In *International Joint Conference on Automated Reasoning*, pages 309–315, 2010.
3. Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. Decidability and Undecidability Results for Propositional Schemata. *Journal of Artificial Intelligence Research*, 40:599–656, 2011.
4. Serge Autexier, Heiko Mantel, and Werner Stephan. Simultaneous quantifier elimination. In *KI-98: Advances in Artificial Intelligence*, pages 141–152. Springer, 1998.
5. David Baelde, Dale Miller, and Zachary Snow. Focused Inductive Theorem Proving. In *Automated Reasoning*, pages 278–292. Springer, 2010.
6. Hong Chen, Jieh Hsiang, and Hwa-Chung Kong. On finite representations of infinite sequences of terms. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems*, volume 516 of *Lecture Notes in Computer Science*, pages 99–114. Springer Berlin Heidelberg, 1991.

7. Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, and Bruno Woltzenlogel-Paleo. System Feature Description: Importing Refutations into the GAPT Framework. In David Pichardie and Tjark Weber, editors, *Second International Workshop on Proof Exchange for Theorem Proving (PxTP 2012)*, volume 878 of *CEUR Workshop Proceedings*, pages 51–57, 2012.

8. Cvetan Dunchev, Alexander Leitsch, Tomer Libal, Martin Riener, Mikheil Rukhaia, Daniel Weller, and Bruno Woltzenlogel-Paleo. ProofTool: a GUI for the GAPT Framework. In Cezary Kaliszyk and Christoph Lüth, editors, *Proceedings 10th International Workshop On User Interfaces for Theorem Provers (UITP 2012)*, volume 118 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–14, 2013.

9. Cvetan Dunchev, Alexander Leitsch, Mikheil Rukhaia, and Daniel Weller. Cut-elimination and proof schemata. In Martin Aher, Daniel Hole, Emil Jeřábek, and Clemens Kupke, editors, *Logic, Language, and Computation*, volume 8984 of *Lecture Notes in Computer Science*, pages 117–136. Springer Berlin Heidelberg, 2015.

10. Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. System Description: GAPT 2.0. In Nicola Olivetti and Ashish Tiwari, editors, Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Proceedings, volume 9706 of *Lecture Notes in Computer Science*, pages 293–301. Springer International Publishing, 2016.

11. Jean-Yves Girard. A new constructive logic: classical logic. In *Mathematical Structures in Computer Science*, volume 1, pages 255–296. Cambridge Univ Press, 1991.

12. Stefan Hetzl, Alexander Leitsch, Giselle Reis, Janos Tapolczai, and Daniel Weller. Introducing quantified cuts in logic with equality. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2014.

13. Stefan Hetzl, Tomer Libal, Martin Riener, and Mikheil Rukhaia. Understanding Resolution Proofs through Herbrand's Theorem. In Didier Galmiche and Dominique Larchey-Wendling, editors, *Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux 2013)*, volume 8123 of *Lecture Notes in Computer Science*, pages 157–171, 2013.

14. Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.

15. Tomer Libal, Martin Riener, and Mikheil Rukhaia. Advanced Proof Viewing in ProofTool. In Christoph Benzmüller and Bruno Woltzenlogel Paleo, editors, *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, (UITP 2014)*, volume 167 of *Electronic Proceedings in Theoretical Computer Science*, pages 35–47, 2014.

16. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima, Inc., 2nd edition, 2010.

17. Mikheil Rukhaia. *About Cut-Elimination in Schematic Proofs. A monograph*. Lambert Academic Publishing, Saarbrücken, 2013.

18. Gaisi Takeuti. *Proof Theory*. North Holland, second edition, 1987.