Kalpa Publications in Computing
Volume 18, 2024, Pages 107–116
LPAR 2024 Complementary Volume

A Natural–style Prover in *Theorema* Using Sequent Calculus with Unit Propagation

Tudor Jebelean[1]

ICAM, West University of Timişoara, Romania
RISC, Johannes Kepler University, Linz, Austria
Tudor.Jebelean@e-uvt.ro

Abstract

For tutorial purposes we realized a propositional prover in the *Theorema* system that works in natural style and is based on sequent calculus with unit propagation.

The version of the sequent calculus that we use is a reductionist one: at each step a formula is decomposed according to a fixed rule attached to its main logical connective. Optionally the prover may use unit propagation. Unit propagation in sequent calculus is a novel inference method introduced by the author that consists in propagating literals that occur either as antecedents of postcedents in the sequent: all occurrences of such a literal in any of the other formulae are replaced by the corresponding truth value and the respective formula is simplified by rewriting.

By natural style we understand a style similar to human activity. This applies to syntax of formulae, to inference steps, and to proof presentation. Although based on sequent calculus, the prover does not produce proofs as sequent proof trees, but as natural–style narratives.

The purpose of the tool is tutorial for the understanding of natural–style proving, of sequent calculus, and of the implementation in the *Theorema* system as a set of rewrite rules for the inferences and a set of accompanying patterns for the explanatory text produced by the prover.

1 Introduction

Education in computer science and mathematics should (and often does) greatly benefit from a solid basis in mathematical logic. In the opinion of the author, the lack of massive usage of software verification tools originates mainly in the lack of sufficient education in logic, which makes software professionals reticent in using such tools.

This work present a tutorial realization of a propositional prover, with the purpose of supporting the student understanding of (a) natural style proving, (b) sequent calculus, and (c) the concrete implementation of the prover in *Theorema*. These three aspects help the students to (a) improve their proving skills, (b) acquire the knowledge about a classical proof method that is close to human proof style, and (c) study the implementation principles and the functioning of a mechanical prover.

The *Theorema* system[2, 3, 8] is built upon *Mathematica*[1] with the aim of supporting the processes of: defining mathematical theories (including definition of algorithms by logical formulae), experimenting by running the algorithms, and developing and using automatic provers. A distinctive feature of the *Theorema* system is the use of natural style (similar to human) for expressing the logical formulae and the algorithms, for the inference rules of the provers, and for the presentation of the proofs.

2 Sequent Calculus

Sequent calculus was introduced as an alternative to other proof systems (e. g. Hilbert) for the purpose of rendering the proofs more human readable by Gentzen [5, 6] and has been developed later in various versions – see [4] for an early survey. We use a concept of sequent calculus where both the antecedents and the postcedents are seen as *sets*, as opposed to lists in other approaches.

We use a particular version of sequent calculus, namely a reductionist one: the proof is developed bottom–up starting from the root sequent, and every step consists in decomposing one of the formulae by applying a sequent rule associated to the main logical connective of it. A practically identical calculus is presented at http://logitext.mit.edu/logitext.fcgi/tutorial together with an interactive implementation. In contrast with this approach and the usual ones, our calculus uses conjunctive and disjunctive sets instead of the classical binary conjunction and disjunction. The rules of our calculus are presented below.

Full	Assumptions	Goal
a	$\dfrac{}{\Phi, \gamma \vdash \gamma, \Psi}\ \text{a}$	
$\mathbb{T}$	$\dfrac{\Phi \vdash \Psi}{\Phi, \mathbb{T} \vdash \Psi}\ \mathbb{T}\vdash$	$\dfrac{}{\Phi \vdash \mathbb{T}, \Psi}\ \vdash\mathbb{T}$
$\mathbb{F}$	$\dfrac{}{\Phi, \mathbb{F} \vdash \Psi}\ \mathbb{F}\vdash$	$\dfrac{\Phi \vdash \Psi}{\Phi \vdash \mathbb{F}, \Psi}\ \vdash\mathbb{F}$
$\neg$	$\dfrac{\Phi \vdash \gamma, \Psi}{\Phi, \neg\gamma \vdash \Psi}\ \neg\vdash$	$\dfrac{\Phi, \gamma \vdash \Psi}{\Phi \vdash \neg\gamma, \Psi}\ \vdash\neg$
$\wedge$	$\dfrac{\Phi, \Gamma \vdash \Psi}{\Phi, \wedge\Gamma \vdash \Psi}\ \wedge\vdash$	$\dfrac{\{\Phi \vdash \gamma, \Psi \mid \gamma \in \Gamma\}}{\Phi \vdash \wedge\Gamma, \Psi}\ \vdash\wedge$
$\vee$	$\dfrac{\{\Phi, \gamma \vdash \Psi \mid \gamma \in \Gamma\}}{\Phi, \vee\Gamma \vdash \Psi}\ \vee\vdash$	$\dfrac{\Phi \vdash \Gamma, \Psi}{\Phi \vdash \vee\Gamma, \Psi}\ \vdash\vee$
$\Rightarrow$	$\dfrac{\Phi \vdash \gamma_1, \Psi \qquad \Phi, \gamma_2 \vdash \Psi}{\Phi, \gamma_1 \Rightarrow \gamma_2 \vdash \Psi}\ \Rightarrow\vdash$	$\dfrac{\Phi, \gamma_1 \vdash \gamma_2, \Psi}{\Phi \vdash \gamma_1 \Rightarrow \gamma_2, \Psi}\ \vdash\Rightarrow$
$\Leftrightarrow$	$\dfrac{\Phi \vdash \gamma_1, \gamma_2, \Psi \qquad \Phi, \gamma_1, \gamma_2 \vdash \Psi}{\Phi, \gamma_1 \Leftrightarrow \gamma_2 \vdash \Psi}\ \Leftrightarrow\vdash$	$\dfrac{\Phi, \gamma_1 \vdash \gamma_2, \Psi \qquad \Phi, \gamma_2 \vdash \gamma_1, \Psi}{\Phi \vdash \gamma_1 \Leftrightarrow \gamma_2, \Psi}\ \vdash\Leftrightarrow$

In this table we use φ, ψ, γ for individual formulae and Φ, Ψ, Γ for sets of formulae. Conjunctive (disjunctive) sets are represented by placing the conjunction (disjunction) connective before the set.

[1] www.wolfram.com/mathematica

Unit Propagation. Another distinctive feature of our approach is unit propagation. Unit propagation in sequent calculus is essentially the same as in SAT solvers: when a literal is among the antecedents then its variable receives the truth value that makes the literal true, all its occurrences are replaced by this value, and the corresponding formulae are simplified until the truth constants disappear or they become a truth constant. In SAT solving this operation reduces to deleting some literals and some clauses, while in sequent calculus the transformations are more complex because they are applied to arbitrary formulae. Furthermore, in sequent calculus we also have situations when the literal occurs among the postcedents. In this case the corresponding variable is assigned the truth value that makes the literal false. The simplification rules are presented in the next section by their implementation.

Unit propagation makes proofs simpler, in particular by reducing the number of branches.

Example 1: A proof using unit propagation.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{a}
\UnaryInfC{$\neg B \;\vdash\; \neg B$}
\RightLabel{Simplify}
\UnaryInfC{$\neg(\mathbb{T}\wedge B) \;\vdash\; \neg B$}
\RightLabel{$A\to\mathbb{T}$}
\UnaryInfC{$\neg(A\wedge B), A \;\vdash\; \neg B$}
\RightLabel{Simplify}
\UnaryInfC{$(A\wedge B)\Rightarrow\mathbb{F}, A \;\vdash\; B\Rightarrow\mathbb{F}$}
\RightLabel{$C\to\mathbb{F}$}
\UnaryInfC{$(A\wedge B)\Rightarrow C, A \;\vdash\; C, B\Rightarrow C$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$(A\wedge B)\Rightarrow C \;\vdash\; A\Rightarrow C, B\Rightarrow C$}
\RightLabel{$\vdash\vee$}
\UnaryInfC{$(A\wedge B)\Rightarrow C \;\vdash\; (A\Rightarrow C)\vee(B\Rightarrow C)$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$\vdash\; ((A\wedge B)\Rightarrow C)\Rightarrow((A\Rightarrow C)\vee(B\Rightarrow C))$}
\end{prooftree}
$$

Example 2: The proof of the same sequent without unit propagation.

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{a}
\UnaryInfC{$A, B \;\vdash\; A, C$}
\AxiomC{}
\RightLabel{a}
\UnaryInfC{$A, B \;\vdash\; B, C$}
\RightLabel{$\vdash\wedge$}
\BinaryInfC{$A, B \;\vdash\; A\wedge B, C$}
\AxiomC{}
\RightLabel{a}
\UnaryInfC{$C, A, B \;\vdash\; C$}
\RightLabel{$\Rightarrow\vdash$}
\BinaryInfC{$(A\wedge B)\Rightarrow C, A, B \;\vdash\; C$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$(A\wedge B)\Rightarrow C, A \;\vdash\; C, B\Rightarrow C$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$(A\wedge B)\Rightarrow C \;\vdash\; A\Rightarrow C, B\Rightarrow C$}
\RightLabel{$\vdash\vee$}
\UnaryInfC{$(A\wedge B)\Rightarrow C \;\vdash\; (A\Rightarrow C)\vee(B\Rightarrow C)$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$\vdash\; ((A\wedge B)\Rightarrow C)\Rightarrow((A\Rightarrow C)\vee(B\Rightarrow C))$}
\end{prooftree}
$$

Example 3: failed proof of an invalid sequent with unit propagation.

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$\vdash$}
\RightLabel{$\mathbb{T}\vdash$}
\UnaryInfC{$\mathbb{T} \;\vdash$}
\RightLabel{Simplify}
\UnaryInfC{$(\neg\mathbb{T})\vee(\mathbb{T}\Rightarrow\mathbb{T}) \;\vdash$}
\RightLabel{$A\to\mathbb{T}, B\to\mathbb{T}$}
\UnaryInfC{$A, B, (\neg A)\vee(B\Rightarrow A) \;\vdash$}
\RightLabel{$\wedge\vdash$}
\UnaryInfC{$A\wedge B, (\neg A)\vee(B\Rightarrow A) \;\vdash$}
\RightLabel{Simplify}
\UnaryInfC{$A\wedge B, (A\Rightarrow\mathbb{F})\vee(B\Rightarrow A) \;\vdash$}
\RightLabel{$C\to\mathbb{F}$}
\UnaryInfC{$A\wedge B, (A\Rightarrow C)\vee(B\Rightarrow A) \;\vdash\; C$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$(A\Rightarrow C)\vee(B\Rightarrow A) \;\vdash\; (A\wedge B)\Rightarrow C$}
\RightLabel{$\vdash\Rightarrow$}
\UnaryInfC{$\vdash\; ((A\Rightarrow C)\vee(B\Rightarrow A))\Rightarrow((A\wedge B)\Rightarrow C)$}
\end{prooftree}
$$

The proof of an invalid branch reduces to the empty sequent, and the counterexample (falsifying interpretation) is given by the assignment of the variables of the propagated literals.

3 Prover Implementation

In *Theorema* any prover implementation consists in two files: the prover itself that specifies the inference rules, and the language file that specifies the explanatory text of the displayed proof.

The simplification engine. Below we present the simplification rules written in *Mathematica* that are part of the prover. Note that in *Mathematica* function application is denoted with square brackets, and lists with curly brackets.

```
truthConstantSimplificationRules = {
 Not$TM[ True] :> False,
 Not$TM[ False] :> True,
 Not$TM[Not$TM[ fml_]] :> fml,
 And$TM[ ___, False, ___] :> False,
 And$TM[ pre___, fml_, rest___, fml_, post___] :> And$TM[ pre, fml, rest, post],
 And$TM[ pre___, fml_, rest___, Not$TM[ fml_], post___] :> False,
 And$TM[ pre___, Not$TM[ fml_], rest___, fml_, post___] :> False,
 And$TM[ pre___, True, post___] :> And$TM[ pre, post],
 And$TM[ fml_] :> fml,
 And$TM[] :> True,
 Or$TM[ ___, True, ___] :> True,
 Or$TM[ pre___, fml_, rest___, fml_, post___] :> Or$TM[ pre, fml, rest, post],
 Or$TM[ ___, fml_, ___, Not$TM[ fml_], ___] :> True,
 Or$TM[ ___, Not$TM[ fml_], ___, fml_, ___] :> True,
 Or$TM[ pre___, False, post___] :> Or$TM[ pre, post],
 Or$TM[ fml_] :> fml,
 Or$TM[] :> False,
 Implies$TM[ True, fml_] :> fml,
 Implies$TM[ fml_, True] :> True,
 Implies$TM[ False, fml_] :> True,
 Implies$TM[ fml_, False] :> Not$TM[ fml],
 Implies$TM[ fml_, fml_] :> True,
 Implies$TM[ fml_, Not$TM[ fml_]] :> Not$TM[ fml],
 Implies$TM[ Not$TM[ fml_], fml_] :> fml,
 Iff$TM[ True, fml_] :> fml,
 Iff$TM[ fml_, True] :> fml,
 Iff$TM[ False, fml_] :> Not$TM[ fml],
 Iff$TM[ fml_, False] :> Not$TM[ fml],
 Iff$TM[ fml_, fml_] :> True,
 Iff$TM[ fml_, Not$TM[ fml_]] :> False,
 Iff$TM[ Not$TM[ fml_], fml_] :> False
};
```

The *Mathematica* evaluation engine is rewriting based, thus one can write programming constructs that are essentially logical formulae. However the engine is a little more general: it is based on pattern matching. A rewrite rule as in the table above applies to any expression that matches the LHS and it will transform it into the corresponding instance of the RHS. In *Mathematica* an underscored symbol matches any expression, for instance `fml_` matches an arbitrary formula. Underscored symbols are essentially predicate logic variables. Furthermore a triple underscore matches a sequence of expressions, including the empty sequence. Triple underscored symbols are *sequence variables* and predicate logic can be extended to handle them – see [1, 7].

For instance the rule:

```
And$TM[ pre___, True, post___] :> And$TM[ pre, post]
```

applies when `True` occurs anywhere among the arguments of the conjunction, and it will eliminate it.

If some pattern elements are not needed on the RHS, then one can use *anonymous* patterns like below. The rule:

```
And$TM[ ___, False, ___] :> False
```

applies when `False` occurs anywhere among the arguments of the conjunction.

The rule:

```
And$TM[ pre___, fml_, rest___, fml_, post___] :> And$TM[ pre, fml, rest, post]
```

removes the second occurrence of `fml`.

Note that unitary and empty conjunctions and disjunctions are also handled.

The prover. In *Theorema* any prover consists in a set of inference rules and a set of declarations regarding their grouping, visibility, and priority. Additionally the file contains the definition of various auxiliary functions and objects that are necessary, like for instance the set of simplification rules shown above.

The inference rules of our prover express the sequent rules of the calculus.

Let us look for instance at the rule “implication in the postcedent” that is shown below.

```
inferenceRule[ goalImplies] =
PRFSIT$[ FML$[ _,
               Goals$TM[ pre___,
                         g:FML$[ _, Implies$TM[ lf_, rg_], lab_, ___],
                         post___],
              ___],
         k_List,
         id_,
         rest___?OptionQ] :>
performProofStep[
   Module[ {asm, gl, goalList},
      asm = makeAssumptionFML[formula -> lf];
      gl = makeGoalFML[formula -> rg];
      goalList = makeGoalFML[formula -> Goals$TM[ pre, gl, post]];
      makeANDNODE[
         makePRFINFO[ name -> goalImplies, used -> g,
                      generated -> {{asm}, {gl}, {goalList}}],
         toBeProved[ goal -> goalList, kb -> Prepend[k, asm], rest]
      ] (* makeANDNODE *)
   ] (* Module *)
]; (* performProofStep *)
```

The implemented name of the rule is `goalImplies` and this will be used when grouping and parametrising it as explained in the sequel.

The whole expression assigns (by `=`) a value to `inferenceRule[ goalImplies]`. The value assigned is a rewrite rule *LHS* `:>` *RHS*.

The left hand side (*LHS*) describes the proof situation to which this rule applies. A proof situation is essentially a sequent, but for practical reasons it also contains some additional information. In *Theorema* a proof situation is represented as `PRFSIT$`[2]. The most important parts of the proof situation are the goal and the list of assumptions, they correspond the the postcedents and the antecedents of a sequent. In contrast to sequent calculus, *Theorema* allows only one goal, thus we needed to hack a little in order to be able to represent several postcedent – see below.

The goal is a *Theorema* formula, which is represented by the construct `FML$`. This contains a first element (not important for this inference rule, represented by the underscore), then the logical formula itself, then the label of the formula (necessary for displaying the proof) and some other elements that are not important for this inference rule (represented by `___`). This is the goal of the proof situation, which consists of a set of postcedents. For representing them we use the construct `Goals$TM` that contains a list of *Theorema* formulae.

For this inference rule we specify that an implication occurs in the list, by using sequence variables `pre___` and `post___` for the other postcedents. The postcedent containing the implication is a *Theorema* formula named `g`[3] and has the label `lab`. The logical formula inside this is an implication with the LHS named `lf` and the RHS named `rg`. Furthermore the proof situation contains the list of antecendents named `k_List`[4] and some other arguments that are not important for this inference rule[5].

The right hand side describes the proof situation[s] that is [are] produced by the rule (the resulting sequent[s]). All functions present here (except `Module`) are provided by the *Theorema* system and can be used by any prover. The RHS must call `PerformProofStep` and this must be applied to an expression returning `makeANDNODE` or `makeORNODE` (with the obvious meaning). In this rule the result is computed using the *Mathematica* construct `Module`: this allows to write a kind of subprogam having some local variables (in our case `asm`, `gl`, and `goalList`) and a list of operations from which the last one provides the return value. We can see that the subprogram constructs a new assumption (antecedent) from `lf`, a new goal from `rg` and a new list of goals in which the new goal takes the place of the decomposed implication.

In `makeANDNODE` the first argument `makePRINFO` contains the elements that are necessary for the display of this proof step, while `toBeProved` specifies the essential elements of the next proof situation. In the case of a branching node we will have several arguments of this kind.

The prover contains five groups of rules that we describe below.

- `terminateRules` for successful proof termination:
  - `goalTrue` – true postcedent,
  - `kbFalse` – false antecedent,
  - `goalKB` – an antecedent occurs as postcedent also,
  - `kbContra` – contradictory antecedents,
  - `goalCompl` – opposite postcedents.
- `simplifyRules` for reduction of the sequent:

[2] In *Theorema* the name of the constants end in `$` or `$TM`. Constants used as heads of expressions play the role of constructors of various data structures handled by the system.

[3] This is a *Mathematica* construct that allows to name a part of the LHS pattern in order to be able to use its corresponding instantiation on the RHS of the rule.

[4] This means it must be a *Mathematica* list.

[5] Various provers may store here additional information.

  - `kbTrue` – eliminate true antecedent,
  - `kbLiteral` – propagate literal antecedent (the antecedent is removed, its variable is assigned the truth value that makes the literal true, the variable is replaced in all other formulae and these are simplified),
  - `goalLiteral` – propagate literal postcedent (the postcedent is removed, its variable is assigned the truth value that makes the literal false, the variable is replaced in all other formulae and these are simplified).

- `decomposeRules` for decomposition of composite formulae:
  - `goalNot` – negated postcedent,
  - `kbNot` – negated antecedent,
  - `kbAnd` – conjunctive antecedent,
  - `goalImplies` – implicative postcedent,
  - `goalOr` – disjunctive postcedent.

- `splitRules` for branching of the sequent proof:
  - `goalAnd` – conjunctive postcedent,
  - `kbOr` – disjunctive antecedent,
  - `goalIff` – equivalence as postcedent,
  - `kbImplies` – implicative antecedent,
  - `kbIff` – equivalence as antecedent,

- `optionRules` are not really rules, they are used for the interactive setting of some options to the prover:
  - `showRemoved` – display messages referring to removal of formulae,
  - `showGoalList` – display the list of postcedents whenever it changes.

For each rule specified by its name the implementation sets whether it is active, whether the step is displayed, its priority, and whether it is a terminating rule. Except the latter, all other parameters can be changed interactively when using the prover – see below in section "Using the Prover".

In particular the rules that perform unit propagation can be disactivated interactively, then the prover applies the usual sequent calculus.

Finally the rules are registered the in the *Theorema* system.

Here we see two additional rules that are not in the previously described groups:

- `makeGoalList` is applied only once as the first proof step, it creates the list of postcedents and other elements of the initial proof situation,

- `failUP` applies when no rule is applicable, thus the proof fails.

Notably, no modification must occur in any of the programs composing *Theorema* in order to add a new prover to the system. One only needs to place the files of the prover in the appropriate directories.

```
registerRuleSet[ "Propositional Prover by Unit Propagation",
unitPropositionalRules, {
 {makeGoalList, True, True, 1},
 terminateRules,
 simplifyRules,
 decomposeRules,
 splitRules,
 optionRules,
 {failUP, True, True, 100, "term"}
 }];
```

The Language File. This file specifies how the proof text will look like in a certain language (more languages can be used.). This file contains the defintions of various titles that have to be displayed by the *Theorema* interface – the "Theorema Commander" – see section "Using the Prover", possibly some definitions of auxiliary functions, and most importantly a clause like the one shown below for each inference rule of the prover.

```
proofStepText[ goalImplies, lang, {{ g_}}, {{asm_}, {gl_}, {goalList_}}, ___] :=
Join[{ textCell[ "Implicative goal ",
formulaReference[g],
" is split. Assume:"
],
assumptionCell[ asm, ","],
textCell[ "and prove:"],
goalCell[ gl, ","]
},
goalListCell[ goalList]
];
```

The clause consists in a definition of the *Theorema* function `proofStepText`[6].

All the other functions in this clause, except `Join`, are provided by the *Theorema* system. Note the correspondence between the arguments of this function and the arguments of `makePRFINFO` in the implementation of the rule shown before.

All elements of this clause are self explanatory, except maybe `lang`: this is the language of the displayed text and in this case it is set to `English` in the surrounding context of this clause.

4 Using the Prover

For using the prover one must first install the *Mathematica* system, available from https://www.wolfram.com/mathematica/.

The *Theorema* system is available at http://www3.risc.jku.at/research/theorema/software/ and a comprehensive tutorial can be found at http://www3.risc.jku.at/research/theorema/software/documentation/tutorial/FirstTour.html. After you install *Theorema* you will have a directory `Theorema` under the directory (in Windows):

```
C:\Program Files\Wolfram Research\Mathematica\11.3\AddOns\Applications
```

[6]In *Mathematica* a function may have multiple defining clauses of the form function[pattern]:=expression, which are treated as rewrite rules. Whenever a subexpression of the currently evaluating expression matches function[pattern], it will be replaced by the corresponding instance of expression.

and something similar in Unix (11.3 is the *Mathematica* version, could be different on your system).

The propositional prover with unit propagation is available as a `zip` file at https://www.risc.jku.at/people/tjebelea/UnitProp.html, which will uncompress into a directory `UnitProp`.

Copy the file

```
UnitProp\Unit-Propositional-Prover\UnitPropositional.m
```

to

```
Theorema\Provers\
```

Copy the file

```
UnitProp\Unit-Propositional-Prover\English\UnitPropositional.m
```

to

```
Theorema\Provers\LanguageData\English
```

and similar for the German version.

Now after launching *Theorema* you will be able to choose the prover in the "Theorema Commander". This has a friendly user interface that allows to construct examples, to fine–tune the prover by [dis]activating certain rules, setting the rule priorities, etc.

For start one can use the examples from

```
UnitProp\Unit-Propositional-Examples
```

and one can look up the proofs from

```
UnitProp\Unit-Propositional-Proofs
```

which are also in pdf format for being readable without *Mathematica*.

Mathematica notebook files (extension `.nb`) may also be viewed with Wolfram CDF player, available free of charge at https://www.wolfram.com/player/.

The prover files (extension `.m`) are in plain text format, thus they can be inspected with any text editor.

References

[1] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema project: A progress report. In *Calculemus 2000*, pages 98–113. A.K. Peters, Natick, Massachusetts, 2000.

[2] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey on the Theorema project. In *In International Symposium on Symbolic and Algebraic Computation*, pages 384–391. ACM Press, 1997.

[3] B. Buchberger, T. Jebelean, T. Kutsia, A. Maletzky, and W. Windsteiger. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *Journal of Formalized Reasoning*, 9(1):149–185, 2016.

[4] Samuel R. Buss. *An introduction to proof theory*, page 1–78. Elsevier, 1998.

[5] Gerhard Gentzen. Untersuchungen über das logische schließen. I. *Mathematische Zeitschrift*, 39:176–210, 1935.

[6] Gerhard Gentzen. Untersuchungen über das logische schließen. II. *Mathematische Zeitschrift*, 39:405–431, 1935.

[7] T. Kutsia and B. Buchberger. Predicate logic with sequence variables and sequence function symbols. In *Proc. of the 3rd Int. Conference on Mathematical Knowledge Management. Vol. 3119 of LNCS*, pages 205–219. Springer, 2004.

[8] W. Windsteiger. Theorema 2.0: A system for mathematical theory exploration. In *ICMS'2014*, volume 8592 of *LNCS*, pages 49–52, 2014.