GAPT

Generic Architecture for Proof Transformations

User Manual

September 7, 2015

Stefan Hetzl - stefan.hetzl@tuwien.ac.at
Bernhard Mallinger - bernhard@logic.at
Giselle Reis - giselle@logic.at
Janos Tapolczai - e0825077@student.tuwien.ac.at
Daniel Weller - weller@logic.at

Contents

1 Introduction

GAPT is a generic architecture for proof transformations implemented in Scala. Formulas, terms, and other expressions are are represented as lambda terms in simple type theory. Terms and formulas of first-order logic and schematic first-order logic are hence encoded as lambda terms, these form regular subsets.

The focus of GAPT are proof transformations (in contrast to proof assistants whose focus is proof formalization and automated deduction systems whose focus is proof search). GAPT is used from a shell which provides access to the functionality in the system in a way that is inspired by computer algebra systems: the basic objects are formulas and (different kinds of) proofs which can be modified by calling GAPT-commands from the command-line. In addition, there is a graphical user interface which allows to view (and up to a certain extent: also to modify) proofs in a flexible and visually appealing way.

The current functionality of GAPT includes data structures for formulas, sequents, resolution proofs, sequent calculus proofs, expansion tree proofs and algorithms for e.g. unification, proof skolemization, cut-elimination, cut-elimination by resolution [1], cut-introducton [4], etc.

This document describes the system from the perspective of a user who has downloaded a jar-file. For information on how to get started as a developer, please see the developer wiki at `https://github.com/gapt/gapt/wiki`.

2 System Requirements

To run GAPT you need to have the Java runtime environment 1.6.0 or higher installed. We had problems with OpenJDK, but you can use it at your own risk.

GAPT contains interfaces to the following automated reasoning systems. Installing them is optional. If GAPT does not find the executables in the path, the functionality of these systems will not be available.

- Prover9 (`http://www.cs.unm.edu/~mccune/mace4/download/`) - make sure the commands `prover9`, `prooftrans` and `tptp_to_ladr` are available.
- VeriT (`http://www.verit-solver.org/`)
- Minisat (`http://minisat.se/`)

3 Downloading and Running the System

You can download a pre-packaged jar-file of the current version of GAPT at `http://www.logic.at/gapt/downloads`. After having downloaded the gapt-cli zip-file you will find a shellscript `cli.sh`.

Running this script

```
./cli.sh
```

will start the command-line interface of GAPT.

The command-line interface of GAPT runs in an interactive Scala-shell. This has the consequence that all functionality of Scala is available to you. In particular it is easy to write Scala-scripts that use the functionality of GAPT.

Here are some useful things you should know about the Scala shell.

If you want to assign the result of a method to a variable, use: `var v = method(args)`. Otherwise, the system will create a variable by itself. You can see it's name and type before the description, like this:

With a variable name:

```
gapt> var i = 12
i: Int = 12
```

Without a variable name (in this case, the system created the variable `res12`):

```
gapt> 12
res12: Int = 12
```

To see the value of a variable, just type it's name and press enter.

The elements of a list in Scala are indexed using parenthesis. So if `lst` is a list, the first element is obtained by `lst(0)`. It is also possible to use the methods `lst.head` (for the first element) and `lst.tail` (for the rest of the list).

The elements of a tuple `t` of size n are accessed by the methods `t._1`, `t._2`, `...` , `t._n`.

To quit the interactive shell, just type `:quit` and press enter.

In order to see the list of all available commands, type `help` and press enter.

4 File Input/Output

This method `loadProofs` will take as an argument a string that represents the path of a file containing a proofdatabase in the xml format (generated by HLK for example), and will return a list of pairs. It expects a file (the string of a proof will not work) and you can use the relative path. On the list returned, each pair is composed of a string and an object representing a proof within the system. The string is the name of the proof defined on the xml file. For example:

```
gapt> val proofs = loadProofs( "examples/simple/fol1.xml.gz" )
```

returns a list of length 1 as shown by entering

```
gapt> proofs.length
```

Its only element is `proofs(0)`, the name of this proof can be obtained by entering

```
gapt> proofs(0)._1
```

and the proof itself by

```
gapt> val proof = proofs(0)._2
```

You can then view this proof in the graphical user interface prooftool by entering

```
gapt> prooftool( proof )
```

In the folder `../examples/simple` you can find a number of further simple examples that illustrate different aspects of GAPT.

The command `exportXML` Exports several proofs to an XML file. The first argument is the list of proofs, the second is the list of the names of the proofs and the third is the name of the file that will be written. The file path can also be specified and it's relative to where the program is being executed.

The code below exports the proofs in the variables $p1$ and $p2$ to the file `result.xml` with names "First proof" and "Second proof", respectively.

```
gapt> exportXML(p1::p2::Nil, "First proof"::"Second proof"::Nil, "result.xml")
```

5 Entering Formulas

Formulas can be entered as strings which are parsed as follows:

```
gapt> val F = parseFOL( "Imp A Imp B And A B" )
```

For a proper first-order formula consider for example:

```
gapt> val G = parseFOL( "Forall x Imp P(x,f(x)) Exists y P(x,y)" )
```

Also prover9 syntax[6] is supported:

```
gapt> val H = parseProver9( "(all x (P(x,f(x)) -> (exists y P(x,y))))" )
```

The prover9 syntax was also extended to higher-order logic, where type declarations are added:

```
gapt> val I = parseLLKFormula ( "var P:o>i>o; const f:o>i; var x:o; var y:i; (
    all x (P(x,f(x))) -> (exists y P(x,y)))" )
```

Please refere to Appendix C for a full description of the languages the parsers accept.

A collection of formula sequences can be found in the file `../examples/FormulaSequence.scala`. Have a look at this code to see how to compose formulas without the parser. This file is a scala script that can be loaded into the CLI by entering

```
gapt> :load examples/FormulaSequences.scala
```

Then you can generate instances of these formula sequences by entering e.g.

```
gapt> val f = BussTautology( 5 )
```

6 Automated Deduction

6.1 SAT Solving using MiniSAT

The following shows an example session, using the MiniSAT solver to verify valdity and satisfiability, and query the thus obtained models. Consider the *pigeon hole principle for* (m, n), $\mathrm{PHP}_{m,n}$, which states that if m pigeons are put into n holes, then there is a hole which contains two pigeons. It is valid iff $m > n$. $\neg\mathrm{PHP}_{m,n}$ states that when putting m pigeons into n holes, there is no hole containing two pigeons. This is satisfiable iff $m \leq n$. Make sure that the pigeon hole principle is available in your current CLI session by entering

```
gapt> :load examples/FormulaSequences.scala
```

if you have not done so already. Then

```
gapt> miniSATprove(PigeonHolePrinciple(3, 2))
res12: Boolean = true
```

shows that $\mathrm{PHP}_{3,2}$ is valid, and

```
gapt> miniSATprove(PigeonHolePrinciple(3, 3))
res13: Boolean = false
```

shows that $\mathrm{PHP}_{3,3}$ is not valid. Furthermore,

```
gapt> val m = miniSATsolve(Neg(PigeonHolePrinciple(3, 3))).get
```

yields a model of $\neg\mathrm{PHP}_{3,3}$ that can be queried:

```
gapt> val p1 = PigeonHolePrinciple.atom(1, 1)
gapt> val p2 = PigeonHolePrinciple.atom(2, 1)
gapt> val p3 = PigeonHolePrinciple.atom(3, 1)
gapt> m.interpret(p1) // Is pigeon 1 in hole 1?
gapt> m.interpret(p2) // Is pigeon 2 in hole 1?
gapt> m.interpret(p3) // Is pigeon 3 in hole 1?
```

We can also interpret quantifier-free formulas:

```
gapt> m.interpret( And(p1, p2) )
```

6.2 SMT Solving using VeriT

6.3 First-Order Resolution Proving using Prover9

6.4 Built-In Resolution Prover

GAPT contains a built-in resultion prover that can be called with the command: `refuteFOL: Seq[Clause] → Option[ResolutionProof[Clause]]` and with the command `refuteFOLI: Seq[Clause] → Option[ResolutionProof[Clause]]` for interactive mode.

6.5 Built-In Tableaux Prover

GAPT contains a built-in tableaux prover for propositional logic which can be called with the command `proveProp`, for example as in:

```
gapt> proveProp( parse.fol( "Imp A Imp B And A B" ))
```

7 Entering Proofs

There are various possibilities for entering proofs into the system. The most basic one is a direct top-down proof-construction using the constructors of the inference rules. For example, continuing in the environment of Section 5, suppose that we want to enter a proof of `F`. It is convenient to prepare the subformulas first.

```
gapt> val F1 = parse.fol( "Imp B And A B" )
gapt> val F2 = parse.fol( "And A B" )
gapt> val A = parse.fol( "A" )
gapt> val B = parse.fol( "B" )
```

We start with the axioms:

```
gapt> val p1 = Axiom( A::Nil, A::Nil )
gapt> val p2 = Axiom( B::Nil, B::Nil )
```

These are joined by an $\land : \mathrm{right}$-inference. See Appendix A for the formal definition of the sequent calculus used in GAPT.

```
gapt> val p3 = AndRightRule( p1, p2, A, B )
```

To finish the proof it remains to apply two $\supset$: right-inferences:

```
gapt> val p4 = ImpRightRule( p3, B, F2 )
gapt> val p5 = ImpRightRule( p4, A, F1 )
```

You can now view this proof by typing:

```
gapt> prooftool( p5 )
```

The system comes with a collection of example proof sequences in the file `examples/ProofSequences.scala` which are generated in the above style. Have a look at this code for more complicated proof constructions. In order to load these proof sequences into the CLI, enter:

```
gapt> :load examples/ProofSequences.scala
```

8 Proof Theory

8.1 Cut-Elimination (Gentzen's method)

The GAPT-system contains an implementation of reductive cut-elimination à la Gentzen. It can be used as follows: first we load a proof p with cuts (as in Section 4).

```
gapt> val p = loadProofs( "examples/simple/fol1.xml.gz" )(0)._2
```

and then call the cut-elimination procedure:

```
gapt> val q = eliminateCuts( p )
```

8.2 Skolemization

Skolemization consists of replacing the variables bound by strong quantifiers in the end-sequent of a proof by new function symbols thus obtaining a validity-equivalent sequent. In the GAPT-system Skolemization is implemented for proofs and can be used, e.g. as follows:

```
gapt> val proofs = loadProofs( "examples/prime/ceres_xml/prime1-1.xml.gz" )
gapt> val p = proofs(0)._2
gapt> val q = skolemize( p )
```

8.3 Interpolation

The command `extractInterpolant` extracts an interpolant from a sequent calculus proof which may contain atomic cuts and/or equality rules. Currently, we allow only reflexivity axioms or axioms of the form $A \vdash A$. The implementation is based on Lemma 6.5 of [8]. The method expects a proof p and an arbitrary partition of the end-sequent $\Gamma \vdash \Delta$ of p into a "negative part" $\Gamma_1 \vdash \Delta_1$ and a "positive part" $\Gamma_2 \vdash \Delta_2$. It returns a formula I s.t. $\Gamma_1 \vdash \Delta_1, I$ and $I, \Gamma_2 \vdash \Delta_2$ are provable and I contains only such predicate symbols that appear in both, $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$. For example, suppose pr is a proof of $p \vee q \vdash p, q$ by a single $\vee$-left inference (see Section 7 for how to construct such a proof), then you can compute an interpolant as follows:

```
gapt> val s = pr.root
gapt> val npart = Set( s.antecedent( 0 ), s.succedent( 0 ) )
gapt> val ppart = Set( s.succedent( 1 ) )
gapt> val I = extractInterpolant( pr, npart, ppart )
```

8.4 Expansion Trees

Expansion trees are a compact representation of cut-free proofs. They have originally been introduced in [7]. GAPT contains an implementation of expansion trees for higher-order logic including functions for extracting expansion trees from proofs, for merging expansion trees, for pruning and transforming them in various ways and for viewing them in a comfortable way in the graphical user interface.

An expansion tree contains the instances of the quantifiers for a formula. In order to represent an LK-proof we use *expansion sequents*, i.e. sequents of expansion trees.

Suppose we are in the command-line interface and have a value `p` which is a cut-free `LKProof`. We can then extract the expansion sequent of `p` by:

```
gapt> val E = extractExpansionSequent( p )
```

This expansion sequent can then be viewed in the graphical user interface by simply calling:

```
gapt> prooftool( E )
```

Prooftool is then intialized with displaying the end-sequent of `p`, i.e. the shallow sequent of `E`. The user can then selectively expand quantifiers by clicking on them, see [5] for a detailed description.

9 Cut-Elimination by Resolution (CERES)

9.1 First-Order Logic

9.2 Higher-Order Logic

9.3 Schematic First-Order Logic

10 Cut-Introduction

The cut-introduction algorithm as described in [4, 3, 2] is implemented in GAPT for introducing Π_1-cuts into a sequent calculus proof. We will use as input one of the proofs generated by the system, namely, `LinearExampleProof(9)`. But the user can also write his own proofs (see Section 7) and input them to the cut-introduction algorithm.

Make sure that the example proof sequences are available in the current CLI session if you have not done so already by entering

```
gapt> :load examples/ProofSequences.scala
```

and obtain the desired proof by:

```
gapt> val p = LinearExampleProof(9)
```

Then compute a proof with a single cut that contains a single quantifier by:

```
gapt> val q = cutIntro.one_cut_one_quantifier( p )
```

The method `cutIntro.one_cut_many_quantifiers` will proceed as above but will compute a single cut with a block of quantifiers. The method `cutIntro.many_cuts_one_quantifier` uses a reduction to a MaxSAT problem and an external MaxSAT-solver for finding a minimal grammar corresponding to a proof with an arbitrary number of cuts with one quantifier each.

11 Miscellaneous

The method `printProofStats: LKProof` $\rightarrow$ `Unit` takes a proof and prints some statistics about it. It is helpful for getting a first impression of a proof; use it for example as:

```
gapt> printPoofStats( p )
------------- Statistics ---------------
Cuts: 1
Number of quantifier inferences: 8
Number of inferences: 19
Quantifier complexity: 6
----------------------------------------
```

A The Sequent Calculus

This section defines the rule systems used in GAPT. The rules can be constructed via Scala-classes, which create the underlying data structure.

A.1 First-order LK

The rules of first-order LK are listed below. Proof trees are constructed top-down, starting with axioms and with each rule introducing new inferences. The constructing functions do perform sporadic checks, but in general, these do not guarantee well-formed proofs and the burden of correctness lies upon the programmer using them. Due to the top-down construction, quantification and equational rules are *especially* brittle and the programmer must himself take care to replace only the desired terms in a formula. Specifically, the programmer must supply the result of the replacement (the `main`-argument), which is accepted by the rule constructors without question.

Axioms

$$\frac{}{\Gamma, A \vdash \Delta, A}\ \text{(Identity Axiom)}$$

$$\frac{}{\Gamma \vdash \Delta, t = t}\ \text{(Reflexivity Axiom)}$$

Cut

$$\frac{\Gamma \vdash \Delta, A \qquad \Sigma, A \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}\ \text{(cut)}$$

Structural rules

Left rules

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{(w:l)}$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ \text{(c:l)}$$

Right rules

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ \text{(w:r)}$$

$$\frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\ \text{(c:r)}$$

Propositional rules

Left rules

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ (\wedge\text{:l}_1)$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ (\wedge\text{:l}_2)$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi}\ (\vee\text{:l})$$

$$\frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta}\ (\neg\text{:l})$$

$$\frac{\Gamma \vdash \Delta, A \qquad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \supset B \vdash \Delta, \Pi}\ (\supset\text{:l})$$

Right rules

$$\frac{\Gamma \vdash \Delta, A \qquad \Sigma \vdash \Pi, B}{\Gamma, \Sigma \vdash \Delta, \Pi, A \wedge B}\ (\wedge\text{:r})$$

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B}\ (\vee\text{:r}_1)$$

$$\frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B}\ (\vee\text{:r}_2)$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\ (\neg\text{:r})$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \supset B, \Delta}\ (\supset\text{:r})$$

Quantification rules

Left rules

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x A \vdash \Delta}\ (\forall\text{:l})$$

$$\frac{\Gamma, A[y/x] \vdash \Delta}{\Gamma, \exists x A \vdash \Delta}\ (\exists\text{:l})$$

Right rules

$$\frac{\Gamma \vdash A[y/x]\Delta}{\Gamma \vdash \forall x A, \Delta}\ (\forall\text{:r})$$

$$\frac{\Gamma \vdash A[t/x]\Delta}{\Gamma \vdash \exists x A, \Delta}\ (\exists\text{:r})$$

The variable y must not occur free in Γ, Δ or A. The term t must avoid variable capture, i.e. it must not contain free occurrences of variables bound in A.

Equational rules

Left rules

$$\frac{\Gamma \vdash \Delta, s = t \qquad \Sigma, A[T/s] \vdash \Pi}{\Gamma, \Sigma, A[T/t] \vdash \Delta, \Pi}\ (=\text{:l}_1)$$

$$\frac{\Gamma \vdash \Delta, s = t \qquad \Sigma, A[T/t] \vdash \Pi}{\Gamma, \Sigma, A[T/s] \vdash \Delta, \Pi}\ (=\text{:l}_2)$$

Right rules

$$\frac{\Gamma \vdash \Delta, s = t \qquad \Sigma \vdash \Pi, A[T/s]}{\Gamma, \Sigma \vdash \Delta, \Pi, A[T/t]}\ (=\text{:r}_1)$$

$$\frac{\Gamma \vdash \Delta, s = t \qquad \Sigma, \vdash \Pi, A[T/t]}{\Gamma, \Sigma \vdash \Delta, \Pi, A[T/s]}\ (=\text{:r}_2)$$

Induction

Let A be a formula in which x occurs freely and t a term.

$$\frac{\Gamma \vdash \Delta, A[x/0] \qquad \Sigma, A \vdash \Pi, A[x/s(x)]}{\Gamma, \Sigma \vdash \Delta, \Pi, A[x/t]} \text{(ind)}$$

B The Resolution Calculus

C Formula Syntax

C.1 Simple FOL Parsing (parseFOL)

The simple FOL parser interpretes names strings built over the alphabet $[\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, 0, \ldots 9, -, \_, \{, \}]$. Also the equality symbol $=$ is a name.

Names starting with lower case letters from a to t are interpeted as constant symbols. Likewise, names starting with lower case letters from u to z are interpreted as variables. Function symbols start with a lower case letter and predicate symbols start with an upper case letter.

Arguments are always seperated by commas and put in parenthesis. Terms are built from variable, constant symbols and function symbols applied to a number of arguments, which are terms. Likewise, atoms are predicate symbols, possibly with arguments which are again terms.

For example, the following are terms:

```
   "constant", "variable", "function(constant,variable)", "s_{21}"
```

And the following are atoms:

```
   "P_1(alpha, beta(x))", "=(25,v)"
```

The logical connectives are called $\{\mathtt{And}, \mathtt{Or}, \mathtt{Neg}, \mathtt{Imp}, \mathtt{Forall}, \mathtt{Exists}\}$ and are written in prefix form.

For example, the formula $\forall x(P(x, f(x)) \to \exists y P(x, y))$ is parsed as:

```
gapt> parseFOL("Forall x Imp P(x,f(x)) Exists y P(x,y)")
res0: at.logic.gapt.expr.FOLFormula = ∀x.(P(x,f(x))⊃∃y.P(x,y))
```

C.2 Prover 9 parser (parseProver9)

The Prover9 parser interpretes names strings built over the alphabet $[\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, 0, \ldots 9]$. Also the equality symbol $=$ and the arithmetic operators $\{+, -, *, /\}$ are names.

Since prover 9 has two naming schemes, we describe both of them. The default is the native LADR scheme.

C.2.1 LADR naming scheme (default)

Names starting with lower case letters from `u` to `z` are interpeted as variable symbols. Every name starting with a different letter is interpeted as a constant/function/predicate symbol, depending on the position in the term.

C.2.2 Prolog naming scheme

Names starting with upper case letters from `A` to `Z` are interpeted as variable symbols. Every name starting with a different letter is interpeted as a constant/function, depending on the position in the term. Predicates start with a lower or an upper case letter.

C.2.3 Terms and Formulas

Arguments are always seperated by commas and put in parenthesis. Terms are built from variable, constant symbols and function symbols applied to a number of arguments, which are terms. Likewise, atoms are predicate symbols, possibly with arguments which are again terms. The function symbols $\{+, -, *, /\}$ and the equality predicate $=$ are written in infix notation.

For example, the following are terms in the LADR naming scheme:

```
   "constant", "variable", "function(constant,variable)", "s_{21} + 5"
```

And the following are atoms:

```
   "P1(alpha, beta(x))", "25 = v"
```

The logical connectives are called $\{\&, |, - >, \texttt{all}, \texttt{exists}\}$ where the quantifiers and negation are written in prefix form and conjunction, disjunction and implication are written in infix form.

Conjunction and disjunction are right-associative. Quantifiers always need to be enclosed in parenthesis but a series of quantfiers doesn't need parenthesis in between. The binding order is (from strongest to weakest): $\neg, \forall, \exists, \wedge\vee, \rightarrow$.

For example, the formula $\forall x(P(x, f(x)) \rightarrow \exists y P(x, y))$ is parsed as:

```
gapt> parseProver9("(all x (P(x,f(x)) -> (exists y P(x,y))))")
res0: at.logic.gapt.expr.FOLFormula = ∀x.(P(x,f(x))⊃∃y.P(x,y))
```

The Prolog scheme is activated by adding an additional false to the parser:

```
gapt> parseProver9("(all X (P(X,f(X)) -> (exists Y P(X,Y))))", false)
res0: at.logic.gapt.expr.FOLFormula = ∀X.(P(X,f(X))⊃∃Y.P(X,Y))
```

References

[1] Matthias Baaz and Alexander Leitsch. Cut-elimination and Redundancy-elimination by Resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.

[2] 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 - 7th International Joint Conference, IJCAR*, volume 8562 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2014.

[3] Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. Algorithmic introduction of quantified cuts. *Theoretical Computer Science*, 549:1–16, 2014.

[4] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. Towards algorithmic cut-introduction. In *Proceedings of the 18th international conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'12, pages 228–242, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] 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, Proceedings*, volume 8123 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2013.

[6] William McCune. Prover9 and mace4 manual - input files, 2005–2010. `https://www.cs.unm.edu/~mccune/mace4/manual/2009-11A/input.html`.

[7] Dale Miller. A Compact Representation of Proofs. *Studia Logica*, 46(4):347–370, 1987.

[8] Gaisi Takeuti. *Proof Theory*. North-Holland, Amsterdam, 2nd edition, March 1987.