

---

# GAPT

## Generic Architecture for Proof Transformations

# User Manual

April 16, 2014

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

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>System Requirements</b>	<b>3</b>
<b>3</b>	<b>Downloading and Running the System</b>	<b>3</b>
<b>4</b>	<b>File Input/Output</b>	<b>4</b>
<b>5</b>	<b>Entering Formulas</b>	<b>5</b>
<b>6</b>	<b>Automated Deduction</b>	<b>6</b>
6.1	SAT Solving using MiniSAT . . . . .	6
6.2	SMT Solving using VeriT . . . . .	7
6.3	First-Order Resolution Proving using Prover9 . . . . .	7
6.4	Built-In Resolution Prover . . . . .	7
6.5	Built-In Tableaux Prover . . . . .	7
<b>7</b>	<b>Entering Proofs</b>	<b>7</b>
<b>8</b>	<b>Proof Theory</b>	<b>9</b>
8.1	Skolemization . . . . .	9
8.2	Interpolation . . . . .	9
8.3	Expansion Trees . . . . .	9
<b>9</b>	<b>Cut-Elimination by Resolution</b>	<b>10</b>
9.1	First-Order Logic . . . . .	10
9.2	Higher-Order Logic . . . . .	11
9.3	Schematic First-Order Logic . . . . .	11
<b>10</b>	<b>Cut-Introduction</b>	<b>11</b>
<b>11</b>	<b>Miscellaneous</b>	<b>13</b>
<b>A</b>	<b>The Sequent Calculus</b>	<b>14</b>
A.1	First-order LK . . . . .	14
<b>B</b>	<b>The Resolution Calculus</b>	<b>16</b>



## 1 Introduction

GAPT is a generic architecture for proof transformations. It is implemented in Scala and its logical basis is simple type theory. It has layers for first-order logic and schematic first-order logic.

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-introduction [6], 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 <http://code.google.com/p/gapt/w/list>.

Meta-comments about this user guide are printed like this.

## 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://code.google.com/p/gapt/downloads/list>. 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:

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

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

```
scala> 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`,  $\dots$ , `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

**TODO: first some general talk on file formats, then: give examples for saving and loading proofs.**

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:

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

returns a list of length 1 as shown by entering

```
scala> proofs.length
```

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

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

and the proof itself by

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

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

```
scala> 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: List[Proof], List[String], String → Unit` 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.

```
scala> 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:

```
scala> val F = parse.fol( "Imp A Imp B And A B" )  
F: at.logic.language.fol.FOLFormula = (A⊃(B⊃(A∧B)))
```

For a first-order examples consider:

```
scala> val G = parse.fol( "Forall x Imp P(x,f(x)) Exists y P(x,y)" )  
G: at.logic.language.fol.FOLFormula = ∀x.(P(x, f(x))⊃ ∃y.P(x, y))
```

Also `prover9` syntax[7] is supported:

```
scala> val PG = parse.p9 "(all x (P(x,f(x)) -> (exists y P(x,y))))"  
PG: at.logic.language.fol.FOLFormula = ∀x.(P(x, f(x))⊃ ∃y.P(x, y))
```

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

```
scala> val HG = parse.hlkformula "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)))"  
HG: at.logic.language.hol.HOLFormula = ∀x:o.(P(x:o, f(x:o):ι): o⊃ ∃y:ι.P(x:o, y:  
  ι): o)
```

TODO: add the other parsers, describe syntax in detail

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

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

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

```
scala> 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 validity and satisfiability, and query the thus obtained models. Consider the *pigeon hole principle for  $(m, n)$* ,  $\text{PHP}_{m,n}$ , which states that if  $m$  pigeons are put into  $n$  holes, then there is a hole which contains two pigeons. It is valid iff  $m > n$ .  $\neg\text{PHP}_{m,n}$  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

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

if you have not done so already. Then

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

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

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

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

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

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

```
scala> val p1 = at.logic.testing.PigeonHolePrinciple.atom(1, 1)
p1: at.logic.language.fol.FOLFormula = R(p_1, h_1)

scala> val p2 = at.logic.testing.PigeonHolePrinciple.atom(2, 1)
p2: at.logic.language.fol.FOLFormula = R(p_2, h_1)

scala> val p3 = at.logic.testing.PigeonHolePrinciple.atom(3, 1)
p3: at.logic.language.fol.FOLFormula = R(p_3, h_1)

scala> m.interpret(p1) // Is pigeon 1 in hole 1?
```

```
res14: Boolean = false

scala> m.interpret(p2) // Is pigeon 2 in hole 1?
res15: Boolean = true

scala> m.interpret(p3) // Is pigeon 3 in hole 1?
res16: Boolean = false
```

We can also interpret quantifier-free formulas:

```
cala> m.interpret( And(p1, p2) )
res17: Boolean = false

scala> m.interpret( Neg(notvalid) )
res18: Boolean = true
```

## 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 resolution 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:

```
scala> 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.

```
scala> val F1 = parse.fol( "Imp B And A B" )
F1: at.logic.language.fol.FOLFormula = (B ⊃ (A ∧ B))

scala> val F2 = parse.fol( "And A B" )
F2: at.logic.language.fol.FOLFormula = (A ∧ B)
```



```
scala> val A = parse.fol( "A" )
A: at.logic.language.fol.FOLFormula = A

scala> val B = parse.fol( "B" )
B: at.logic.language.fol.FOLFormula = B
```

We start with the axioms:

```
scala> val p1 = Axiom( A::Nil, A::Nil )
p1: at.logic.utils.ds.trees.LeafTree[at.logic.calculi.lk.base.Sequent] with at.
    logic.calculi.lk.base.NullaryLKProof{def rule: at.logic.calculi.lk.
        propositionalRules.InitialRuleType.type} = InitialRuleType(A :- A)

scala> val p2 = Axiom( B::Nil, B::Nil )
p2: at.logic.utils.ds.trees.LeafTree[at.logic.calculi.lk.base.Sequent] with at.
    logic.calculi.lk.base.NullaryLKProof{def rule: at.logic.calculi.lk.
        propositionalRules.InitialRuleType.type} = InitialRuleType(B :- B)
```

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

```
scala> val p3 = AndRightRule( p1, p2, A, B )
p3: at.logic.calculi.lk.base.BinaryLKProof with at.logic.calculi.lk.base.
    BinaryLKProof with at.logic.calculi.lk.base.AuxiliaryFormulas with at.logic
    .calculi.lk.base.PrincipalFormulas = AndRightRuleType(A, B :- (A^B),
    InitialRuleType(A :- A), InitialRuleType(B :- B))
```

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

```
scala> val p4 = ImpRightRule( p3, B, F2 )
p4: at.logic.utils.ds.trees.UnaryTree[at.logic.calculi.lk.base.Sequent] with at
    .logic.calculi.lk.base.UnaryLKProof with at.logic.calculi.lk.base.
    AuxiliaryFormulas with at.logic.calculi.lk.base.PrincipalFormulas =
    ImpRightRuleType(A :- (B^((A^B))), AndRightRuleType(A, B :- (A^B),
    InitialRuleType(A :- A), InitialRuleType(B :- B)))

scala> val p5 = ImpRightRule( p4, A, F1 )
p5: at.logic.utils.ds.trees.UnaryTree[at.logic.calculi.lk.base.Sequent] with at
    .logic.calculi.lk.base.UnaryLKProof with at.logic.calculi.lk.base.
    AuxiliaryFormulas with at.logic.calculi.lk.base.PrincipalFormulas =
    ImpRightRuleType( :- (A^((B^((A^B))))), ImpRightRuleType(A :- (B^((A^B))),
    AndRightRuleType(A, B :- (A^B), InitialRuleType(A :- A), InitialRuleType(B
    :- B))))
```

You can now view this proof by typing:

```
scala> 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:

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

mention hlk here!?

## 8 Proof Theory

### 8.1 Skolemization

short example for proof skolemization

### 8.2 Interpolation

The command `extractInterpolant` extracts an interpolant from a cut-free sequent calculus proof. 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, then you can compute an interpolant as follows:

```
scala> val s = pr.root
s: at.logic.calculi.lk.base.Sequent = (p:o ∨ q:o) :- p:o, q:o

scala> val npart = Set( s.antecedent( 0 ), s.succedent( 0 ) )
npart: scala.collection.immutable.Set[at.logic.calculi.occurrences.
  FormulaOccurrence] = Set((p:o ∨ q:o)[10005], p:o[10006])

scala> val ppart = Set( s.succedent( 1 ) )
ppart: scala.collection.immutable.Set[at.logic.calculi.occurrences.
  FormulaOccurrence] = Set(q:o[10007])

scala> val I = extractInterpolant( pr, npart, ppart )
I: at.logic.language.hol.HOLFormula = (⊥:o ∨ q:o)
```

### 8.3 Expansion Trees

how to extract expansion trees, how to view them in prooftool, remove `extractHerbrandSequent` in the long run

**extractHerbrandSequent: LKProof → Sequent** A proof in first order logic can be concisely represented with a Herbrand sequent. Since this sequent is purely propositional, this is a way of reducing first order logic to propositional logic. In short, if  $\varphi$  is a proof and  $H$  is its Herbrand sequent, then  $H$  contains all instances of the formulas in the end-sequent of  $\varphi$  and  $H$  is a tautology. It is important to note that  $\varphi$  may not have strong quantifiers nor cuts on quantified formulas. A more detailed explanation of Herbrand sequent extraction can be found in ??.

This method extracts a Herbrand sequent from a proof  $\varphi$  that satisfies the constraints. An execution and output of this command is shown below. The proof  $p$  used is the “Linear Example Proof” for

$n = 4$ , which is a proof without cuts of the end sequent:

$$P0, \forall x.(Px \rightarrow Psx) \vdash Ps^4 0$$

```
scala> var hs = extractHerbrandSequent(p)
hs: at.logic.calculi.lk.base.types.FSequent =
([ (P(s(s(s(0)))) -> P(s(s(s(s(0))))), (P(s(s(0))) -> P(s(s(s(0))))),
(P(s(0)) -> P(s(s(0)))), (P(0) -> P(s(0))), P(0)], [P(s(s(s(s(0))))])
```

Observe that this sequent (represented by two lists, one for the antecedent and one for the succedent) is a tautology, is propositional and contains all instances of  $\forall x.(Px \rightarrow Psx)$  that must be used in a cut-free proof.

## 9 Cut-Elimination by Resolution

### 9.1 First-Order Logic

The *ceres*-functionality should be demonstrated by an example session using `extractStruct`, `structToClausesList`, `prover9` etc.

**extractStruct: LKProof  $\rightarrow$  Struct** Extracts a struct from a LKProof. A struct is referred to as a clause term in [2]. I will give a quick definition of it. For more details and for an explanation of this structure on the CERES method, please refer to [2], Chapter 6.

Given a proof with cuts, by removing all the inference rules that operate on end sequent ancestors, we obtain a proof of the empty sequent (refutation). The axioms of this refutation are represented by *clause terms*. Clause terms are  $\{\oplus, \otimes\}$ -terms over clause sets, and its interpretation is the following:

$$\begin{aligned} |\mathcal{C}| &= \mathcal{C} \text{ if } \mathcal{C} \text{ is a set of clauses.} \\ |X \oplus Y| &= |X| \cup |Y| \\ |X \otimes Y| &= |X| \times |Y| \end{aligned}$$

where  $\mathcal{C} \times \mathcal{D} = \{C \circ D \mid C \in \mathcal{C} \wedge D \in \mathcal{D}\}$  and if  $S = \Gamma \vdash \Delta$  and  $S' = \Pi \vdash \Lambda$ ,  $S \circ S' = \Gamma, \Pi \vdash \Delta, \Lambda$ .

**structToClausesList: Struct  $\rightarrow$  List[Sequent]** computes the standard characteristic clause set from the struct, see [9, Section 4.2.1] for details.

**structToLabelledClausesList: Struct  $\rightarrow$  List[LabelledSequent]** TODO

## 9.2 Higher-Order Logic

## 9.3 Schematic First-Order Logic

# 10 Cut-Introduction

The cut-introduction algorithm as described in [4] is implemented in GAPT for introducing a single  $\Pi_1$ -cut into a sequent calculus proof. In this section we show the commands, step by step, that need to be executed for this algorithm. We will use as input one of the proofs generated by the system, namely, `LinearExampleProof(9)`. But the user can also, for example, write his own proofs in `hlk`<sup>1</sup> and input these files to the system.

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

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

First of all, we instantiate the desired proof and store this in a variable:

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

You will see that a big string representing the proof is printed. If desired, you can view this proof using `prooftool`. It is possible to see some information about a proof on the command line by calling:

```
scala> printProofStats(p)
----- Statistics -----
Cuts: 0
Number of quantifier rules: 9
Number of rules: 28
Quantifier complexity: 9
-----
```

Now we need to extract the terms used to instantiate the  $\forall$  quantifiers of the end-sequent:

```
scala> val ts = extractTerms(p)
The term set contains 9 terms.
```

The system indicates how many terms were extracted, which is nine for this case, as expected. The next step consists in computing grammars that generate this term set:

```
scala> val grms = computeGrammars(ts)
693 grammars found.
```

The number of grammars found is shown, 693 in this case. They are ordered by size, and one can see the first ones by calling:

```
scala> seeNFirstGrammars(grms, 5)
```

---

<sup>1</sup><http://www.logic.at/hlk/>

```

0. \{ tuple1(s(s(s(\alpha))))), tuple1(\alpha), tuple1(s(s(s(s(s(\alpha)))))) \} o \{ 0, s(0), s(s(0)) \}
(size = 6)
1. \{ tuple1(s(\alpha)), tuple1(s(s(\alpha))), tuple1(\alpha) \} o \{ 0, s(s(0)), s(s(s(0))) \}
(size = 6)
2. \{ tuple1(s(0)), tuple1(\alpha), tuple1(s(s(\alpha))), tuple1(s(s(\alpha)))) \} o \{ 0, s(s(s(0))),
s(s(s(s(0)))) \}(size = 7)
3. \{ tuple1(s(s(0))), tuple1(\alpha), tuple1(s(s(\alpha))))), tuple1(
s(\alpha)) \} o \{ 0, s(s(s(0))),
s(s(s(s(0)))) \}(size = 7)
4. \{ tuple1(\alpha), tuple1(s(s(\alpha))), tuple1(s(s(s(\alpha))))),
tuple1(s(\alpha)) \} o \{ 0, s(s(s(0))),
s(s(s(s(0)))) \}(size = 7)

```

Note that the function symbols 'tuple1' are inserted by the system as part of the algorithm.

This will print on the screen the first 5 grammars, and we can choose which one to use for compressing the proof, in this case we take the second one:

```
scala> val g = grms(1)
```

Given the end-sequent of the proof and a grammar, the extended Herbrand sequent can be computed:

```

scala> val ehs = generateExtendedHerbrandSequent(p.root, g)
ehs: at.logic.algorithms.cutIntroduction.ExtendedHerbrandSequent =
P(0), (P(s(\alpha))\(\impl\)P(s(s(\alpha))))), (P(s(s(\alpha))\(\impl\)P(s(s(s(\alpha))))), (P(\alpha)\(\impl\)P(s(\alpha))),
(X(\alpha)\(\impl\)X(s(s(s(s(s(0))))))\(\wedge\)X(s(s(s(0))))\(\wedge\)X(0))) :-
P(s(s(s(s(s(s(s(0)))))))),

```

As it was shown in [4], the cut-introduction problem has a canonical solution:

```

scala> val cs = computeCanonicalSolution(p.root, g)
Note that the clauses that do not contain the eigenvariable were already removed.
cs: at.logic.language.fol.FOLFormula =
\(\forall x. ((P(x)\(\impl\)P(s(x))\(\wedge\)((P(s(s(x))\(\impl\)P(s(s(x)))\(\wedge\)P(s(s(s(x))))\(\wedge\)P(s(x))\(\impl\)P(s(s(x))))))\(\wedge\)P(s(x))\(\impl\)P(s(s(x))))))\(\wedge\)P(s(x))\(\impl\)P(s(s(x))))))

```

The extended Herbrand sequent generated previously has the canonical solution as default, but this solution can be improved..

```

scala> minimizeSolution(ehs)
Previous solution: \(\forall x. ((P(x)\(\impl\)P(s(x))\(\wedge\)((P(s(s(x))\(\impl\)P(s(s(x)))\(\wedge\)P(s(s(s(x))))\(\wedge\)P(s(x))\(\impl\)P(s(s(x))))))\(\wedge\)P(s(x))\(\impl\)P(s(s(x))))))\(\wedge\)P(s(x))\(\impl\)P(s(s(x))))))
Improved solution: \(\forall x. (P(s(s(s(x))))\(\vee\)neg P(x))

```

Finally, the proof with cut is constructed:

```
scala> val fp = buildProofWithCut(ehs)
```

In order to compare this with the initial proof, one can again count the number of rules:

```
scala> printProofStats(fp)
----- Statistics -----
Cuts: 1
Number of quantifier rules: 7
Number of rules: 25
Quantifier complexity: 6
-----
```

We showed how to run the cut-introduction algorithm step by step. There is, though, a command comprising all these steps:

```
scala> val fp2 = cutIntro(p)
```

Regarding the choice of the grammar, this command will compute the proofs with all minimal grammars and choose the smallest one (with respect to the number of rules).

## 11 Miscellaneous

The method `printProofStats: LKProof → Unit` takes a proof and prints some statistics about it, namely the number of unary, binary and cut rules. If you have a variable storing the result of `loadProofs(...)`, one way to get the first proof of the list is: `proof.head._2`. If `proof` is the one defined on the previous example:

```
scala> printPoofStats(proof.head._2)
unary: 2
binary: 1
cuts: 0
```

Of course you can always assing this to a variable and use it as a parameter:

```
scala> var theproof = proof.head._2
theproof: at.logic.calculi.lk.base.LKProof = ExistsRightRuleType( :-  $\exists(x.(\forall(y.(((a \text{ P } x) \wedge (a \text{ Q } y))))))$ ), ForallRightRuleType( :-  $\forall(y.(((a \text{ P } b) \wedge (a \text{ Q } y)))$ ), AndRightRuleType( :-  $((a \text{ P } b) \wedge (a \text{ Q } \backslash\text{beta}))$ ), InitialRuleType( :-  $(a \text{ P } b)$ ), InitialRuleType( :-  $(a \text{ Q } \backslash\text{beta}))$ ))

scala> printPoofStats(theproof)
unary: 2
binary: 1
cuts: 0
```

## 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

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

#### 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**

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} (\wedge l_1)$$

$$\frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} (\wedge l_2)$$

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

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

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

**Right rules**

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

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B} (\vee r_1)$$

$$\frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B} (\vee r_2)$$

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

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

**Quantification rules****Left rules**

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

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

**Right rules**

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

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

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

**Equational rules****Left rules**

$$\frac{\Gamma \vdash \Delta, s = t \quad \Sigma, A[T/s] \vdash \Pi}{\Gamma, \Sigma, A[T/t] \vdash \Delta, \Pi} (=l_1)$$

$$\frac{\Gamma \vdash \Delta, s = t \quad \Sigma, A[T/t] \vdash \Pi}{\Gamma, \Sigma, A[T/s] \vdash \Delta, \Pi} (=l_2)$$

**Right rules**

$$\frac{\Gamma \vdash \Delta, s = t \quad \Sigma \vdash \Pi, A[T/s]}{\Gamma, \Sigma \vdash \Delta, \Pi, A[T/t]} (=r_1)$$

$$\frac{\Gamma \vdash \Delta, s = t \quad \Sigma \vdash \Pi, A[T/t]}{\Gamma, \Sigma \vdash \Delta, \Pi, A[T/s]} (=r_2)$$



## B The Resolution Calculus

TODO: give formal definition of our resolution calculus

## C Command Reference

Shall we really keep a command reference in the user manual? This may make more sense as `help <command>` in the CLI.

**prover9:** `List[Sequent], Seq[Clause] → Option[ResolutionProof[Clause]]` sends the input clause set (given as either `List[Sequent]` or `Seq[Clause]`) to prover9. Returns the resolution proof obtained from replaying the output proof of prover9, see [3] for details.

**miniSATsolve:** `HOLFormula → Option[Interpretation]` Searches a model for a quantifier-free formula using the MiniSAT SAT Solver. Returns `None` if unsatisfiable, and `Some(Interpretation)` otherwise.

**miniSATprove:** `HOLFormula → Boolean` Checks if a quantifier-free formula is valid using the MiniSAT SAT Solver.

**lkTolksk:** `LKProof → LKProof` This method takes a proof in classical logic (LK), such as one generated by HLK and loaded by the method `loadProofs`, and transforms it to a proof on the calculus  $LK_{sk}$ . This calculus was proposed to solve the problem of Skolemization in higher-order logic, and it basically replaces eigenvariables with Skolem terms. For more information, see [5].

**cutIntro:** `LKProof → LKProof` This method is the implementation of the cut-introduction algorithm described on [6]. It takes a cut-free proof in classical logic, automatically computes a universally quantified cut formula and builds a new proof with this cut.

**termsExtraction:** `LKProof → Map[FormulaOccurrence, List[List[FOLTerm]]]` A crucial part of the cut-introduction algorithm of [6] is the computation of the term set, which are the witnesses of the existential quantifiers of the end-sequent of a proof. This method takes a proof and returns a map. This map associates each existentially quantified formula of the end sequent with a list of tuples of terms. These tuples will have the same size as the number of quantifiers of the formula.

**regularize:** `LKProof → LKProof` TODO

**createHOLExpression:** `String → HOLExpression (Forall x1: (i -λ (i -λ i)) a(x1: (i -λ (i -λ i)), x2: i, c1: (i -λ i)))` TODO

**fsequent2sequent: FSequent  $\rightarrow$  Sequent**    **TODO**

**deleteTautologies: List[FSequent]  $\rightarrow$  List[FSequent]**    **TODO**

**removeDuplicates: List[FSequent]  $\rightarrow$  List[FSequent]**    **TODO**

**unitResolve: List[FSequent]  $\rightarrow$  List[FSequent]**    **TODO**

**removeSubsumed: List[FSequent]  $\rightarrow$  List[FSequent]**    **TODO**

**normalizeClauses: List[FSequent]  $\rightarrow$  List[FSequent]**    **TODO**

**writeLatex: List[FSequent], String  $\rightarrow$  Unit**    **TODO**

**writeLabelledSequentListLatex: List[LabelledSequent], String  $\rightarrow$  Unit**    **TODO**

## References

- [1] Matthias Baaz and Alexander Leitsch. Cut-elimination and Redundancy-elimination by Resolution. *Journal of Symbolic Computation*, 29(2):149–176, 2000.
- [2] Matthias Baaz and Alexander Leitsch. *Methods of Cut-Elimination*, volume 34 of *Trends in Logic*. Springer, 2011.
- [3] 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 *2nd International Workshop on Proof Exchange for Theorem Proving (PxTP)*, 2012.
- [4] Stefan Hetzl, Alexander Leitsch, Giselle Reis, and Daniel Weller. Algorithmic introduction of quantified cuts. submitted to Theoretical Computer Science, 2013.
- [5] Stefan Hetzl, Alexander Leitsch, and Daniel Weller. CERES in Higher-Order Logic. *to appear in the Annals of Pure and Applied Logic*, 2011.
- [6] 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.
- [7] William McCune. Prover9 and mace4 manual - input files, 2005–2010. <https://www.cs.unm.edu/~mccune/mace4/manual/2009-11A/input.html>.
- [8] Gaisi Takeuti. *Proof Theory*. North-Holland, Amsterdam, 2nd edition, March 1987.
- [9] Bruno Woltzenlogel Paleo. *A General Analysis of Cut-Elimination by CERes*. PhD thesis, Vienna University of Technology, 2009.