GAPT

General Architecture for Proof Theory

User manual

Version 2.8-SNAPSHOT

September 28, 2017

List of Corrections

Contents

1 Introduction

GAPT is a generic architecture for proof transformations implemented in Scala.

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 that 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 that allows the user to view (and—to a certain extent— 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 [2], cut-introduction [8], etc.

2 Download and execution

There are three ways you can obtain GAPT:

1. **The recommended way:** You can download a package of the current version of GAPT at https://logic.at/gapt/. After extracting the tar.gz-file, you will find a shell script gapt.sh.

   Running this script will start the command line interface of GAPT:

   ```
   ./gapt.sh
   ```

2. If you are adventurous, you can also download an unstable development version from github:

   ```
   git clone https://github.com/gapt/gapt
   cd gapt
   sbt console
   ```

3. If you like GAPT and want to use it as a library in your Scala project, it is available as a Maven artifact on JCenter. All you need to do is add two lines to your build.sbt:

   ```
   resolvers += Resolver.jcenterRepo
   libraryDependencies += "at.logic.gapt" %% "gapt" % "2.7"
   ```

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

You don't need to know anything about Scala to try out the examples in this manual, but if you do want to learn more about Scala we recommend the book "Programming in Scala" [12].

Interactions with the Scala shell are typeset in the following way:

```
gapt> println("Hello, world!")
Hello, world!
```

Here, **println("Hello, world!")** is the user input, and Hello, world! is the output from the Scala shell.

If you want to consult the in-depth API documentation of a function, you can use the help command:

```
gapt> help(containsQuantifierOnLogicalLevel)
```

2.1 System requirements

To run GAPT you need to have Java 8 (or higher) installed.

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 and prooftrans are available.
- E theorem prover (http://eprover.org/)
- Vampire 4.0 (http://www.vprover.org/)
- SPASS (http://www.spass-prover.org/)
- LeanCoP (http://leancop.de/)
- Metis (http://www.gilith.com/software/metis/)
- iProver (http://www.cs.man.ac.uk/~korovink/iprover/, requires a development version as of September 11, 2017)
- VeriT (http://www.verit-solver.org/)
- Z3 (https://github.com/Z3Prover/z3)
- MiniSAT (http://minisat.se/)
- Glucose (http://www.labri.fr/perso/lsimon/glucose/)
- PicoSAT (http://fmv.jku.at/picosat/)
- Sat4J (http://sat4j.org/)
- OpenWBO (http://sat.inesc-id.pt/open-wbo/)
- CVC4 (http://cvc4.cs.nyu.edu/web/)
- TIP tools (https://github.com/tip-org/tools)

3 Data structures

3.1 Expressions and formulas

Formulas, terms, and all other expressions are represented as terms in a polymorphic simply-typed lambda calculus. For example, the formula $\forall x\, P(x, y)$ is encoded as the term '∀' (λx (P x) y). This term has the type o, which is the type of Boolean values. The variable x in this term has the type i, which is the default type for first-order variables.

There are two ways of entering expressions: you can parse them or construct them manually.

3.1.1 Formula parsing

Here is an example of parsing a first-order formula:

```
gapt> val F = fof"!x (P(x,f(x)) -> ?y P(x,y))"
F: at.logic.gapt.expr.FOLFormula = ∀x (P(x, f(x)) ⊃ ∃y P(x, y))
```

Every kind of expression that GAPT supports can be parsed by writing <prefix>"<string>". The prefix indicates the Scala type of the expression. The following prefixes are available:

ty	type
le	lambda expression
hof	higher-order formula
hoa	higher-order atom
hov	higher-order variable
hoc	higher-order constant
foe	first-order expression
fof	first-order formula
fot	first-order term
foa	first-order atom
fov	first-order variable
foc	first-order constant

This parser supports Scala string interpolation. For example, you can do:

```
gapt> val t = fot"f(f(x))"
t: at.logic.gapt.expr.FOLTerm = f(f(x))

gapt> val G = fof"!x (P(x,$t) -> ?y P(x,y))"
G: at.logic.gapt.expr.FOLFormula = ∀x (P(x, f(f(x))) ⊃ ∃y P(x, y))
```

The input language has full type inference, and the formula prefixes make sure that the expression is of type o (Boolean). If no particular type is required, we default to ι:

```
gapt> hof"!x?y!z x(z) = y(y(z))"
res2: at.logic.gapt.expr.Formula = ∀x ∃y ∀z x(z) = y(y(z))
```

So far we have only used the ASCII-safe part of the syntax, however Unicode input is of course supported as well—you can paste any of the output right back in:

```
gapt> hof"∀x ∃y ∀z x(z) = y(y(z))"
res3: at.logic.gapt.expr.Formula = ∀x ∃y ∀z x(z) = y(y(z))
```

Here is a summary of the available syntax (there are usually multiple variants of each construct, these are separated by commas here):

x1, uvw	variables (need to start with u-z or U-Z, or be bound)
c, theorem	constants
f(x,c), f(x)(c), f x c	function application
\x f(x), λx f(x), ^x f(x)	lambda abstraction
!x p(x), !(x:i) p(x), ∀x p(x)	universal quantification
?x p(x), ?(x:i) p(x), ∃x p(x)	existential quantification
-p, ¬ p	negation
p & q, p ∧ q	conjunction
p | q, p ∨ q	disjunction
p -> q, p ⊃ q	implication
p <-> q	equivalence (this is the same as p ⊃ q ∧ q ⊃ p)
p = q, p = q = r	equality
p != q	disequality
p < q <= r > s >= t	various infix relations
a*b/c + d - e	infix operators
f: i>i>o	type annotation

3.1.2 Constructing formulas manually

Every kind of expression that exists in GAPT can be constructed manually. For instance, you can define variables and constants like this:

```
gapt> val x = FOLVar("x")
x: at.logic.gapt.expr.FOLVar = x

gapt> val P = Const("P", Ti -> To)
P: at.logic.gapt.expr.Const = P:i>o
```

Var and Const require you to supply types, whereas FOLVar and FOLConst automatically have type ι. Terms and atomic formulas are constructed similarly:

```
gapt> val x = FOLVar("x")
x: at.logic.gapt.expr.FOLVar = x

gapt> val fx = FOLFunction("f",x)
fx: at.logic.gapt.expr.FOLTerm = f(x)

gapt> val Pfx = FOLAtom("P", fx)
Pfx: at.logic.gapt.expr.FOLAtom = P(f(x)): o
```

On the formulas themselves, there are operators for the various Boolean connectives:

-A	$\neg A$
A & B	$A \wedge B$
A | B	$A \vee B$
A --> B	$A \supset B$
A <-> B	$A \leftrightarrow B$

```
gapt> val A = FOLAtom("A")
A: at.logic.gapt.expr.FOLAtom = A:o

gapt> val B = FOLAtom("B")
B: at.logic.gapt.expr.FOLAtom = B:o

gapt> val C = FOLAtom("C")
C: at.logic.gapt.expr.FOLAtom = C:o

gapt> (A & B) --> C
res4: at.logic.gapt.expr.FOLFormula = A ∧ B ⊃ C
```

3.1.3 Predefined formulas

A collection of formula sequences can be found in the file examples/FormulaSequences.scala. You can generate instances of these formula sequences by entering for example:

```
gapt> val f = BussTautology( 5 )
f: at.logic.gapt.proofs.HOLSequent =
((c_1 ∨ d_1) ∧ (c_2 ∨ d_2) ∧ (c_3 ∨ d_3) ∧ (c_4 ∨ d_4) ⊃ c_5) ∨
  ((c_1 ∨ d_1) ∧ (c_2 ∨ d_2) ∧ (c_3 ∨ d_3) ∧ (c_4 ∨ d_4) ⊃ d_5),
((c_1 ∨ d_1) ∧ (c_2 ∨ d_2) ∧ (c_3 ∨ d_3) ⊃ c_4) ∨
  ((c_1 ∨ d_1) ∧ (c_2 ∨ d_2) ∧ (c_3 ∨ d_3) ⊃ d_4),
((c_1 ∨ d_1) ∧ (c_2 ∨ d_2) ⊃ c_3) ∨ ((c_1 ∨ d_1) ∧ (c_2 ∨ d_2) ⊃ d_3),
(c_1 ∨ d_1 ⊃ c_2) ∨ (c_1 ∨ d_1 ⊃ d_2),
c_1 ∨ d_1
⊢
c_5,
d_5
```

3.2 Sequents

Sequents are an important data structure in GAPT. A sequent is a pair of lists:

$$A_1, ..., A_m \vdash B_1, ..., B_n$$

The list to the left of the sequent symbol $\vdash$ is called the antecedent, the one on the right the succedent. Usually, but not always, the elements of the sequences are going to be formulas.

In GAPT, you can create sequents by supplying an antecedent and a succedent:

```
gapt> val S1 = Sequent()
S1: at.logic.gapt.proofs.Sequent[Nothing] = ⊢

gapt> val S2 = Sequent(List(1,2), List(3,4))
S2: at.logic.gapt.proofs.Sequent[Int] = 1, 2 :- 3, 4

gapt> val S3 = Sequent(List(foa"A", foa"B"), List(foa"C", foa"D"))
S3: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.FOLAtom] = A, B ⊢ C, D
```

Sequents of formulas can also be parsed:

```
gapt> hos"P a, a = b :- P b"
res5: at.logic.gapt.proofs.HOLSequent = P(a), a = b ⊢ P(b)
```

The following prefixes are available (a clause is a sequent of atoms):

hos	higher-order (formula) sequent
hcl	higher-order clause
fos	first-order (formula) sequent
fcl	first-order clause

Sequents have append operations for both the antecedent and the succedent. In the antecedent, elements are appended to the left, in the succedent, to the right:

```
gapt> val S1 = fcl"B :- C"
S1: at.logic.gapt.proofs.FOLClause = B ⊢ C

gapt> val S2 = foa"A" +: S1
S2: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.FOLAtom] = A, B ⊢ C

gapt> val S3 = S2 :+ foa"D"
S3: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.FOLAtom] = A, B ⊢ C, D

gapt> foa"A" +: foa"B" +: Sequent() :+ foa"C" :+ foa"D"
res6: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.FOLAtom] = A, B ⊢ C, D
```

You can retrieve elements from a sequent either by accessing the antecedent or succedent directly ...

```
gapt> val S = fcl"A, B :- C, D"
S: at.logic.gapt.proofs.FOLClause = A, B ⊢ C, D

gapt> val b = S.antecedent(1)
b: at.logic.gapt.expr.FOLAtom = B:o

gapt> val c = S.succedent(0)
c: at.logic.gapt.expr.FOLAtom = C:o
```

... or by using the `SequentIndex` class:

```
gapt> val i = Ant(0)
i: at.logic.gapt.proofs.Ant = Ant(0)
```

```
gapt> val j = Suc(1)
j: at.logic.gapt.proofs.Suc = Suc(1)

gapt> val a = S(i)
a: at.logic.gapt.expr.FOLAtom = A:o

gapt> val d = S(j)
d: at.logic.gapt.expr.FOLAtom = D:o
```

3.3 Proofs

3.3.1 LK

GAPT contains an implementation of Gentzen's sequent calculus **LK**. The inference rules are defined in appendix B.1.

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. We discuss this possibility in this section. For entering bigger proofs, it is more convenient to use the "gaptic" tactics language which is discussed in section 4.1.

Note: Many correctness properties of **LK** proofs are purely syntactic and can be checked at construction time. For instance, it is not possible to construct a proof that violates the eigenvariable condition of strong quantifier rules. However, some rules require additional assumptions to be correct. For example, the induction rule is only correct under the assumption that the cases used in the rule correspond precisely to the inductive type's constructors. Assumptions of this kind are collected in a `Context`, see section 3.4. Since top-down proof construction does not take contexts into account, it can result in proofs violating these assumptions. You can ensure that a proof you have constructed conforms to a context `ctx` by using the check method on `ctx`.

We start with the axioms:

```
gapt> val p1 = LogicalAxiom(fof"A")
p1: at.logic.gapt.proofs.lk.LogicalAxiom =
[p1] A ⊢ A   (LogicalAxiom(A:o))

gapt> val p2 = LogicalAxiom(fof"B")
p2: at.logic.gapt.proofs.lk.LogicalAxiom =
[p1] B ⊢ B   (LogicalAxiom(B:o))
```

These are joined by an $\wedge:\mathrm{right}$-inference.

```
gapt> val p3 = AndRightRule( p1, fof"A", p2, fof"B" )
p3: at.logic.gapt.proofs.lk.AndRightRule =
[p3] A, B ⊢ A ∧ B  (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B ⊢ B   (LogicalAxiom(B:o))
[p1] A ⊢ A   (LogicalAxiom(A:o))
```

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

```
gapt> val p4 = ImpRightRule( p3, fof"B", fof"A & B" )
p4: at.logic.gapt.proofs.lk.ImpRightRule =
[p4] A ⊢ B ⊃ A ∧ B (ImpRightRule(p3, Ant(1), Suc(0)))
[p3] A, B ⊢ A ∧ B (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B ⊢ B  (LogicalAxiom(B:o))
[p1] A ⊢ A  (LogicalAxiom(A:o))

gapt> val p5 = ImpRightRule( p4, fof"A", fof"B -> A&B" )
p5: at.logic.gapt.proofs.lk.ImpRightRule =
[p5]  ⊢ A ⊃ B ⊃ A ∧ B (ImpRightRule(p4, Ant(0), Suc(0)))
[p4] A ⊢ B ⊃ A ∧ B (ImpRightRule(p3, Ant(1), Suc(0)))
[p3] A, B ⊢ A ∧ B (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B ⊢ B  (LogicalAxiom(B:o))
[p1] A ⊢ A  (LogicalAxiom(A:o))
```

You can now view this proof by typing:

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

There are also several macro rules that make proof construction more convenient. For instance:

```
gapt> val p1 = LogicalAxiom(fof"A")
p1: at.logic.gapt.proofs.lk.LogicalAxiom =
[p1] A ⊢ A  (LogicalAxiom(A:o))

gapt> val p2 = AndLeftMacroRule(p1, fof"A", fof"B")
p2: at.logic.gapt.proofs.lk.AndLeftRule =
[p3] A ∧ B ⊢ A  (AndLeftRule(p2, Ant(1), Ant(0)))
[p2] B, A ⊢ A  (WeakeningLeftRule(p1, B:o))
[p1] A ⊢ A  (LogicalAxiom(A:o))
```

Here, the $\wedge : l$ macro rule automatically adds B via weakening before performing the $\wedge : l$ inference.

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. You can generate instances of these proof sequences by entering, e.g.,

```
gapt> val p = SumExampleProof( 5 )
p: at.logic.gapt.proofs.lk.LKProof =
[p25] ∀x ∀y (P(s(x), y) ⊃ P(x, s(y))), P(s(s(s(s(s(0))))), 0) ⊢ P(0, s(s(s(s(s(0)))))) (
    ContractionLeftRule(p24, Ant(0), Ant(1)))
[p24] ∀x ∀y (P(s(x), y) ⊃ P(x, s(y))),
∀x ∀y (P(s(x), y) ⊃ P(x, s(y))),
P(s(s(s(s(s(0))))), 0)
⊢
P(0, s(s(s(s(s(0)))))) (ForallLeftRule(p23, Ant(0), ∀y (P(s(x), y) ⊃ P(x, s(y))), 0, x))
[p23] ∀y (P(s(0), y) ⊃ P(0, s(y))),
∀x ∀y (P(s(x), y) ⊃ P(x, s(y))),
P(s(s(s(s(s(0))))), 0)
⊢
P(0, s(s(s(s(s(0)))))) (ForallLeftRule(p22, Ant(0), P(s(0), y) ⊃ P(0, s(y)), s(s(s(s(0)))
    ), y))
```

```
[p22] P(s(0), s(s(s(s(0))))) ⊃ P(0, s(s(s(s(s(0)))))),
∀x ∀y (P(s(x), y) ⊃ P(x, s(y))),
P(s(s(s(s(s(0))))), 0)
⊢
P(0, s(s(s(s(s(0)))))) (ImpLeftRule(p20, Suc(0), p21, Ant(0)))
[p21] P(0, s(s(s(s(s(0)))))) ⊢ P(0, s(s(s(s(s(0)))))) (LogicalA...
```

3.3.2 ND

GAPT furthermore contains an implementation of Gentzen's natural deduction calculus **ND**. The inference rules are defined in appendix B.2. To use the natural deduction inference rules you need to qualify the rule names with "nd.".

We show that from $P \wedge Q \supset R$ and P follows $Q \supset R$. We start with the axioms:

```
gapt> val p1 = nd.LogicalAxiom( fof"P" )
p1: at.logic.gapt.proofs.nd.LogicalAxiom =
[p1] P ⊢ P   (LogicalAxiom(P:o))

gapt> val p2 = nd.LogicalAxiom( fof"Q" )
p2: at.logic.gapt.proofs.nd.LogicalAxiom =
[p1] Q ⊢ Q   (LogicalAxiom(Q:o))

gapt> val p3 = nd.LogicalAxiom( fof"P & Q -> R" )
p3: at.logic.gapt.proofs.nd.LogicalAxiom =
[p1] P ∧ Q ⊃ R ⊢ P ∧ Q ⊃ R (LogicalAxiom(P ∧ Q ⊃ R))
```

P and Q are joined by an $\wedge$-introduction inference.

```
gapt> val p4 = nd.AndIntroRule( p1, p2 )
p4: at.logic.gapt.proofs.nd.AndIntroRule =
[p3] P, Q ⊢ P ∧ Q  (AndIntroRule(p1, p2))
[p2] Q ⊢ Q   (LogicalAxiom(Q:o))
[p1] P ⊢ P   (LogicalAxiom(P:o))
```

Next, we apply an $\supset$-elimination inference on $P \wedge Q \supset R$ and $P \wedge Q$ to arrive at R.

```
gapt> val p5 = nd.ImpElimRule( p3, p4 )
p5: at.logic.gapt.proofs.nd.ImpElimRule =
[p5] P ∧ Q ⊃ R, P, Q ⊢ R (ImpElimRule(p1, p4))
[p4] P, Q ⊢ P ∧ Q  (AndIntroRule(p2, p3))
[p3] Q ⊢ Q   (LogicalAxiom(Q:o))
[p2] P ⊢ P   (LogicalAxiom(P:o))
[p1] P ∧ Q ⊃ R ⊢ P ∧ Q ⊃ R (LogicalAxiom(P ∧ Q ⊃ R))
```

Finally, by using an $\supset$-introduction inference on Q, we arrive at the desired sequent.

```
gapt> val p6 = nd.ImpIntroRule( p5, Ant( 2 ) )
p6: at.logic.gapt.proofs.nd.ImpIntroRule =
[p6] P ∧ Q ⊃ R, P ⊢ Q ⊃ R (ImpIntroRule(p5, Ant(2)))
[p5] P ∧ Q ⊃ R, P, Q ⊢ R (ImpElimRule(p1, p4))
```

```
[p4] P, Q ⊢ P ∧ Q  (AndIntroRule(p2, p3))
[p3] Q ⊢ Q   (LogicalAxiom(Q:o))
[p2] P ⊢ P   (LogicalAxiom(P:o))
[p1] P ∧ Q ⊃ R ⊢ P ∧ Q ⊃ R (LogicalAxiom(P ∧ Q ⊃ R))
```

You can now view this proof by typing:

```
gapt> prooftool( p6 )
```

GAPT provides several convenience constructors which simplify proof construction, which can be found in the API documentation.

3.4 Proof contexts

The `Context` class captures the notion of a logical signature and background theory.

A context may contain declarations of:

- sorts and inductive types
- constants with previously declared types
- definitions
- Skolem functions
- Proof links

Various data structures and algorithms in GAPT require the presence of an implicit value of type `Context` in order to work. For example, the expression parser uses type and constant declarations to decide how to parse identifiers. Another example is the `eliminateDefinitions`, proof transformation: you may manually pass it a list of definitions to eliminate from a proof, or have it automatically eliminate all definitions in the current context. Some gaptic tactics (see 4.1) also require a context.

The typical way to declare a context is by starting with a default value and adding elements to it. The `Context.default` object contains only the sort o (truth values) and the fundamental logical symbols:

```
object ContextExample {
  implicit val ctx = MutableContext.default()
  ctx += Context.InductiveType("Nat", hoc" 0: Nat", hoc" s: Nat > Nat") //Adding a type
      declaration
  ctx += hoc" '+' : Nat>Nat>Nat" //Adding a constant declaration
  ctx += "plus_zero" -> hos" :- ∀n (n + 0 = n)" //Adding a theory axiom
  ctx += "1" -> le" s 0" //Adding a definition
  ctx += hof" leq x y = (∃z x + z = y)" //Adding a definition as an equation
}
```

It is important that you declare the context as `implicit`, so that it can be found automatically by the functions requiring it.

Once you have constructed a context `ctx`, you can check whether an expression, formula, sequent, or proof conforms to it by using the check method.

4 Feature walkthrough

4.1 Gaptic

GAPT contains a tactics language called `gaptic`. In contrast to the top-down construction presented in section 3.3, gaptic allows a comfortable bottom-up development of proofs, similar to popular proof assistants such as Coq, Isabelle, etc.

Gaptic can not be (easily) used in the interactive Scala shell, as it requires multi-line input. Gaptic scripts are usually developed as external files:

```
import at.logic.gapt.expr._
import at.logic.gapt.proofs.{Context, Sequent}
import at.logic.gapt.proofs.gaptic._

object example extends TacticsProof {
  ctx += Context.Sort("i")
  ctx += hoc"P: i>o"
  ctx += hoc"Q: i>o"

  val lemma = Lemma(
    ("a" -> fof"P a") +:
    ("b" -> fof"∀x (P x ⊃ Q x)") +:
    Sequent()
    :+ ("c" -> fof"Q a")
  ) {
    chain("b")
    chain("a")
  }
}
```

Gaptic proofs start with a context declaration. For more information on contexts, see section 3.4.

Note: Unlike top-down proof construction, proofs constructed with Gaptic are automatically correct with respect to the current context.

Each proof is then assigned to a Scala variable. The function `Lemma(labelledSequent) { tactics... }` constructs a proof using the gaptic language. The first argument of `Lemma` is the labelled end sequent, i.e. the sequent you want to prove in which each formula has a string label. The second argument consists of a list of statements, called tactics, separated by line breaks.

At the moment, there are two ways to execute gaptic scripts:

1. From the Scala shell, using the :load command. This command evaluates the Scala file, but *not* the code inside the object declaration. So we have to explicitly evaluate the proof ourselves.

```
gapt> :load example.scala
gapt> example.lemma
```

2. As a separate SBT project, see https://github.com/gapt/gaptic-example for a template project. This approach has the advantage that SBT can automatically run your script whenever you save it:

```
> ~runMain example
[info] Running example
[success] Total time: 1 s, completed Apr 5, 2016 11:16:32 AM
1. Waiting for source changes... (press enter to interrupt)
```

Let us use gaptic to input a very simple proof. Our first try might be the following (we now omit the boilerplate for brevity):

```
val lemmaEx =
  Lemma(Sequent(
      Seq("a" -> fof"P a", "b" -> fof"!x (P x -> Q x)"),
      Seq("c" -> fof"Q a"))) {
    allL(fot"a")
  }
```

```
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are still 1
    open sub goals:
b_0: P(a) ⊃ Q(a)
a: P(a)
b: ∀x (P(x) ⊃ Q(x))
:-
c: Q(a)

  at at.logic.gapt.proofs.gaptic.LemmaMacros$.finish(language.scala:45)
  at at.logic.gapt.proofs.gaptic.LemmaMacros$.finishLemma(language.scala:55)
  ... 25 elided
```

As seen above, the currently open goals are shown when the proof is not yet completed. Upon completion of the proof, the value of lemmaEx is the resulting proof:

```
val lemmaEx =
  Lemma(Sequent(
      Seq("a" -> fof"P a", "b" -> fof"!x (P x -> Q x)"),
      Seq("c" -> fof" Q a"))) {
    allL(fot"a")
    impL
    trivial
    trivial
  }
```

Most tactics can be called with or without a label argument. If a tactic is called with a label, it will be applied to that specific formula, if possible. Otherwise, the system will attempt to determine a target formula on its own. If there is either no applicable formula or more than one, the tactic will fail.

4.1.1 Basic tactics

We now give a description of a few basic tactics, you can find the full list in the API documentation:

```
gapt> help(at.logic.gapt.proofs.gaptic.TacticCommands)
```

The `forget` tactic corresponds to weakening rules in LK. It accepts a list of labels and removes the formulas with those labels from the current subgoal:

```
val lemmaEx =
  Lemma(Sequent(
      Seq("a" -> fof"P a", "b" -> fof"!x (P x -> Q x)"),
      Seq("c" -> fof" Q a"))) {
    forget("b")
  }
```

```
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are still 1
    open sub goals:
a: P(a)
:-
c: Q(a)

  at at.logic.gapt.proofs.gaptic.LemmaMacros$.finish(language.scala:45)
  at at.logic.gapt.proofs.gaptic.LemmaMacros$.finishLemma(language.scala:55)
  ... 25 elided
```

The tactics `axiomLog`, `axiomRefl`, `axiomBot` and `axiomTop` cover the logical, reflexivity, bottom and top axioms, respectively. The `trivial` tactic automatically selects the applicable axiom. Also, any weakening rules required to reach an actual axiom sequent are automatically applied.

The following example shows the use of the `trivial` tactic to end the proof by a logical axiom:

```
val axiomEx =
  Lemma(Sequent(Nil,
      Seq("D" -> fof"?x (P x -> !y P y)"))) {
    exR(fot"c")
    impR
    allR
    exR(fov"y")
    impR
    allR
    trivial
  }
```

The tactic eql covers the left and right equality rules. Its first argument is the label of an equality in the antecedent. The second argument is the label of the formula to apply the rule to. Furthermore, you may specify if the equality should be used from left to right or vice versa. Also, a target formula can be specified, if not all occurrences need to be replaced (in either direction). If neither direction nor a target formula is specified, the tactic will only work if the direction is unambiguous.

```
val eqEx = Lemma(Sequent(
    Seq("c" -> fof"P(y) & Q(y)",
        "eq1" -> fof"u = v",
        "eq2" -> fof"y = x",
        "a" -> fof"P(u) -> Q(u)"),
    Seq("b" -> fof"P(x) & Q(x)"))) {
  eql("eq1", "a").yielding(fof"P(v) -> Q(v)")
  eql("eq1", "a").yielding(fof"P(v) -> Q(u)")
  eql("eq2", "b").fromRightToLeft
  trivial
}
```

The tactics for the weak quantifiers are allL and exR. They are called with the list of terms to instantiate the quantified formula with. One call of allL or exR can instantiate any number of quantifiers in a formula. The tactics for the strong quantifiers are allR and exL. They are optionally called with the variable that should be used as an eigenvariable. If no eigenvariable is provided, a fresh variable will automatically be generated. The weak quantifier formulas are kept in the sequent after instantiations while the strong quantifier formulas are automatically removed.

```
val quantEx = Lemma(Sequent(
    Seq("D" -> fof"(!x (P(x) & (?y -P(y))))"),
    Nil)) {
  allL(fot"c")
  andL
  exL(fov"y_0")
  negL
  allL(fot"y_0")
  andL
  exL(fov"y_1")
  negL
  axiomLog
}
```

The implication, negation, disjunction and conjunction rules are covered by the tactics impL, impR, negL, negR, disL, disR, conL and conR, respectively. They are similar in the sense that they take no arguments apart from an optional label to apply the tactic to.

```
val propEx = Lemma(Sequent(
    Seq("initAnt" -> fof"A -> B"),
    Seq("initSuc" -> fof"(A & B) | -A"))) {
  orR("initSuc")
  negR("initSuc_1")
```

```
  andR("initSuc_0")
  trivial
  impL
  trivial
  trivial
}
```

The cut tactic is used to introduce a cut rule. The first argument is the (unique new) label for the cut formula, the second argument is the cut formula itself. Both arguments are mandatory. In the case where a non-unique label is provided the tactic simply fails.

```
val cutEx = Lemma(Sequent(
    Seq("A" -> fof"A"),
    Seq("C" -> fof"?x?y ( -x=y & f(x)=f(y) )"))) {
  cut("I1", fof"I(1)")
  cut("I0", fof"I(0)")
}
```

```
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are still 3
    open sub goals:
A: A
:-
C: ∃x ∃y (¬ x = y ∧ f(x) = f(y))
I1: I(1)
I0: I(0)

I0: I(0)
A: A
:-
C: ∃x ∃y (¬ x = y ∧ f(x) = f(y))
I1: I(1)

I1: I(1)
A: A
:-
C: ∃x ∃y (¬ x = y ∧ f(x) = f(y))

  at at.logic.gapt.proofs.gaptic.LemmaMacros$.finish(language.scala:45)
  at at.logic.gapt.proofs.gaptic.LemmaMacros$.finishLemma(language.scala:55)
  ... 25 elided
```

Using gaptic, we can also create proofs with induction. For example, let us prove that concatenation of lists is associative:

```
ctx += Context.Sort("i")

// Define the type of lists.
ctx += Context.InductiveType("list",
  hoc"nil: list",
```

```
  hoc"cons: i>list>list")

// Declare a constant denoting concatenation.
// We will axiomatize its definition in the end-sequent.
ctx += hoc"'+': list>list>list"

val catassoc =
  Lemma(
      ("conscat" -> hof"∀x ∀y ∀z cons(x,y)+z = cons(x,y+z)") +:
      ("nilcat" -> hof"∀x nil+x = x") +:
      Sequent()
      :+ ("goal" -> hof"∀x ∀y ∀z x+(y+z) = (x+y)+z")
    ) {
  decompose; induction(hov"x: list")
  rewrite.many ltr "nilcat"; refl
  rewrite.many ltr ("conscat", "IHx_0"); refl
}
```

4.2 SAT solver interface

The following shows an example session, using the Sat4j SAT solver to verify validity 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 $m \leq n$.

```
gapt> Sat4j isValid PigeonHolePrinciple(3, 2)
res10: Boolean = true
```

shows[1] that $\mathrm{PHP}_{3,2}$ is valid, and

```
gapt> Sat4j isValid PigeonHolePrinciple(3, 3)
res11: Boolean = false
```

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

```
gapt> val Some(m) = Sat4j solve -PigeonHolePrinciple(3, 3)
m: at.logic.gapt.models.PropositionalModel =
R(p_1, h_1): o -> true
R(p_1, h_2): o -> false
R(p_1, h_3): o -> false
R(p_2, h_1): o -> false
R(p_2, h_2): o -> true
R(p_2, h_3): o -> false
R(p_3, h_1): o -> false
R(p_3, h_2): o -> false
R(p_3, h_3): o -> true
```

[1] In Scala, `Sat4j isValid formula` is syntactic sugar for `Sat4j.isValid(formula)`.

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

```
gapt> val p1 = PigeonHolePrinciple.atom(1, 1)
p1: at.logic.gapt.expr.FOLAtom = R(p_1, h_1): o

gapt> val p2 = PigeonHolePrinciple.atom(2, 1)
p2: at.logic.gapt.expr.FOLAtom = R(p_2, h_1): o

gapt> m(p1) // Is pigeon 1 in hole 1?
res12: Boolean = true

gapt> m(p2) // Is pigeon 2 in hole 1?
res13: Boolean = false
```

We can also interpret quantifier-free formulas:

```
gapt> m(p1 & p2)
res14: Boolean = false
```

We can also convert $\neg\mathrm{PHP}_{3,3}$ into DIMACS format:

```
gapt> val cnf = structuralCNF(Sequent() :+ PigeonHolePrinciple(3,3)).map(_.conclusion.
    asInstanceOf[HOLClause])
cnf: scala.collection.immutable.Set[at.logic.gapt.proofs.HOLClause] = Set(R(p_3, h_1), R(
    p_2, h_1) ⊢ , ⊢ R(p_2, h_3), R(p_2, h_1), R(p_2, h_2), R(p_2, h_1), R(p_1, h_1) ⊢ , R
    (p_2, h_2), R(p_1, h_2) ⊢ , R(p_2, h_3), R(p_1, h_3) ⊢ , ⊢ R(p_1, h_3), R(p_1, h_1),
    R(p_1, h_2), R(p_3, h_2), R(p_2, h_2) ⊢ , R(p_3, h_3), R(p_1, h_3) ⊢ , R(p_3, h_1), R
    (p_1, h_1) ⊢ , ⊢ R(p_3, h_3), R(p_3, h_1), R(p_3, h_2), R(p_3, h_3), R(p_2, h_3) ⊢ ,
    R(p_3, h_2), R(p_1, h_2) ⊢ )

gapt> val encoding = new DIMACSEncoding
encoding: at.logic.gapt.formats.dimacs.DIMACSEncoding = DIMACSEncoding()

gapt> writeDIMACS(encoding encodeCNF cnf)
res15: String =
"p cnf 9 12
-1 -2 0
3 2 4 0
-2 -5 0
-4 -6 0
-3 -7 0
7 5 6 0
-8 -4 0
-9 -7 0
-1 -5 0
9 1 8 0
-9 -3 0
-8 -6 0
"
```

If you want to know which variable in the DIMACS output corresponds to which atom in GAPT, you can query the `DIMACSEncoding` object:

```
gapt> encoding decodeAtom 1
res16: at.logic.gapt.expr.Atom = R(p_3, h_1): o
```

GAPT also supports other SAT solvers such as MiniSAT or Glucose out of the box:

```
gapt> MiniSAT isValid PigeonHolePrinciple(3,2)
res17: Boolean = true
```

```
gapt> Glucose isValid PigeonHolePrinciple(3,2)
res18: Boolean = true
```

If you have another DIMACS-compliant solver installed or want to pass extra options to the SAT solver, you can pass a custom command to GAPT as well:

```
gapt> val solver = new ExternalSATSolver("minisat", "-mem-lim=1024")
solver: at.logic.gapt.provers.sat.ExternalSATSolver = ExternalSATSolver("minisat", "-mem-
    lim=1024")

gapt> solver isValid PigeonHolePrinciple(3,2)
res19: Boolean = true
```

GAPT can import DRUP proofs from Sat4j, Glucose, and PicoSAT:

```
gapt> Sat4j getDrupProof PigeonHolePrinciple(4,3)
res20: Option[at.logic.gapt.proofs.drup.DrupProof] =
Some([derive] ⊢
[derive] R(p_3, h_2) ⊢
[derive] ⊢ R(p_4, h_3)
[derive] R(p_3, h_3) ⊢ R(p_2, h_1), R(p_1, h_1)
[derive] R(p_4, h_2) ⊢ R(p_3, h_1)
[derive] R(p_3, h_1) ⊢ R(p_4, h_3)
[input] R(p_1, h_1), R(p_4, h_1) ⊢
[input] ⊢ R(p_1, h_1), R(p_1, h_2), R(p_1, h_3)
[input] R(p_3, h_2), R(p_4, h_2) ⊢
[input] R(p_1, h_1), R(p_2, h_1) ⊢
[input] R(p_2, h_1), R(p_3, h_1) ⊢
[input] ⊢ R(p_2, h_1), R(p_2, h_2), R(p_2, h_3)
[input] R(p_1, h_2), R(p_4, h_2) ⊢
[input] R(p_3, h_3), R(p_2, h_3) ⊢
[input] R(p_2, h_2), R(p_4, h_2) ⊢
[input] R(p_3, h_3), R(p_4, h_3) ⊢
[input] R(p_1, h_3), R(p_2, h_3) ⊢
[input] R(p_3, h_1), R(p_4, h_1) ⊢
[input] R(p_4, h_3), R(p_1, h_3) ⊢
[input] R(p_2, h_2), R(p_3, h_2) ⊢
[input] R(p_1, h_2), R(p_3, h_2) ⊢
[input] R...
```

Just as in the first-order prover interface, you can call `getResolutionProof` and `getLKProof` to get the proofs in the desired format:

```
gapt> Sat4j getLKProof PigeonHolePrinciple(4,3)
res21: Option[at.logic.gapt.proofs.lk.LKProof] =
Some([p402]
⊢
(R(p_1, h_1) ∨ R(p_1, h_2) ∨ R(p_1, h_3)) ∧
    (R(p_2, h_1) ∨ R(p_2, h_2) ∨ R(p_2, h_3)) ∧
    (R(p_3, h_1) ∨ R(p_3, h_2) ∨ R(p_3, h_3)) ∧
    (R(p_4, h_1) ∨ R(p_4, h_2) ∨ R(p_4, h_3)) ⊃
  R(p_2, h_1) ∧ R(p_1, h_1) ∨
    (R(p_3, h_1) ∧ R(p_1, h_1) ∨ R(p_3, h_1) ∧ R(p_2, h_1)) ∨
    (R(p_4, h_1) ∧ R(p_1, h_1) ∨
      R(p_4, h_1) ∧ R(p_2, h_1) ∨
      R(p_4, h_1) ∧ R(p_3, h_1)) ∨
    (R(p_2, h_2) ∧ R(p_1, h_2) ∨
      (R(p_3, h_2) ∧ R(p_1, h_2) ∨ R(p_3, h_2) ∧ R(p_2, h_2)) ∨
      (R(p_4, h_2) ∧ R(p_1, h_2) ∨
        R(p_4, h_2) ∧ R(p_2, h_2) ∨
        R(p_4, h_2) ∧ R(p_3, h_2))) ∨
    (R(p_2, h_3) ∧ R(p_1, h_3) ∨
      (R(p_3, h_3) ∧ R(p_1, h_3) ∨ R(p_3, h_3) ∧ R(p_2, h_3)) ∨
      (R(p_4, h_3) ∧ R(p_1, h_3) ∨
        R(p_4,...
```

4.3 MaxSAT solver interface

The MaxSAT interface supports generating optimal solutions for weighted partial MaxSAT instances: these consist of a list of hard clauses, which must be satisfied in the solution; and a list of weighted soft clauses, where weight of the satisfied soft clauses must be maximized. See [1] for an overview.

Let us solve a simple example using the MaxSAT solver from SAT4J:

```
gapt> MaxSat4j.solve(hard = hof"a|b|c", soft = Seq(hof"-a" -> 4, hof"-b" -> 3))
res22: Option[at.logic.gapt.models.PropositionalModel] =
Some(a:o -> false
b:o -> false
c:o -> true)
```

GAPT also supports other MaxSAT solvers out of the box, just write `OpenWBO` or `ToySolver` instead of `MaxSat4j`.

4.4 SMT solver interface

The SMT solver interface in GAPT supports validity queries for `QF_UF` formulas. For example we can check whether a quantifier-free formula is a quasi-tautology using veriT:

```
gapt> val f = hof"(a=b | a=c) & P(c) & P(b) -> P(a)"
f: at.logic.gapt.expr.Formula = (a = b ∨ a = c) ∧ P(c) ∧ P(b) ⊃ P(a)
```

```
gapt> VeriT isValid f
res23: Boolean = true
```

GAPT also supports Z3 and CVC4 out of the box (if they are installed):

```
gapt> Z3 isValid f
res24: Boolean = true

gapt> CVC4 isValid f
res25: Boolean = true
```

You can export QF_UF formulas (or sequents) as SMT-LIB benchmarks; note that we apply a drastic renaming to the constant symbols in order to support arbitrary (even Unicode) names in GAPT:

```
gapt> val (benchmark, typeRenaming, constantRenaming) = SmtLibExporter(Sequent() :+ f)
benchmark: String =
"(set-logic QF_UF)
(declare-sort t_i 0)
(declare-fun f_b () t_i)
(declare-fun f_P (t_i) Bool)
(declare-fun f_c () t_i)
(declare-fun f_a () t_i)
(assert (not (=> (and (and (or (= f_a f_b) (= f_a f_c)) (f_P f_c)) (f_P f_b)) (f_P f_a))))
(check-sat)
"
typeRenaming: Map[at.logic.gapt.expr.TBase,at.logic.gapt.expr.TBase] = Map(o -> Bool, i ->
     t_i)
constantRenaming: Map[at.logic.gapt.expr.Const,at.logic.gapt.expr.Const] = Map(b -> f_b:
    t_i, P:i>o -> f_P:t_i>Bool, c -> f_c:t_i, a -> f_a:t_i)
```

We can also extract instances for basic equality axioms (reflexivity, symmetry, and congruences) from veriT's proof output:

```
gapt> val Some(expansionProof) = VeriT getExpansionProof f
expansionProof: at.logic.gapt.proofs.expansion.ExpansionProof =
∀x ∀y (x = y ⊃ y = x)
  +^{a}
    (∀y (a = y ⊃ y = a) +^{b} ((a = b)+ ⊃ (b = a)-) +^{c} ((a = c)+ ⊃ (c = a)-)),
∀x1 ∀y1 (x1 = y1 ∧ P(x1) ⊃ P(y1))
  +^{b} (∀y1 (b = y1 ∧ P(b) ⊃ P(y1)) +^{a} ((b = a)+ ∧ P(b)+ ⊃ P(a)-))
  +^{c} (∀y1 (c = y1 ∧ P(c) ⊃ P(y1)) +^{a} ((c = a)+ ∧ P(c)+ ⊃ P(a)-))
:-
(((a = b)- ∨ (a = c)-) ∧ P(c)-) ∧ P(b)- ⊃ P(a)+

gapt> extractInstances(expansionProof) foreach println
a = b ⊃ b = a
a = c ⊃ c = a
b = a ∧ P(b) ⊃ P(a)
c = a ∧ P(c) ⊃ P(a)
(a = b ∨ a = c) ∧ P(c) ∧ P(b) ⊃ P(a)
```

4.5 First-order theorem prover interface

GAPT includes interfaces to several first-order theorem provers, such as Prover9, E prover, and LeanCoP. For Vampire, SPASS, E, Prover9, and Metis we can read back resolution proofs, and construct LK and expansion proofs from them. The LeanCoP interface reads back expansion proofs (and converts them to LK if desired).

Here is how you can get all of these kinds of proofs using Prover9:

```
gapt> val sequent = hos"p(0), !x (p(x) -> p(s(x))) :- p(s(s(0)))"
sequent: at.logic.gapt.proofs.HOLSequent = p(0), ∀x (p(x) ⊃ p(s(x))) ⊢ p(s(s(0)))
```

```
gapt> Prover9 isValid sequent
res27: Boolean = true

gapt> Prover9 getResolutionProof sequent
res28: Option[at.logic.gapt.proofs.resolution.ResolutionProof] =
Some([p13] ⊢   (Resolution(p8, Suc(0), p12, Ant(0)))
[p12] p(s(0)) ⊢  (Resolution(p9, Suc(0), p11, Ant(0)))
[p11] p(s(s(0))) ⊢  (Subst(p10, Substitution()))
[p10] p(s(s(0))) ⊢  (Input(p(s(s(0))) ⊢ ))
[p9] p(s(0)) ⊢ p(s(s(0))) (Subst(p6, Substitution(v0 -> s(0))))
[p8] ⊢ p(s(0))  (Resolution(p2, Suc(0), p7, Ant(0)))
[p7] p(0) ⊢ p(s(0)) (Subst(p6, Substitution(v0 -> 0)))
[p6] p(v0) ⊢ p(s(v0)) (Subst(p5, Substitution(x -> v0)))
[p5] p(x) ⊢ p(s(x)) (ImpR(p4, Suc(0)))
[p4] ⊢ p(x) ⊃ p(s(x)) (AllR(p3, Suc(0), x))
[p3] ⊢ ∀x (p(x) ⊃ p(s(x))) (Input( ⊢ ∀x (p(x) ⊃ p(s(x)))))
[p2] ⊢ p(0)  (Subst(p1, Substitution()))
[p1] ⊢ p(0)  (Input( ⊢ p(0)))
)

gapt> Prover9 getLKProof sequent
res29: Option[at.logic.gapt.proofs.lk.LKProof] =
Some([p11] ∀x (p(x) ⊃ p(s(x))), p(0) ⊢ p(s(s(0))) (ContractionLeftRule(p10, Ant(2), Ant
    (1)))
[p10] p(0), ∀x (p(x) ⊃ p(s(x))), ∀x (p(x) ⊃ p(s(x))) ⊢ p(s(s(0))) (CutRule(p5, Suc(0),
    p9, Ant(1)))
[p9] ∀x (p(x) ⊃ p(s(x))), p(s(0)) ⊢ p(s(s(0))) (CutRule(p8, Suc(0), p6, Ant(0)))
[p8] ∀x (p(x) ⊃ p(s(x))), p(s(0)) ⊢ p(s(s(0))) (ForallLeftRule(p7, Ant(0), p(x) ⊃ p(s(x)
    ), s(0), x))
[p7] p(s(0)) ⊃ p(s(s(0))), p(s(0)) ⊢ p(s(s(0))) (ImpLeftRule(p2, Suc(0), p6, Ant(0)))
[p6] p(s(s(0))) ⊢ p(s(s(0))) (LogicalAxiom(p(s(s(0))): o))
[p5] p(0), ∀x (p(x) ⊃ p(s(x))) ⊢ p(s(0)) (CutRule(p1, Suc(0), p4, Ant(1)))
[p4] ∀x (p(x) ⊃ p(s(x))), p(0) ⊢ p(s(0)) (ForallLeftRule(p3, Ant(0), p(x) ⊃ p(s(x)), 0,
    x))
[p3] p(0) ⊃ p(s(0)), p(0) ⊢ p(s(0)) ...

gapt> Prover9 getExpansionProof sequent
res30: Option[at.logic.gapt.proofs.expansion.ExpansionProof] =
```

```
Some(∀x (p(x) ⊃ p(s(x))) +^{0} (p(0)+ ⊃ p(s(0))-) +^{s(0)} (p(s(0))+ ⊃ p(s(s(0)))-),
p(0)-
:-
p(s(s(0)))+)
```

All of the above works with the E prover (EProver), SPASS (SPASS), Vampire (Vampire), and Metis (Metis) as well, we will just show EProver.getLKProof as an example:

```
gapt> EProver getLKProof sequent
res31: Option[at.logic.gapt.proofs.lk.LKProof] =
Some([p11] ∀x (p(x) ⊃ p(s(x))), p(0) ⊢ p(s(s(0))) (ContractionLeftRule(p10, Ant(2), Ant
    (1)))
[p10] p(0), ∀x (p(x) ⊃ p(s(x))), ∀x (p(x) ⊃ p(s(x))) ⊢ p(s(s(0))) (CutRule(p5, Suc(0),
    p9, Ant(1)))
[p9] ∀x (p(x) ⊃ p(s(x))), p(s(0)) ⊢ p(s(s(0))) (CutRule(p8, Suc(0), p6, Ant(0)))
[p8] ∀x (p(x) ⊃ p(s(x))), p(s(0)) ⊢ p(s(s(0))) (ForallLeftRule(p7, Ant(0), p(x) ⊃ p(s(x)
    ), s(0), x))
[p7] p(s(0)) ⊃ p(s(s(0))), p(s(0)) ⊢ p(s(s(0))) (ImpLeftRule(p2, Suc(0), p6, Ant(0)))
[p6] p(s(s(0))) ⊢ p(s(s(0))) (LogicalAxiom(p(s(s(0))): o))
[p5] p(0), ∀x (p(x) ⊃ p(s(x))) ⊢ p(s(0)) (CutRule(p1, Suc(0), p4, Ant(1)))
[p4] ∀x (p(x) ⊃ p(s(x))), p(0) ⊢ p(s(0)) (ForallLeftRule(p3, Ant(0), p(x) ⊃ p(s(x)), 0,
    x))
[p3] p(0) ⊃ p(s(0)), p(0) ⊢ p(s(0)) ...
```

Note that getLKProof only works for sequents without strong quantifiers (i.e. sequents that are already Skolemized); however getExpansionProof will happily return expansion proofs with Skolem quantifiers in that case:

```
gapt> Prover9 getExpansionProof hof"?x!y p x y -> !y?x p x y"
res32: Option[at.logic.gapt.proofs.expansion.ExpansionProof] =
Some(
:-
∃x ∀y p(x, y) +sk^{s_0} (∀y p(s_0, y) +^{s_1} p(s_0, s_1)-) ⊃
  ∀y ∃x p(x, y) +sk^{s_1} (∃x p(x, s_1) +^{s_0} p(s_0, s_1)+))
```

The LeanCoP interface supports the getExpansionProof as well:

```
gapt> LeanCoP getExpansionProof sequent
res33: Option[at.logic.gapt.proofs.expansion.ExpansionProof] =
Some(∀x (p(x) ⊃ p(s(x))) +^{0} (p(0)+ ⊃ p(s(0))-) +^{s(0)} (p(s(0))+ ⊃ p(s(s(0)))-),
p(0)-
:-
p(s(s(0)))+)
```

You can also export sequents as TPTP problems if you want to pass them to other provers manually:

```
gapt> TPTPFOLExporter(sequent)
res34: at.logic.gapt.formats.tptp.TptpFile =
fof(ant_0, axiom, p('0')).
fof(ant_1, axiom, ![X]: (p(X) => p(s(X)))).
fof(suc_0, conjecture, p(s(s('0')))).
```

You can also parse TPTP problems:

```
gapt> val tptp = TptpParser.load(pwd/"examples"/"import"/"irrationals.p")
tptp: at.logic.gapt.formats.tptp.TptpFile =
fof(a, axiom, i(sr2)).
fof(b, axiom, ~ i(two)).
fof(c, axiom, times(sr2, sr2) = two).
fof(d, axiom, ![X,Y,Z]: exp(exp(X, Y), Z) = exp(X, times(Y, Z))).
fof(e, axiom, ![X]: exp(X, two) = times(X, X)).
fof(f, conjecture, ?[X,Y]: (~ i(exp(X, Y)) & i(X) & i(Y))).

gapt> tptp.toSequent
res35: at.logic.gapt.proofs.HOLSequent =
i(sr2),
¬ i(two),
times(sr2, sr2) = two,
∀X ∀Y ∀Z exp(exp(X, Y), Z) = exp(X, times(Y, Z)),
∀X exp(X, two) = times(X, X)
⊢
∃X ∃Y (¬ i(exp(X, Y)) ∧ i(X) ∧ i(Y))
```

4.6 Built-in superposition prover

GAPT contains a simple built-in superposition prover called Escargot. It is used for proof replay to import proofs from other provers. Escargot can natively solve many-sorted problems, see Section 4.17. You can use it with the same interface as Prover9 and the other first-order provers:

```
gapt> val formula = fof"!x!y!z (x+y)+z=x+(y+z) & !x!y x+y=y+x -> d+a+c+b=a+b+c+d"
formula: at.logic.gapt.expr.FOLFormula =
∀x ∀y ∀z x + y + z = x + (y + z) ∧ ∀x ∀y x + y = y + x ⊃
  d + a + c + b = a + b + c + d

gapt> Escargot getResolutionProof formula
res36: Option[at.logic.gapt.proofs.resolution.ResolutionProof] =
Some([p34] ⊢   (Resolution(p1, Suc(0), p33, Ant(0)))
[p33] d + (b + (c + a)) = d + (b + (c + a)) ⊢ (Paramod(p17, Suc(0), true, p32, Ant(0), λx
      d + (b + (c + a)) = d + x))
[p32] d + (b + (c + a)) = d + (c + (b + a)) ⊢ (Paramod(p18, Suc(0), true, p31, Ant(0), λx
      x = d + (c + (b + a))))
[p31] b + (d + (c + a)) = d + (c + (b + a)) ⊢ (Paramod(p19, Suc(0), true, p30, Ant(0), λx
      b + x = d + (c + (b + a))))
[p30] b + (c + (d + a)) = d + (c + (b + a)) ⊢ (Paramod(p20, Suc(0), true, p29, Ant(0), λx
      b + (c + (d + a)) = d + x))
[p29] b + (c + (d + a)) = d + (b + a + c) ⊢ (Paramod(p21, Suc(0), true, p28, Ant(0), λx b
      + (c + (d + a)) = d + (x + c)))
[p28] b + (c + (d + a)) = d + (a + b + c) ⊢ (Paramod(p22, Suc(0), tru...
```

Escargot can also be used from the command-line using the escargot.sh script. This script expects a problem in TPTP format:

```
./escargot.sh examples/tptp/SET001-1.p
```

4.7 Built-in inductive theorem prover

GAPT contains a built-in inductive theorem prover: viper (Vienna inductive theorem prover). It can be started from the command line using the `viper.sh` script. It takes input in the TIP format [4]. Viper has a mode for analytic induction and a mode for the tree grammar-based method described in [5]. As of version 2.7 of GAPT, viper is in an early stage of development. By default, the `viper.sh` scripts tries several different strategies to solve the given problem, including analytic induction and tree-grammar-based methods. The `--help` argument shows the available options.

```
./viper.sh --treegrammar --cansolsize 2 3 --gramw scomp \
   examples/induction/prod_prop_31.smt2
```

You can also use Viper from within GAPT:

```
gapt> val problem = TipSmtParser.parse(FilePath("examples/tip/isaplanner/prop_06.smt2"))
problem: at.logic.gapt.formats.tip.TipProblem =
∀x0 p(S(x0)) = x0,
∀y plus(Z, y) = y,
∀z ∀y plus(S(z), y) = S(plus(z, y)),
∀y minus(Z, y) = Z,
∀z minus(S(z), Z) = S(z),
∀z ∀x2 minus(S(z), S(x2)) = minus(z, x2),
∀y0 ¬ Z = S(y0)
⊢
∀n ∀m minus(n, plus(n, m)) = Z

gapt> val Some(proof) = Viper(problem, verbosity=0)
proof: at.logic.gapt.proofs.lk.LKProof =
[p149] ∀z (minus(S(z:Nat): Nat, #c(Z: Nat)): Nat) = S(z),
∀y0 ¬ #c(Z: Nat) = S(y0),
∀x0 (p(S(x0)): Nat) = x0,
∀y (plus(#c(Z: Nat), y:Nat): Nat) = y,
∀y minus(#c(Z: Nat), y) = #c(Z: Nat),
∀z ∀x2 minus(S(z), S(x2)) = minus(z, x2),
∀z ∀y plus(S(z), y) = S(plus(z, y))
⊢
∀n ∀m minus(n, plus(n, m)) = #c(Z: Nat) (CutRule(p22, Suc(0), p148, Ant(3)))
[p148] ∀z (minus(S(z:Nat): Nat, #c(Z: Nat)): Nat) = S(z),
∀y0 ¬ #c(Z: Nat) = S(y0),
∀x0 (p(S(x0)): Nat) = x0,
∀m
  ((⊤  ⊃ minus(#c(Z: Nat), plus(#c(Z: Nat), m:Nat)) = #c(Z: Nat)) ∧
     ∀n_0
       (minus(n_0, plus(n_0, m)) = #c(Z: Nat) ⊃
         minus(S(n_0), plus(S(n_0), m)) = #c(Z: Nat)) ⊃
   ∀n minus(n, plus(n, m)) = #c(Z: Nat)),
∀y plus(#c(Z: Nat), y) = y,
```

```
∀y minus(#c(Z: Nat), y) = #c(Z: Nat),
∀z ∀x...
```

4.8 Built-in tableaux prover

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

```
gapt> solvePropositional(hof"a -> b -> a&b").get
res37: at.logic.gapt.proofs.lk.LKProof =
[p5] ⊢ a ⊃ b ⊃ a ∧ b (ImpRightRule(p4, Ant(0), Suc(0)))
[p4] a ⊢ b ⊃ a ∧ b (ImpRightRule(p3, Ant(1), Suc(0)))
[p3] a, b ⊢ a ∧ b (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] b ⊢ b  (LogicalAxiom(b:o))
[p1] a ⊢ a  (LogicalAxiom(a:o))
```

The tableaux prover can also prove quasi-tautologies if you call it as solveQuasiPropositional:

```
gapt> solveQuasiPropositional(hof"a = b & f a = b -> a = f(f(b))").get
res38: at.logic.gapt.proofs.lk.LKProof =
[p9] ⊢ a = b ∧ f(a) = b ⊃ a = f(f(b)) (ImpRightRule(p8, Ant(0), Suc(0)))
[p8] a = b ∧ f(a) = b ⊢ a = f(f(b)) (AndLeftRule(p7, Ant(1), Ant(0)))
[p7] f(a) = b, a = b ⊢ a = f(f(b)) (EqualityLeftRule(p6, Ant(0), Ant(1), λx f(x) = b))
[p6] a = b, f(b) = b ⊢ a = f(f(b)) (EqualityRightRule(p5, Ant(0), Suc(0), λx x = f(f(b)))
    )
[p5] a = b, f(b) = b ⊢ b = f(f(b)) (WeakeningLeftRule(p4, a = b))
[p4] f(b) = b ⊢ b = f(f(b)) (EqualityRightRule(p3, Ant(0), Suc(0), λx b = f(x)))
[p3] f(b) = b ⊢ b = f(b) (EqualityRightRule(p2, Ant(0), Suc(0), λx b = x))
[p2] f(b) = b ⊢ b = b (WeakeningLeftRule(p1, f(b) = b))
[p1] ⊢ b = b  (ReflexivityAxiom(b))
```

4.9 Cut-elimination (Gentzen's method)

The GAPT-system contains an implementation of Gentzen-style reductive cut-elimination. It can be used as follows: first we load a proof p with cuts:

```
gapt> val p = examples.fol1.proof
p: at.logic.gapt.proofs.lk.LKProof =
[p25] ∀x ∀y (P(x, y) ⊃ Q(x, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (CutRule(p9, Suc(0),
    p24, Ant(0)))
[p24] ∀x ∃y (¬ P(x, y) ∨ Q(x, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ForallLeftRule(p23,
    Ant(0), ∃y (¬ P(x, y) ∨ Q(x, y)), b, x))
[p23] ∃y (¬ P(b, y) ∨ Q(b, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ExistsLeftRule(p22, Ant
    (0), y, y))
[p22] ¬ P(b, y) ∨ Q(b, y) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ExistsRightRule(p21, Suc(0),
     ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)), b, x))
[p21] ¬ P(b, y) ∨ Q(b, y) ⊢ ∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) (ExistsRightRule(p20, Suc(0), ¬
    Q(b, y) ⊃ ¬ P(b, y), y, y))
```

```
[p20] ¬ P(b, y) ∨ Q(b, y) ⊢ ¬ Q(b, y) ⊃ ¬ P(b, y) (ContractionRightRule(p19, Suc(1),
    Suc(0)))
[p19] ¬ P(b, y) ∨ Q(b, y) ⊢ ¬ Q(b, y) ⊃ ¬ P(b, y), ¬ Q(b, y) ⊃ ¬ P(b, y) (OrLeftRule(
    p14, Ant(0), p18, Ant(0...
```

and then call the cut-elimination procedure:

```
gapt> val q = ReductiveCutElimination( p )
q: at.logic.gapt.proofs.lk.LKProof =
[p14] ∀x ∀y (P(x, y) ⊃ Q(x, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ForallLeftRule(p13,
    Ant(0), ∀y (P(x, y) ⊃ Q(x, y)), b, x))
[p13] ∀y (P(b, y) ⊃ Q(b, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ForallLeftRule(p12, Ant
    (0), P(b, y) ⊃ Q(b, y), a, y))
[p12] P(b, a) ⊃ Q(b, a) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ExistsRightRule(p11, Suc(0), ∃
    y (¬ Q(x, y) ⊃ ¬ P(x, y)), b, x))
[p11] P(b, a) ⊃ Q(b, a) ⊢ ∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) (ExistsRightRule(p10, Suc(0), ¬ Q(
    b, y) ⊃ ¬ P(b, y), a, y))
[p10] P(b, a) ⊃ Q(b, a) ⊢ ¬ Q(b, a) ⊃ ¬ P(b, a) (ContractionRightRule(p9, Suc(1), Suc
    (0)))
[p9] P(b, a) ⊃ Q(b, a) ⊢ ¬ Q(b, a) ⊃ ¬ P(b, a), ¬ Q(b, a) ⊃ ¬ P(b, a) (ImpRightRule(p8
    , Ant(0), Suc(1)))
[p8] ¬ Q(b, a), P(b, a) ⊃ Q(b, a) ⊢ ¬ Q(b, a) ⊃ ¬ P(b, a), ¬ P(b, a) (WeakeningLeftRule
    (p7,...
```

4.10 Induction-elimination

As an extension of Gentzen cut-elimination, GAPT can also eliminate induction inferences in a restricted class of proofs with quantifier-free conclusions.

```
gapt> val p = instanceProof(examples.induction.numbers.pluscomm,
                            Seq(le"s (s 0 : nat)", le"0 : nat"))
p: at.logic.gapt.proofs.lk.LKProof =
[p67] ∀x ∀y ((s(x:nat): nat) + (y:nat): nat) = s(x + y),
∀x 0 + x = x
⊢
s(s(0)) + 0 = 0 + s(s(0)) (CutRule(p63, Suc(0), p66, Ant(0)))
[p66] ∀x ∀y ((x:nat) + (y:nat): nat) = y + x ⊢ s(s(0:nat): nat) + 0 = 0 + s(s(0)) (
    ForallLeftRule(p65, Ant(0), ∀y ((x:nat) + (y:nat): nat) = y + x, s(s(0:nat): nat), x:
    nat))
[p65] ∀y (s(s(0:nat): nat) + (y:nat): nat) = y + s(s(0)) ⊢ s(s(0)) + 0 = 0 + s(s(0)) (
    ForallLeftRule(p64, Ant(0), (s(s(0:nat): nat) + (y:nat): nat) = y + s(s(0)), 0:nat, y:
    nat))
[p64] (s(s(0:nat): nat) + 0: nat) = 0 + s(s(0)) ⊢ s(s(0)) + 0 = 0 + s(s(0)) (LogicalAxiom
    ((s(s(0:nat): nat) + 0: nat) = 0 + s(s(0))))
[p63] ∀x ∀y ((s(x:nat): nat) + (y:nat): nat) = s(x + y),
∀x 0 + x = x
⊢
∀x ∀y x + y = y + x  (ContractionLeftRule(p62, Ant(2),...
```

```
gapt> val q = ReductiveCutElimination.eliminateInduction(p)(
                                  examples.induction.numbers.ctx)
q: at.logic.gapt.proofs.lk.LKProof =
[p48] ∀x ∀y ((s(x:nat): nat) + (y:nat): nat) = s(x + y),
∀x 0 + x = x
⊢
s(s(0)) + 0 = 0 + s(s(0)) (ContractionLeftRule(p47, Ant(2), Ant(1)))
[p47] ∀x ((0:nat) + (x:nat): nat) = x,
∀x ∀y s(x:nat) + y = s(x + y),
∀x ∀y s(x) + y = s(x + y)
⊢
s(s(0)) + 0 = 0 + s(s(0)) (ContractionLeftRule(p46, Ant(0), Ant(2)))
[p46] ∀x ((0:nat) + (x:nat): nat) = x,
∀x ∀y s(x:nat) + y = s(x + y),
∀x 0 + x = x,
∀x ∀y s(x) + y = s(x + y)
⊢
s(s(0)) + 0 = 0 + s(s(0)) (ContractionLeftRule(p45, Ant(3), Ant(2)))
[p45] ∀x ∀y ((s(x:nat): nat) + (y:nat): nat) = s(x + y),
∀x 0 + x = x,
∀x 0 + x = x,
∀x 0 + x = x,
∀x ∀y s(x) + y = s(x + y)
⊢
s(s(0)) + 0 = 0 + s(s(0)) (ForallLeftRule(p44, Ant(0), ∀y ((s(x:nat): nat) + (y:nat): nat
    ) = s(x + y), s(0:nat): nat, x:nat))
[p44] ...
```

The resulting proof q contains only atomic cuts, and we can view its Herbrand sequent by converting to an expansion proof:

```
gapt> LKToExpansionProof(q)
res39: at.logic.gapt.proofs.expansion.ExpansionProof =
∀x ∀y ((s(x:nat): nat) + (y:nat): nat) = s(x + y)
  +^{0} (∀y s(0) + y = s(0 + y) +^{0} (s(0) + 0 = s(0 + 0))-)
  +^{s(0)} (∀y s(s(0)) + y = s(s(0) + y) +^{0} (s(s(0)) + 0 = s(s(0) + 0))-),
∀x ((0:nat) + (x:nat): nat) = x
  +^{0} (0 + 0 = 0)-
  +^{s(0)} (0 + s(0) = s(0))-
  +^{s(s(0))} (0 + s(s(0)) = s(s(0)))-
:-
((s(s(0:nat): nat) + 0: nat) = 0 + s(s(0)))+
```

4.11 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> var p: LKProof = LogicalAxiom(hof"P(x,y)")
p: at.logic.gapt.proofs.lk.LKProof =
[p1] P(x, y) ⊢ P(x, y) (LogicalAxiom(P(x, y): o))

gapt> p = ExistsRightRule(p, hof"?x P(x,y)", le"x")
p: at.logic.gapt.proofs.lk.LKProof = [p2] P(x, y) ⊢ ∃x P(x, y) (ExistsRightRule(p1, Suc
    (0), P(x, y): o, x, x))
[p1] P(x, y) ⊢ P(x, y) (LogicalAxiom(P(x, y): o))

gapt> p = ForallLeftRule(p, hof"!y P(x,y)", le"y")
p: at.logic.gapt.proofs.lk.LKProof = [p3] ∀y P(x, y) ⊢ ∃x P(x, y) (ForallLeftRule(p2, Ant
    (0), P(x, y): o, y, y))
[p2] P(x, y) ⊢ ∃x P(x, y) (ExistsRightRule(p1, Suc(0), P(x, y): o, x, x))
[p1] P(x, y) ⊢ P(x, y) (LogicalAxiom(P(x, y): o))

gapt> p = ForallRightRule(p, hof"!y?x P(x,y)", fov"y")
p: at.logic.gapt.proofs.lk.LKProof = [p4] ∀y P(x, y) ⊢ ∀y ∃x P(x, y) (ForallRightRule(p3,
     Suc(0), y, y))
[p3] ∀y P(x, y) ⊢ ∃x P(x, y) (ForallLeftRule(p2, Ant(0), P(x, y): o, y, y))
[p2] P(x, y) ⊢ ∃x P(x, y) (ExistsRightRule(p1, Suc(0), P(x, y): o, x, x))
[p1] P(x, y) ⊢ P(x, y) (LogicalAxiom(P(x, y): o))

gapt> p = ExistsLeftRule(p, hof"?x!y P(x,y)", fov"x")
p: at.logic.gapt.proofs.lk.LKProof = [p5] ∃x ∀y P(x, y) ⊢ ∀y ∃x P(x, y) (ExistsLeftRule(
    p4, Ant(0), x, x))
[p4] ∀y P(x, y) ⊢ ∀y ∃x P(x, y) (ForallRightRule(p3, Suc(0), y, y))
[p3] ∀y P(x, y) ⊢ ∃x P(x, y) (ForallLeftRule(p2, Ant(0), P(x, y): o, y, y))
[p2] P(x, y) ⊢ ∃x P(x, y) (ExistsRightRule(p1, Suc(0), P(x, y): o, x, x))
[p1] P(x, y) ⊢ P(x, y) (LogicalAxiom(P(x, y): o))

gapt> val q = skolemize(p)
q: at.logic.gapt.proofs.lk.LKProof =
[p3] ∀y P(s_0, y) ⊢ ∃x P(x, s_1) (ForallLeftRule(p2, Ant(0), P(s_0, y): o, s_1, y))
[p2] P(s_0, s_1) ⊢ ∃x P(x, s_1) (ExistsRightRule(p1, Suc(0), P(x, s_1): o, s_0, x))
[p1] P(s_0, s_1) ⊢ P(s_0, s_1) (LogicalAxiom(P(s_0, s_1): o))
```

4.12 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, and axioms of the form $A \vdash A$; $\bot \vdash$, or $\vdash \top$. The implementation is based on Lemma 6.5 of [13]. 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 instance, suppose pr is the following proof:

$$\dfrac{\dfrac{P(a) \vdash P(a)}{a = b, P(a) \vdash P(a)}\text{(w:l)}}{a = b, P(a) \vdash P(b)} =\text{:r}$$

First, we construct the proof pr:

```
gapt> val axpa = LogicalAxiom( fof"P(a)" )
axpa: at.logic.gapt.proofs.lk.LogicalAxiom =
[p1] P(a) ⊢ P(a)  (LogicalAxiom(P(a): o))

gapt> val axpb = LogicalAxiom( fof"P(b)" )
axpb: at.logic.gapt.proofs.lk.LogicalAxiom =
[p1] P(b) ⊢ P(b)  (LogicalAxiom(P(b): o))

gapt> val proof = WeakeningLeftRule( axpa, fof"a=b" )
proof: at.logic.gapt.proofs.lk.WeakeningLeftRule =
[p2] a = b, P(a) ⊢ P(a) (WeakeningLeftRule(p1, a = b))
[p1] P(a) ⊢ P(a)  (LogicalAxiom(P(a): o))

gapt> val pr = EqualityRightRule( proof, fof"a=b", Suc( 0 ), fof"P(b)" )
pr: at.logic.gapt.proofs.lk.EqualityRightRule =
[p3] a = b, P(a) ⊢ P(b) (EqualityRightRule(p2, Ant(0), Suc(0), λx P(x): o))
[p2] a = b, P(a) ⊢ P(a) (WeakeningLeftRule(p1, a = b))
[p1] P(a) ⊢ P(a)  (LogicalAxiom(P(a): o))
```

In order to apply interpolation, we need to specify a partition of the end-sequent into $\Gamma_1 \vdash \Delta_1$ and $\Gamma_2 \vdash \Delta_2$, i.e. into the negative and positive part, respectively. In this case, we set $\Delta_1 = \{P(b)\}$, $\Gamma_2 = \{a = b, P(a)\}$ and $\Gamma_1 = \Delta_2 = \emptyset$.

Then we can call `ExtractInterpolant( pr, positivePart )`, which returns the interpolant $I = (a = b \supset \neg P(a))$ of pr:

```
gapt> val I = ExtractInterpolant(pr, Seq(Ant(0), Ant(1)))
I: at.logic.gapt.expr.Formula = a = b ⊃ ¬ P(a)
```

4.13 Expansion trees

Expansion proofs are a compact representation of the quantifier inferences in a proof. They have originally been introduced in [11]. GAPT contains an implementation of expansion proofs with cut for higher-order logic, including functions to extract expansion trees from proofs, to merge expansion trees, to prune and transform them in various ways, to eliminate first-order cuts, and to display them in the graphical user interface.

An expansion tree contains the instances of the quantifiers for a formula. In order to represent a proof of a sequent we use sequents of expansion trees. An expansion proof consists of such a sequent of expansion trees where the strong quantifiers do not form cycles. For example we can obtain an expansion proof by:

```
gapt> val expansion = LKToExpansionProof(examples.fol1.proof)
```

```
expansion: at.logic.gapt.proofs.expansion.ExpansionProof =
∀X (X ⊃ X)
  +^{∀x ∃y (¬ P(x, y) ∨ Q(x, y))}
    (∀x ∃y (¬ P(x, y) ∨ Q(x, y)) +ev^{x}
        (∃y (¬ P(x, y) ∨ Q(x, y)) +^{a} (¬ P(x, a)- ∨ Q(x, a)+)) ⊃
      ∀x ∃y (¬ P(x, y) ∨ Q(x, y))
        +^{b} (∃y (¬ P(b, y) ∨ Q(b, y)) +ev^{y} (¬ P(b, y)+ ∨ Q(b, y)-))),
∀x ∀y (P(x, y) ⊃ Q(x, y))
  +^{x} (∀y (P(x, y) ⊃ Q(x, y)) +^{a} (P(x, a)+ ⊃ Q(x, a)-))
:-
∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y))
  +^{b} (∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) +^{y} (¬ Q(b, y)+ ⊃ ¬ P(b, y)-))
```

The expansion proof returned by LKToExpansionProof contains the quantifier inferences of the proof in LK and the quantified cuts. Quantifier-free cuts are not included, as they can never be involved in quantifier inferences.

Expansion proofs have shallow and deep sequents. The shallow sequent corresponds to the end-sequent of the proof in LK, and is the sequent that is proven. The deep sequent consists of instances of the shallow sequent: the (quasi-)tautology of the deep sequent implies the validity of the shallow sequent.

```
gapt> expansion.shallow
res40: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.Formula] = ∀X (X ⊃ X), ∀x ∀y (P(x,
     y) ⊃ Q(x, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y))

gapt> expansion.deep
res41: at.logic.gapt.proofs.Sequent[at.logic.gapt.expr.Formula] = ¬ P(x, a) ∨ Q(x, a) ⊃
    ¬ P(b, y) ∨ Q(b, y), P(x, a) ⊃ Q(x, a) ⊢ ¬ Q(b, y) ⊃ ¬ P(b, y)

gapt> Sat4j isValid expansion.deep
res42: Boolean = true
```

This expansion proof contains a cut. Cuts are stored as expansions of the second-order formula $\forall X(X \supset X)$ in the antecedent. GAPT contains a procedure to eliminate such cuts in expansion proofs as described in [10]:

```
gapt> eliminateCutsET(expansion)
res43: at.logic.gapt.proofs.expansion.ExpansionProof =
∀x ∀y (P(x, y) ⊃ Q(x, y))
  +^{b} (∀y (P(b, y) ⊃ Q(b, y)) +^{a} (P(b, a)+ ⊃ Q(b, a)-))
:-
∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y))
  +^{b} (∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) +^{a} (¬ Q(b, a)+ ⊃ ¬ P(b, a)-))
```

We can also convert expansion proofs to LK; this works even in the presence of cuts, and also if the proof requires equational reasoning:

```
gapt> ExpansionProofToLK(expansion).get
res44: at.logic.gapt.proofs.lk.LKProof =
```

```
[p21] ∀x ∀y (P(x, y) ⊃ Q(x, y)) ⊢ ∃x ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)) (ExistsRightRule(p20,
    Suc(0), ∃y (¬ Q(x, y) ⊃ ¬ P(x, y)), b, x))
[p20] ∀x ∀y (P(x, y) ⊃ Q(x, y)) ⊢ ∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) (CutRule(p9, Suc(0), p19,
    Ant(0)))
[p19] ∀x ∃y (¬ P(x, y) ∨ Q(x, y)) ⊢ ∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) (ForallLeftRule(p18, Ant
    (0), ∃y (¬ P(x, y) ∨ Q(x, y)), b, x))
[p18] ∃y (¬ P(b, y) ∨ Q(b, y)) ⊢ ∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) (ExistsLeftRule(p17, Ant(0)
    , y, y))
[p17] ¬ P(b, y) ∨ Q(b, y) ⊢ ∃y (¬ Q(b, y) ⊃ ¬ P(b, y)) (ExistsRightRule(p16, Suc(0), ¬
    Q(b, y) ⊃ ¬ P(b, y), y, y))
[p16] ¬ P(b, y) ∨ Q(b, y) ⊢ ¬ Q(b, y) ⊃ ¬ P(b, y) (ImpRightRule(p15, Ant(0), Suc(0)))
[p15] ¬ Q(b, y), ¬ P(b, y) ∨ Q(b, y) ⊢ ¬ P(b, y) (NegLeftRule(p14, Suc(0)))
[p14] ¬ P(b, y) ∨ Q(b, y) ⊢ Q(b, y), ...
```

You can also view this expansion proof in the graphical user interface by calling:

```
gapt> prooftool( expansion )
```

A window then opens that displays the shallow sequent of `expansion`. You can selectively expand quantifiers by clicking on them, see [9] for a detailed description.

4.14 Cut-elimination by resolution (CERES)

Cut-elimination by resolution (CERES) is a method which transforms a proof with arbitrary cut-formulas into one with only atomic cuts [2, 3]. Since expansion proofs can be extracted directly from a proof with quantifier-free cut-formulas, we can skip the elimination of atomic cuts.

For instance, the example proof `Pi2Pigeonhole` formalizes the fact that given an aviary with two holes and an infinite number of pigeons, one hole has to house at least two pigeons. The pigeons and the holes are represented by numerals in unary notation with zero 0 and successor s. The function symbol f maps pigeons to holes, which allows us to state the mapping of pigeons to holes as $\forall x(f(x) = 0 \lor f(x) = s(0))$. The actual statement to prove is then $\exists x \exists y(s(x) \leq y \land f(x) = f(y))$. In order to prove it we also need to axiomatize $\leq$ with $\forall x \forall y(s(x) \leq y \supset x \leq y)$ and transitivity $\forall x \forall y(x \leq M(x,y) \land M(x,y) \leq y \supset x \leq y)$ in its skolemized form.

We can extract the cut formulas using the `cutFormulas` command and find two cuts on quantified formulas: $\forall x \exists y(x \leq y \land f(y) = 0)$ and $\forall x \exists y(x \leq y \land f(y) = s(0))$. This corresponds to a case distinction for each of the two holes which may contain the collision. The actual simplification is performed using the `CERES` command. Please note that the input proof must be regular and have a skolemized end-sequent. The commands `regularize` and `skolemize` provide this functionality, if necessary.

```
gapt> prooftool(Pi2Pigeonhole.proof)

gapt> cutFormulas(Pi2Pigeonhole.proof) filter {containsQuantifier(_)} foreach println
∀x ∃y (x <= y ∧ f(y) = s(0))
∀x ∃y (x <= y ∧ f(y) = 0)

gapt> val acnf = CERES(Pi2Pigeonhole.proof)
```

```
acnf: at.logic.gapt.proofs.lk.LKProof =
[p220] ∀x_0 (f(x_0) = 0 ∨ f(x_0) = s(0)),
∀x_1 ∀y_0 (x_1 <= M(x_1, y_0) ∧ y_0 <= M(x_1, y_0))
⊢
∃x ∃y_1 (s(x) <= y_1 ∧ f(x) = f(y_1)) (CutRule(p1, Suc(0), p219, Ant(2)))
[p219] ∀x_0 (f(x_0) = 0 ∨ f(x_0) = s(0)),
∀x_1 ∀y_0 (x_1 <= M(x_1, y_0) ∧ y_0 <= M(x_1, y_0)),
0 = 0
⊢
∃x ∃y_1 (s(x) <= y_1 ∧ f(x) = f(y_1)) (ContractionRightRule(p218, Suc(1), Suc(0)))
[p218] ∀x_0 (f(x_0) = 0 ∨ f(x_0) = s(0)),
∀x_1 ∀y_0 (x_1 <= M(x_1, y_0) ∧ y_0 <= M(x_1, y_0)),
0 = 0
⊢
∃x_0 ∃y_1 (s(x_0) <= y_1 ∧ f(x_0) = f(y_1)),
∃x ∃y_1 (s(x) <= y_1 ∧ f(x) = f(y_1)) (ContractionLeftRule(p217, Ant(3), Ant(1)))
[p217] ∀x_1 ∀y_0 (x_1 <= M(x_1, y_0) ∧ y_0 <= M(x_1, y_0)),
∀x_0 (f(x_0) = 0 ∨ f(x_0) = s(0)),
0 = 0,
∀x_0 (f(x_0) = 0 ∨ f(x_0) = s(0))
⊢
∃x_0 ∃y_1 (s(x_0) <= y_1...

gapt> prooftool(acnf)

gapt> val et = LKToExpansionProof(acnf)
et: at.logic.gapt.proofs.expansion.ExpansionProof =
∀x_0 (f(x_0) = 0 ∨ f(x_0) = s(0))
  +^{M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))),
     s(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))), x_6)))}
    ((f(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))),
           s(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))), x_6)))) =
        0)- ∨
      (f(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))),
           s(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))), x_6)))) =
        s(0))-)
  +^{M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))), x_6)}
    ((f(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))), x_6)) = 0)- ∨
      (f(M(s(M(s(M(x_2, x_4)), s(M(x_2, x_4)))), x_6)) = s(0))-)
  +^{M(s(M(x_2, x_4)), s(M(s(M(x_2, x_4)), x_6)))}
    ((f(M(s(M(x_2, x_4)), s(M(s(M(x_2, x_4)), x_6)))) = 0)- ∨
      (f(M(s(M(x_2, x_4)), s(M(s(M(x_2, x_4)), x_6)))) = s(0))-)...

gapt> prooftool(et)
```

4.15 Cut-introduction

The cut-introduction algorithm as described in [8, 7, 6] 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 3.3) and input them to the cut-introduction algorithm.

Take an example proof:

```
gapt> val p = LinearExampleProof(9)
p: at.logic.gapt.proofs.lk.LKProof =
[p36] ∀x (P(x) ⊃ P(s(x))), P(0) ⊢ P(s(s(s(s(s(s(s(s(s(0)))))))))) (ContractionLeftRule(
    p35, Ant(0), Ant(1)))
[p35] ∀x (P(x) ⊃ P(s(x))), ∀x (P(x) ⊃ P(s(x))), P(0) ⊢ P(s(s(s(s(s(s(s(s(s(0)))))))))) (
    ForallLeftRule(p34, Ant(0), P(x) ⊃ P(s(x)), s(s(s(s(s(s(s(s(0)))))))), x))
[p34] P(s(s(s(s(s(s(s(s(0)))))))) ⊃ P(s(s(s(s(s(s(s(s(s(0))))))))),
∀x (P(x) ⊃ P(s(x))),
P(0)
⊢
P(s(s(s(s(s(s(s(s(s(0)))))))))) (ImpLeftRule(p32, Suc(0), p33, Ant(0)))
[p33] P(s(s(s(s(s(s(s(s(s(0)))))))))) ⊢ P(s(s(s(s(s(s(s(s(s(0)))))))))) (LogicalAxiom(P(s
    (s(s(s(s(s(s(s(s(0)))))))))): o))
[p32] ∀x (P(x) ⊃ P(s(x))), P(0) ⊢ P(s(s(s(s(s(s(s(s(0))))))))) (ContractionLeftRule(p31,
    Ant(0), Ant(1)))
[p31] ∀x (P(x) ⊃ P(s(x))), ∀x (P(x) ⊃ P(s(x))), P(0) ⊢ P(s(s(s(s(s(s(s(s(0...
```

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

```
gapt> val q = CutIntroduction(p, method=DeltaTableMethod())
q: Option[at.logic.gapt.proofs.lk.LKProof] =
Some([p27] ∀x (P(x) ⊃ P(s(x))), P(0) ⊢ P(s(s(s(s(s(s(s(s(s(0)))))))))) (CutRule(p14, Suc
    (0), p26, Ant(0)))
[p26] ∀x1 (P(x1) ⊃ P(s(s(s(x1))))), P(0) ⊢ P(s(s(s(s(s(s(s(s(s(0)))))))))) (
    ContractionLeftRule(p25, Ant(1), Ant(0)))
[p25] ∀x1 (P(x1) ⊃ P(s(s(s(x1))))),
∀x1 (P(x1) ⊃ P(s(s(s(x1))))),
P(0)
⊢
P(s(s(s(s(s(s(s(s(s(0)))))))))) (ForallLeftRule(p24, Ant(1), P(x1) ⊃ P(s(s(s(x1)))), 0,
    x1))
[p24] ∀x1 (P(x1) ⊃ P(s(s(s(x1))))),
P(0) ⊃ P(s(s(s(0)))),
P(0)
⊢
P(s(s(s(s(s(s(s(s(s(0)))))))))) (ContractionLeftRule(p23, Ant(1), Ant(0)))
[p23] ∀x1 (P(x1) ⊃ P(s(s(s(x1))))),
∀x1 (P(x1) ⊃ P(s(s(s(x1))))),
P(0) ⊃ P(s(s(s(0)))),
P(0)
⊢
```

```
P(s(s(s(s(s(s(s(s(s(0)))))))))) (ForallLeftRule(p22, Ant(3), P(x1) ⊃ P(s(s(s(x1)))), s(s(
    s(0))), x1))
[p22] ∀x1...
```

You can also try `MaxSATMethod(1,2)`, this uses a reduction to a MaxSAT problem and an external MaxSAT-solver to a minimal grammar corresponding to a proof with a cut with two cuts, one with 1 quantifier, one with 2 quantifiers. If you want to see more information about what is happening during cut-introduction, you can make the output more verbose by running:

```
gapt> CutIntroduction.makeVerbose()
```

4.16 Tree grammars

The cut-introduction method described in Section 4.15 is based on the use of certain tree grammars for representing Herbrand-disjunctions. These are totally rigid acyclic tree grammars (TRATGs) and vectorial TRATGs (VTRATGs). As shown in [7], these grammars are intimately related to the structure of proofs with cuts. GAPT contains an implementation of these tree grammars, and given a finite tree language (i.e., a set of terms), is able to automatically find a (V)TRATG that covers this language:

```
gapt> val lang = 1 to 18 map { Numeral(_) }
lang: scala.collection.immutable.IndexedSeq[at.logic.gapt.expr.FOLTerm] = Vector(s(0), s(s
    (0)), s(s(s(0))), s(s(s(s(0)))), s(s(s(s(s(0))))), s(s(s(s(s(s(0)))))), s(s(s(s(s(s(s
    (0))))))), s(s(s(s(s(s(s(s(0)))))))), s(s(s(s(s(s(s(s(s(0))))))))), s(s(s(s(s(s(s(s(s(
    s(0)))))))))), s(s(s(s(s(s(s(s(s(s(s(0))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))
    )), s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))
    ))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s
    (0)))))))))))))))), s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))), s(s(s(s(s(
    s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))))))

gapt> val grammar = findMinimalVTRATG(lang.toSet, 2)
grammar: at.logic.gapt.grammars.VTRATG =
Non-terminal vectors: (x_0), (x_1), (x_2)
Terminals: 0, s:i>i

x_0 → s(s(s(x_1)))

x_0 → s(x_1)

x_1 → s(s(x_2))

x_1 → s(x_2)

x_1 → x_2

x_2 → 0

x_2 → s(s(s(s(s(0)))))
```

```
x_2 → s(s(s(s(s(s(s(s(s(s(s(0)))))))))))

gapt> lang.toSet subsetOf grammar.language
res51: Boolean = true
```

You can also find minimal sub-grammars that still generate certain terms:

```
gapt> minimizeVTRATG(grammar, (1 to 5).map(Numeral(_)).toSet)
res52: at.logic.gapt.grammars.VTRATG =
Non-terminal vectors: (x_0), (x_1), (x_2)
Terminals: 0, s:i>i

x_0 → s(s(s(x_1))))

x_0 → s(x_1)

x_1 → s(s(x_2))

x_1 → s(x_2)

x_1 → x_2

x_2 → 0
```

4.17 Many-sorted logic

The lambda calculus implemented in GAPT supports multiple base sorts. When entering a many-sorted formula you need to provide enough type annotations to infer the types:

```
gapt> val formula = hof"P(cons(0:nat, cons(s(0), nil)): list)"
formula: at.logic.gapt.expr.Formula = P(cons(0:nat, cons(s(0): nat, nil:list): list)): o

gapt> val axiom = hof"P(nil) & !x!y!z (P(x) -> P(cons(y: nat, cons(z, x)): list))"
axiom: at.logic.gapt.expr.Formula = P(nil:list) ∧ ∀x ∀y ∀z (P(x) ⊃ P(cons(y:nat, cons(z:
    nat, x): list)))

gapt> val problem = hof"$axiom -> $formula"
problem: at.logic.gapt.expr.Formula =
P(nil:list) ∧ ∀x ∀y ∀z (P(x) ⊃ P(cons(y:nat, cons(z:nat, x): list))) ⊃
  P(cons(0, cons(s(0), nil)))
```

The built-in prover Escargot can natively solve many-sorted problems:

```
gapt> Escargot.getExpansionProof(problem)
res53: Option[at.logic.gapt.proofs.expansion.ExpansionProof] =
Some(
:-
P(nil:list)- ∧
    (∀x ∀y ∀z (P(x) ⊃ P(cons(y:nat, cons(z:nat, x): list)))
```

```
      +^{nil}
        (∀y ∀z (P(nil) ⊃ P(cons(y, cons(z, nil))))
          +^{0}
            (∀z (P(nil) ⊃ P(cons(0, cons(z, nil))))
              +^{s(0)} (P(nil)+ ⊃ P(cons(0, cons(s(0), nil)))-)))) ⊃
  P(cons(0, cons(s(0), nil)))+)
```

For other provers we need to reduce this problem to a first-order one. For example in this way we can obtain a many-sorted expansion proof from Prover9 (which only supports a single sort):

```
gapt> val reduction = PredicateReductionET |> ErasureReductionET
reduction: at.logic.gapt.proofs.reduction.Reduction[at.logic.gapt.proofs.HOLSequent,at.
    logic.gapt.proofs.HOLSequent,at.logic.gapt.proofs.expansion.ExpansionProof,at.logic.
    gapt.proofs.expansion.ExpansionProof] = PredicateReductionET |> ErasureReductionET

gapt> val (firstOrderProblem, back) = reduction forward (Sequent() :+ problem)
firstOrderProblem: at.logic.gapt.proofs.HOLSequent =
∀x0 (P_is_nat(x0) ⊃ P_is_nat(f_s(x0))),
⊤ ⊃ P_is_list(f_nil),
∀x0 (P_is_list(x0) ⊃ P_is_o(f_P(x0))),
∀x0 ∀x1 (P_is_nat(x0) ∧ P_is_list(x1) ⊃ P_is_list(f_cons(x0, x1))),
⊤ ⊃ P_is_nat(f_0),
P_is_o(f_nonempty_o),
P_is_nat(f_nonempty_nat),
P_is_list(f_nonempty_list)
⊢
P_P(f_nil) ∧
    ∀x
      (P_is_list(x) ⊃
        ∀y
          (P_is_nat(y) ⊃
            ∀z (P_is_nat(z) ⊃ P_P(x) ⊃ P_P(f_cons(y, f_cons(z, x)))))) ⊃
  P_P(f_cons(f_0, f_cons(f_s(f_0), f_nil)))
back: at.logic.gapt.proofs.expansion.ExpansionProof => at.logic.gapt.proofs.expansion.
    ExpansionProof = <function>

gapt> Prover9 getExpansionProof firstOrderProblem map back
res54: Option[at.logic.gapt.proofs.expansion.ExpansionProof] =
Some(
:-
P(nil:list)- ∧
    (∀x ∀y ∀z (P(x) ⊃ P(cons(y:nat, cons(z:nat, x): list)))
      +^{nil}
        (∀y ∀z (P(nil) ⊃ P(cons(y, cons(z, nil))))
          +^{0}
            (∀z (P(nil) ⊃ P(cons(0, cons(z, nil))))
              +^{s(0)} (P(nil)+ ⊃ P(cons(0, cons(s(0), nil)))-)))) ⊃
  P(cons(0, cons(s(0), nil)))+)
```

4.18 LK to ND translation

GAPT supports translation of a sequent calculus (Appendix B.1) proof without skolem functions, to a natural deduction proof (Appendix B.2).

Consider the following example:

```
gapt> examples.gapticExamples.lemma
res55: at.logic.gapt.proofs.lk.LKProof =
[p7] A ⊃ B ⊢ A ∧ B ∨ ¬ A (OrRightRule(p6, Suc(0), Suc(1)))
[p6] A ⊃ B ⊢ A ∧ B, ¬ A (NegRightRule(p5, Ant(0)))
[p5] A, A ⊃ B ⊢ A ∧ B (ContractionLeftRule(p4, Ant(2), Ant(0)))
[p4] A, A ⊃ B, A ⊢ A ∧ B (AndRightRule(p1, Suc(0), p3, Suc(0)))
[p3] A ⊃ B, A ⊢ B (ImpLeftRule(p1, Suc(0), p2, Ant(0)))
[p2] B ⊢ B  (LogicalAxiom(B:o))
[p1] A ⊢ A  (LogicalAxiom(A:o))

gapt> LKToND( examples.gapticExamples.lemma, Some( Suc( 0 ) ) )
res56: at.logic.gapt.proofs.nd.NDProof =
[p16] A ⊃ B ⊢ A ∧ B ∨ ¬ A (ExcludedMiddleRule(p2, Ant(0), p15, Ant(0)))
[p15] ¬ (A ∧ B), A ⊃ B ⊢ A ∧ B ∨ ¬ A (OrIntro2Rule(p14, A ∧ B))
[p14] ¬ (A ∧ B), A ⊃ B ⊢ ¬ A (NegIntroRule(p13, Ant(1)))
[p13] ¬ (A ∧ B), A, A ⊃ B ⊢ ⊥ (BottomElimRule(p12, ⊥ ))
[p12] ¬ (A ∧ B), A, A ⊃ B ⊢ ⊥ (NegElimRule(p3, p11))
[p11] A, A ⊃ B ⊢ A ∧ B (ContractionRule(p10, Ant(0), Ant(2)))
[p10] A, A ⊃ B, A ⊢ A ∧ B (AndIntroRule(p4, p9))
[p9] A ⊃ B, A ⊢ B (ImpElimRule(p6, p8))
[p8] A ⊃ B, A ⊢ B (ImpElimRule(p7, p4))
[p7] A ⊃ B ⊢ A ⊃ B (LogicalAxiom(A ⊃ B))
[p6] ⊢ B ⊃ B  (ImpIntroRule(p5, Ant(0)))
[p5] B ⊢ B  (LogicalAxiom(B:o))
[p4] A ⊢ A  (LogicalAxiom(A:o))
[p3] ¬ (A ∧ B) ⊢ ¬ (A ∧ B) (LogicalAxiom(¬ (A ∧ B)))
[p2] A ∧ B ⊢ A ∧ B ∨ ¬ A (OrIntro...
```

The LKToND function takes an LK proof, and optionally an Option[SequentIndex], as parameters. Because ND proofs can only contain a single formula in the succedent, the translation must focus on one of the formulas in the succedent of the LK proof that is to be proved in the ND proof. Thus, sometimes formulas need to be exchanged between the antecedents and succedents in the ND proof. This exchange is inherently classical and introduces the excluded middle rule into the proof.

A Lambda calculus

GAPT uses a polymorphic simply-typed lambda calculus to represent formulas and terms. The syntax of types and terms is as follows. A type is either a type function, an arrow (function) type or a type variable.

$$\mathrm{Type} ::= f(\mathrm{Type}, \ldots, \mathrm{Type}) \mid \mathrm{Type} \to \mathrm{Type} \mid ?\alpha$$

There are 4 kinds of expressions: constants, variables, applications, and abstractions:

$$\mathrm{Expr} ::= v : \mathit{Type} \mid c : \mathit{Type} \mid \mathrm{Expr}\,\mathrm{Expr} \mid \lambda(v : \mathit{Type})\,\mathrm{Expr}$$

This lambda calculus is simply typed in the sense that we do not have *quantification* over types. Instead, we allow inductive data types and definitions to be polymorphic. That is, data types and the types of definitions can have type variables. In this manner, we can define a function concat of type `list ?a > list ?a > list ?a`. With this definition we can use all instances, where we substitute ?a for any other type. For example when we use this function for lists of numbers, we would use the instance concat: `list nat > list nat > list nat`.

B Proof systems

B.1 LK

The rules of LK are listed below. Proof trees are constructed top-down, starting with axioms and with each rule introducing new inferences. With the exception of the definition rules, proof links, and induction rules, the constructors of the rules only allow inferences that are actually valid. Note that the rules are presented here as if they always act upon the outermost formulas in the upper sequent, but this is only for convenience of presentation. The basic constructors actually require the user to specify on which concrete formulas the inference should be performed.

Apart from those basic constructors, there is also a multitude of convenience constructors that facilitate easier proof construction. Moreover, there are so-called macro rules that reduce several inferences to a single command (e.g. introducing quantifier blocks). See the API documentation of the individual rules for details.

Axioms

$$\frac{}{A \vdash A}\ \text{(Logical axiom)} \qquad \frac{}{\vdash t = t}\ \text{(Reflexivity axiom)}$$

$$\frac{}{\vdash \top}\ \top\ \text{axiom} \qquad \frac{}{\bot \vdash}\ \bot\ \text{axiom}$$

$$\frac{(t)}{\Gamma \vdash \Delta}\ \text{Proof link}$$

Cut

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

Structural rules

Left rules

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

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

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

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

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

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

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

Right rules

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

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

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

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

Quantifier rules

Left rules

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

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

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

Right rules

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

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

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

The variable y must not occur free in Γ, Δ or A.

Equality rules

Left rules

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

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

Right rules

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

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

Each equation rule replaces an arbitrary number of occurrences of T.

Definition rules

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

These definition rules are extremely liberal, as they allow the replacement of any formula by any other formula. When checking these rule against a context, we verify that both A and B normalize to the same normal form.

Induction

The induction rule applies to arbitrary algebraic data types. Let $c_1, \ldots, c_n$ be the constructors of a type and let k_i be the arity of c_i. Let $F[x]$ be a formula with x a free variable of the appropriate type. Then we call the sequent $\mathcal{S}_i := F[x_1], \ldots, F[x_{k_i}], \Gamma_i \vdash \Delta_i, F[c_i(x_1, \ldots, x_{k_i})]$ the i-th induction step. In this case, the induction rule has the form

$$\frac{\begin{matrix}(\pi_1) & (\pi_2) & & (\pi_n) \\ \mathcal{S}_1 & \mathcal{S}_2 & \ldots & \mathcal{S}_n\end{matrix}}{\Gamma \vdash \Delta, F[t]} \text{ (ind)}$$

In the case of the natural numbers, there are two constructors: 0 of arity 0 and s of arity 1. Consequently, the induction rule reduces to

$$\frac{\begin{matrix}(\pi_1) & (\pi_2) \\ \Gamma_1 \vdash \Delta_1, F[0] & F[x], \Gamma_2 \vdash \Delta_2, F[sx]\end{matrix}}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, F[t]} \text{ (ind)}$$

B.2 ND

The rules of ND are listed below. Classical logic is supported by providing the excluded middle rule. We use ND rules in sequent form.

As in LK, proof trees are constructed top-down, starting with axioms and with each rule introducing new inferences. With exception of the proof links and the induction rules, the constructors of the rules only allow inferences that are actually valid. Note that the rules are presented here as if they always act upon the outermost formulas in the upper sequent, but this is only for convenience of presentation. The basic constructors actually require the user to specify on which concrete formulas the inference should be performed.

Apart from those basic constructors, there is also a multitude of convenience constructors that facilitate easier proof construction. See the API documentation of the individual rules for details.

Axioms

$$\frac{}{A \vdash A}\ \text{(Logical axiom)} \qquad \frac{}{\vdash A}\ \text{(Theory axiom)}$$

Structural rules

$$\frac{\Gamma \vdash B}{A, \Gamma \vdash B}\ \text{(w)} \qquad \frac{A, A, \Gamma \vdash B}{A, \Gamma \vdash B}\ \text{(c)}$$

Propositional rules

Elimination rules

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge\text{:e1})$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge\text{:e2})$$

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

$$\frac{\Gamma \vdash \neg A \qquad \Pi \vdash A}{\Gamma, \Pi \vdash \bot}\ (\neg\text{:e})$$

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

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot\text{:e})$$

Introduction rules

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

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee\text{:i1})$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash B \vee A}\ (\vee\text{:i2})$$

$$\frac{A, \Gamma \vdash \bot}{\Gamma \vdash \neg A}\ (\neg\text{:i})$$

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

$$\frac{}{\vdash \top}\ (\top\text{:i})$$

Quantifier rules

Elimination rules

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

$$\frac{\Gamma \vdash \exists x A \qquad \Pi, A[y/x] \vdash B}{\Gamma, \Pi \vdash B}\ (\exists\text{:e})$$

Introduction rules

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

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

The variable y must not occur free in Γ in case of $\forall$ introduction, and must not occur free in Π or B in case of $\exists$ elimination.

Equality rules

Elimination rules

$$\frac{\Gamma \vdash s = t \qquad \Pi \vdash A[s/x]}{\Gamma, \Pi \vdash A[t/x]}\ (=\text{:e})$$

Introduction rules

$$\frac{}{\vdash t = t}\ (=\text{:i})$$

Induction

The induction rule applies to arbitrary algebraic data types. Let $c_1, \ldots, c_n$ be the constructors of a type and let k_i be the arity of c_i. Let $F[x]$ be a formula with x a free variable of the appropriate type. Then we call the sequent $\mathcal{S}_i := F[x_1], \ldots, F[x_{k_i}], \Gamma_i \vdash F[c_i(x_1, \ldots, x_{k_i})]$ the i-th induction step. In this case, the induction rule has the form

$$\frac{\begin{matrix}(\pi_1) & (\pi_2) & & (\pi_n)\\ \mathcal{S}_1 & \mathcal{S}_2 & \ldots & \mathcal{S}_n\end{matrix}}{\Gamma \vdash F[t]}\ (\text{ind})$$

In the case of the natural numbers, there are two constructors: 0 of arity 0 and s of arity 1. Consequently, the induction rule reduces to

$$\frac{\begin{matrix}(\pi_1) & (\pi_2)\\ \Gamma_1 \vdash F[0] & F[x], \Gamma_2 \vdash F[sx]\end{matrix}}{\Gamma_1, \Gamma_2 \vdash F[t]}\ (\text{ind})$$

Excluded Middle

$$\frac{\Gamma, A \vdash B \qquad \Pi, \neg A \vdash B}{\Gamma, \Pi \vdash B}\ (\text{em})$$

B.3 Resolution

Our resolution calculus integrates higher-order reasoning, structural clausification, and Avatar-style splitting as in [14]. The judgments of this calculus are A-sequents. An A-sequent $S \leftarrow A$ is a pair of a sequent S of HOL formulas, and a conjunction A of propositional literals:

$$\Gamma \vdash \Delta \leftarrow A$$

Internally, we represent the (negation of the) assertion as a clause. The judgment $\Gamma \vdash \Delta \leftarrow A$ is interpreted as the following formula, where $\overline{x}$ are the free variables of the sequent:

$$A \supset \forall \overline{x} \left(\bigwedge \Gamma \supset \bigvee \Delta \right)$$

Inferences such as resolution or paramodulation do not operate on the assertions. Unless specified otherwise, assertions are inherited by default, combined with a conjunction:

$$\frac{\Gamma \vdash \Delta, a \leftarrow A \qquad a, \Pi \vdash \Lambda \leftarrow B}{\Gamma, \Pi \vdash \Delta, \Lambda \leftarrow A \wedge B} \text{ Resolution}$$

There is no factoring on assertions, duplicate assertions are automatically removed. Substitutions are not absorbed into resolution, factoring, and paramodulation; they are explicitly represented using the Subst inference.

Initial sequents

$$\frac{}{S} \text{ Input}$$

$$\frac{}{\vdash t = t} \text{ Refl}$$

$$\frac{}{a \vdash a} \text{ Taut}$$

$$\frac{}{\vdash \forall x \, (D(x) \equiv \varphi[x])} \text{ Defn}$$

Structural rules

$$\frac{a, a, \Gamma \vdash \Delta}{a, \Gamma \vdash \Delta} \text{ Factor} \qquad \frac{\Gamma \vdash \Delta, a, a}{\Gamma \vdash \Delta, a} \text{ Factor}$$

$$\frac{S}{S\sigma} \text{ Subst}$$

Logical rules

$$\frac{\Gamma \vdash \Delta, a \qquad a, \Pi \vdash \Lambda}{\Gamma, \Pi \vdash \Delta, \Lambda}\ \text{Resolution}$$

$$\frac{\Gamma \vdash \Delta, t = s \qquad \Pi \vdash \Lambda, a[t]}{\Gamma, \Pi \vdash \Delta, \Lambda, a[s]}\ \text{Paramod}$$

(We also allow rewriting in the antecedent, and rewriting from right to left.)

$$\frac{\Gamma \vdash \Delta, t = s}{\Gamma \vdash \Delta, s = t}\ \text{Flip} \qquad \frac{t = s, \Gamma \vdash \Delta}{s = t, \Gamma \vdash \Delta}\ \text{Flip}$$

Propositional rules

$$\frac{\top, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}\ \text{TopL} \qquad \frac{\Gamma \vdash \Delta, \bot}{\Gamma \vdash \Delta}\ \text{BottomR}$$

$$\frac{\neg a, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, a}\ \text{NegL} \qquad \frac{\Gamma \vdash \Delta, \neg a}{a, \Gamma \vdash \Delta}\ \text{NegR}$$

$$\frac{a \wedge b, \Gamma \vdash \Delta}{a, b, \Gamma \vdash \Delta}\ \text{AndL} \qquad \frac{\Gamma \vdash \Delta, a \wedge b}{\Gamma \vdash \Delta, a}\ \text{AndR1} \qquad \frac{\Gamma \vdash \Delta, a \wedge b}{\Gamma \vdash \Delta, b}\ \text{AndR2}$$

$$\frac{a \vee b, \Gamma \vdash \Delta}{a, \Gamma \vdash \Delta}\ \text{OrL1} \qquad \frac{a \vee b, \Gamma \vdash \Delta}{b, \Gamma \vdash \Delta}\ \text{OrL2} \qquad \frac{\Gamma \vdash \Delta, a \vee b}{\Gamma \vdash \Delta, a, b}\ \text{OrR}$$

$$\frac{a \supset b, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, a}\ \text{ImpL1} \qquad \frac{a \supset b, \Gamma \vdash \Delta}{b, \Gamma \vdash \Delta}\ \text{ImpL2} \qquad \frac{\Gamma \vdash \Delta, a \supset b}{a, \Gamma \vdash \Delta, b}\ \text{ImpR}$$

$$\frac{\forall x\, \varphi x, \Gamma \vdash \Delta}{\varphi(s(...)), \Gamma \vdash \Delta}\ \text{AllL} \qquad \frac{\Gamma \vdash \Delta, \forall x\, \varphi x}{\Gamma \vdash \Delta, \varphi x}\ \text{AllR}$$

$$\frac{\exists x\, \varphi x, \Gamma \vdash \Delta}{\varphi x, \Gamma \vdash \Delta}\ \text{ExL} \qquad \frac{\Gamma \vdash \Delta, \exists x\, \varphi x}{\Gamma \vdash \Delta, \varphi(s(...))}\ \text{ExR}$$

$$\frac{\varphi(\overline{x}), \Gamma \vdash \Delta}{D(\overline{x}), \Gamma \vdash \Delta}\ \text{DefIntro} \qquad \frac{\Gamma \vdash \Delta, \varphi(\overline{x})}{\Gamma \vdash \Delta, D(\overline{x})}\ \text{DefIntro}$$

Avatar rules

By $[C]$ we denote the propositional atom representing the clause component C.

$$\frac{C, S \leftarrow A}{S \leftarrow A \wedge \neg[C]}\ \text{AvatarSplit}$$

(For simplicity, the AvatarSplit rule only splits away a single clause component at a time.)

$$\frac{}{C \leftarrow [C]}\ \text{AvatarComponent}$$

$$\frac{\Gamma \vdash \Delta\ \leftarrow\ a_1 \wedge a_2 \wedge \cdots \wedge \neg b_1 \wedge \neg b_2 \wedge \cdots}{a_1, a_2, \ldots, \Gamma \vdash \Delta, b_1, b_2, \ldots\ \leftarrow\ \top}\ \text{AvatarContradiction}$$

B.4 Expansion trees

Expansion trees are a compact representation of quantifier inferences in proofs with cuts. They have originally been introduced in [11]. GAPT contains an extension by Skolem nodes, weakening nodes, definitions, merges, and cuts [10].

`ETAtom`	A (where A is a HOL atom)
`ETWeakening`	$\mathsf{wk}(\varphi)$ (where φ is a formula)
`ETMerge`	$E_1 \sqcup E_2$
`ETDefinition`	$D +_{\mathsf{def}} E$ (where D is definitionally equal to the shallow formula of E)
`ETTop`	$\top$
`ETBottom`	$\bot$
`ETNeg`	$\neg E$
`ETAnd`	$E_1 \wedge E_2$
`ETOr`	$E_1 \vee E_2$
`ETImp`	$E_1 \supset E_2$
`ETWeakQuantifier`	$Qx\varphi +^{t_1} \varphi[t_1/x] \cdots +^{t_n} \varphi[t_n/x]$ (where Q is a quantifier and t_i terms)
`ETStrongQuantifier`	$Qx\varphi +^{\alpha}_{\mathsf{ev}} \varphi[\alpha/x]$ (where Q is a quantifier and α an eigenvariable)
`ETSkolemQuantifier`	$Qx\varphi +^{s}_{\mathsf{sk}} \varphi[s/x]$ (where Q is a quantifier and s a Skolem term)
cut	$E_1 \supset E_2$ (where E_1 and E_2 have the same shallow formula)

References

[1] Josep Argelich, Chu Min Li, Felip Manya, and Jordi Planes. The first and second max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):251–278, 2008.

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

[3] Matthias Baaz and Alexander Leitsch. *Methods of Cut-Elimination*, volume 34 of *Trends in Logic*. Springer, 2011.

[4] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: tons of inductive problems. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *International Conference on Intelligent Computer Mathematics (CICM) 2015*, volume 9150 of *Lecture Notes in Computer Science*, pages 333–337. Springer, 2015.

[5] Sebastian Eberhard and Stefan Hetzl. Inductive theorem proving based on tree grammars. *Annals of Pure and Applied Logic*, 166(6):665–700, 2015.

[6] 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.

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

[8] 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.

[9] 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.

[10] Stefan Hetzl and Daniel Weller. Expansion trees with cut. preprint, available at http://arxiv.org/abs/1308.0428, 2013.

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

[12] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2011.

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

[14] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014.