gaptic

Tactic-language for Proof Formalization

Alexander Birch Jensen

February 11, 2016

Technical University of Vienna

1 Introduction

The aim of this report is to provide a brief introduction to the notion of a tactic-language, and to serve as a reference for the current features of the *gaptic* tactic-language. The language is developed as an integrated part of the *GAPT* system[1]. We will not consider how to implement the language. Rather, we will focus on how to use the tactic language for proof search while also describing the underlying logic. As such, this report can partly be considered a user's guide to gaptic. The system is tested towards first-order logic, but is as such developed to also handle higher-order logic.

2 Background

Tactic-languages in the area of proof formalization serve the purpose of providing a meaningful and easy-to-use way of manually searching for a proof. Proof searching in tactic languages is usually a goal directed backwards stepwise construction of the proof as known from proof assistants like *Isabelle* and *Coq*. The main advantage is the fact that the proof is verified by the computer system, such that only valid proofs are found. Inspection of the current goals serve as a critical tool for finding a proof.

Formally, the proofs are conducted in the Gentzen system **LK** fully described in [3, p. 22]. However, the result of applying the tactic for a given rule is closer to the system **G3c** described in [2, p. 77] where weakening and contraction have been absorbed into the rules. For example the $(\wedge : \mathrm{l})$ LK rule splits the sequent in two, but when searching for the proof bottom-up it is often more helpful to carry the entire sequent into each premise such that no information is lost.

$$\dfrac{\dfrac{\Gamma_1 \vdash \Delta_1, C \qquad D, \Gamma_1 \vdash \Delta_1}{\Gamma_1, C \to D \vdash \Delta_1}\,(\to : \mathrm{l}) \qquad \dfrac{}{\varphi \vdash \varphi}\,Ax}{\dfrac{\begin{array}{cc} \vdots & \vdots \\ A, \Gamma_0 \vdash \Delta_0 & B, \Gamma_0 \vdash \Delta_0 \end{array}}{\Gamma_0, A \wedge B \vdash \Delta_0}\,(\wedge : \mathrm{l})}$$

The initial state when starting a proof from scratch in the tactic-language is a labeled sequent that is to be proven. The sequent is labeled in the sense that each formula is denoted by a string that is unique in the sequent. The labels are used when a tactic is to be applied to a specific formula, thus providing an intuitive way to specify which formula to apply it to. At any state during the proof search, the unsolved sub goals are the sequents (that are not axioms) in the leaves of the current proof tree. This is shown by an example in Figure 2. A sub goal is then of course solved when an axiom tactic is successfully applied to it. The proof is considered complete when there are no more sub goals to solve.

As previously mentioned, gaptic is an integrated part of the GAPT system, also developed in Scala. At the current state of implementation, there is no IDE available like known from other popular tools like Isabelle and Coq. Instead, the input is provided as actual Scala code. The details of how to specify the proof input is covered in Section 3. It is however important to note that no knowledge of Scala is required as such, apart from basic acquaintance with the GAPT shell. Furthermore, a complete proof is provided as an `LKProof` object which can be viewed in `prooftool`, further post-processed, etc.

For background knowledge of the underlying proof theory, please refer to [1].

3 Formalizing Proofs with the Tactic-language

The tactic-language package is imported by the command `import at.logic.gapt.proofs.gaptic._`.

We identify a tactic as function that if it succeeds yields a new proof state from the current proof state, with one or more sub goals replaced by new proof segments. We further define a proof state as a proof segment and an ordered list of the sub goals it contains (the sub goals are ordered by the ordering of a left-recursive traversal of the proof tree). It is important to note that a tactic is not guaranteed to succeed, i.e. there may be requirements to the structure of the formula to which it is applied. When nothing else is specified, a tactic is always applied to the left-most sub goal.

On the implementation level, there is a clear distinction between a *tactical* and a *tactic*. The former is more general in the sense that it may replace multiple sub goals, while the latter is a subclass of the former that always replaces one single sub goal.

[1] Please refer to [3] for further information on the GAPT system.

A new proof in the tactic-language is initialized with use of the `Lemma` macro, which must then be provided input in the form of tacticals separated by line breaks (or `;`):

```
gapt> val lemmaEx = Lemma( Sequent( Seq( "a" -> parseFormula("P(a)"), "b" ->
    parseFormula("(all x (P(x) -> Q(x)))") ), Seq( "c" -> parseFormula("Q(a)") ) )
    ) { allL(parseTerm( "a" )) }
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are
    still 1 unproved sub goals:
a: P(a)
b: ∀x.(P(x)→Q(x))
b_0: (P(a)→Q(a))
:-
c: Q(a)
```

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

```
gapt> val lemmaEx = Lemma( Sequent( Seq( "a" -> parseFormula("P(a)"), "b" ->
    parseFormula("(all x (P(x) -> Q(x)))") ), Seq( "c" -> parseFormula("Q(a)") ) )
    ) { allL(parseTerm( "a" )); impL; axiom; axiom }
lemmaEx: at.logic.gapt.proofs.lk.LKProof =
[p4] ∀x.(P(x)→Q(x)), P(a) :- Q(a) (ForallLeftRule(p3, Ant(0), (P(x)→Q(x)), a, x))
[p3] (P(a)→Q(a)), P(a) :- Q(a) (ImpLeftRule(p1, Suc(0), p2, Ant(0)))
[p2] Q(a) :- Q(a) (LogicalAxiom(Q(a)))
[p1] P(a) :- P(a) (LogicalAxiom(P(a)))
```

The following sections contain descriptions and examples of the currently available tactics and features. The examples are mainly toy examples to showcase the given tactic, and may not have any proficient meaning beyond that. For a large scale example please see the lattice proof in the file `examples/lattice/lattice.scala`.

3.1 LK Tactics

The LK tactics cover the most basic tactics in the sense that they correspond closely to the **G3c** rules applied backwards. There are however also weakening rules available.

Weakening

Weakening rules can be quite useful when searching for a proof. As more rules are applied, there may be formulas that are known to longer be necessary. Removing some of the formulas from the sequent may then give a better overview. The tactic `forget` is used for both the left and right weakening rule, and removes the formula from the sequent of the current sub goal that has the label that is provided as input. Since each label in the sequent is unique, it is sufficient to have one tactic for both rules.

```
gapt> val weakEx = Lemma( Sequent( Seq( "a" -> parseFormula("P(a)"), "b" ->
    parseFormula("(all x (P(x) -> Q(x)))") ), Seq( "c" -> parseFormula("Q(a)") ) )
    ) { forget( "b" ) }
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are
    still 1 unproved sub goals:
a: P(a)
:-
c: Q(a)
```

Axioms

The tactics `axiomLog`, `axiomTh`, `axiomRefl`, `axiomBot` and `axiomTop` cover the logical, theory, reflexivity, bottom and top axioms, respectively. The `axiom` 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 `axiom` tactic to end the proof by a logical axiom:

```
gapt> val axiomEx = Lemma( Sequent( Nil, Seq( "D" -> parseFormula( "(exists x (P(x)
     -> (all y P(y))))" ) ) ) ) { exR( parseTerm( "c" ) ); impR; allR; exR(
    parseTerm( "y" ) ); impR; allR; axiom }
```

```
axiomEx: at.logic.gapt.proofs.lk.LKProof =
[p9] :- ∃x.(P(x)→∀y.P(y)) (ContractionRightRule(p8, Suc(0), Suc(1)))
[p8] :- ∃x.(P(x)→∀y.P(y)), ∃x.(P(x)→∀y.P(y)) (ExistsRightRule(p7, Suc(1), (P(x)→∀
    y.P(y)), c, x))
[p7] :- ∃x.(P(x)→∀y.P(y)), (P(c)→∀y.P(y)) (ImpRightRule(p6, Ant(0), Suc(1)))
[p6] P(c) :- ∃x.(P(x)→∀y.P(y)), ∀y.P(y) (WeakeningLeftRule(p5, P(c)))
[p5] :- ∃x.(P(x)→∀y.P(y)), ∀y.P(y) (ForallRightRule(p4, Suc(0), y, y))
[p4] :- P(y), ∃x.(P(x)→∀y.P(y)) (ExistsRightRule(p3, Suc(1), (P(x)→∀y.P(y)), y, x)
    )
[p3] :- P(y), (P(y)→∀yy0.P(y0)) (ImpRightRule(p2, Ant(0), Suc(1)))
[p2] P(y) :- P(y),∀y0.P(y0) (WeakeningRightRule(p1, ∀y0.P(y0)))
[p1] P(y) :- P(y) (LogicalAxiom(P(y)))
```

Definitions

The tactics `defL` and `defR` cover the left and right definition rules. The first argument of both tactics is the label to replace, and the second argument is the replacement formula.

```
gapt> val defEx = Lemma( Sequent( Seq( "c" -> parseFormula( "P(b) | Q(b)" ), "a" ->
     parseFormula( "P(u) -> Q(u)" ) ), Seq( "b" -> parseFormula( "P(x) & Q(x)" ) )
    ) ) { defL( "a", parseFormula( "P(a) -> Q(a)" ) ); defR( "b", parseFormula( "P(
    b) | Q(b)" ) ); axiom }
defEx: at.logic.gapt.proofs.lk.LKProof =
[p3] (P(u)→Q(u)), (P(b)∨Q(b)) :- (P(x)∨Q(x)) (WeakeningLeftRule(p2, (P(u)→Q(u)))
    )
[p2] (P(b)∨Q(b)) :- (P(x)∧Q(x)) (DefinitionRightRule(p1, Suc(0), (P(x)∨Q(x))))
[p1] (P(b)∨Q(b)) :- (P(b)∨Q(b)) (LogicalAxiom((P(b)∨Q(b))))
```

Equality

The tactics `eqL` and `eqR` cover the left and right equality rules. The tactics take as first argument the label of an equality to use from the antecedent. The second argument is the label of the formula to apply the rule to. Furthermore, it can be specified 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 equality will be applied left to right. It that is not applicable, it will try to apply the equality from right to left.

```
gapt> val eqEx = Lemma( Sequent( Seq( "c" -> parseFormula( "P(y) & Q(y)" ), "eq1"
    -> parseFormula( "u = v" ), "eq2" -> parseFormula( "y = x" ), "a" ->
    parseFormula( "P(u) -> Q(u)" ) ), Seq( "b" -> parseFormula( "P(x) & Q(x)" ) ) )
     ) { eqL( "eq1", "a" ).to( parseFormula( "P(v) -> Q(v)" ) ); eqL( "eq1", "a" ).
    to( parseFormula( "P(v) -> Q(u)" ) ); eqR( "eq2", "b" ).fromRightToLeft; axiom
    }
eqEx: at.logic.gapt.proofs.lk.LKProof =
[p5] u=v, (P(u)→Q(u)), y=x, (P(y)∧Q(y)) :- (P(x)∧Q(x)) (WeakeningLeftRule(p4, u=v
    ))
[p4] (P(u)→Q(u)), y=x, (P(y)∧Q(y)) :- (P(x)∧Q(x)) (WeakeningLeftRule(p3, (P(u)→Q
    (u))))
[p3] y=x, (P(y)∧Q(y)) :- (P(x)∧Q(x)) (EqualityRightRule(p2, Ant(0), Suc(0), List
    ([1,2], [2,2])))
[p2] y=x, (P(y)∧Q(y)) :- (P(y)∧Q(y)) (WeakeningLeftRule(p1, y=x))
[p1] (P(y)∧Q(y)) :- (P(y)∧Q(y)) (LogicalAxiom((P(y)∧Q(y))))
```

Quantifiers

The tactics for the weak quantifiers are `allL` and `exR`. They take as first argument always the term to instantiate the quantified formula with. The second argument is an optional label. If not specified, the tactic is applied to the first applicable weak quantifier. The tactics for the strong quantifiers are `allR` and `exL`. The first argument here is an eigenvariable which can be provided optionally, while the second argument is the same as for the weak quantifiers. 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 not.

```
gapt> val quantEx = Lemma( Sequent( Seq( "D" -> parseFormula( "(all x (P(x) & (
    exists y -P(y))))" ) ), Nil ) ) { allL( parseTerm( "c" ) ); andL; exL( FOLVar(
    "y_0" ) ); negL; allL( parseTerm( "y_0" ) ); andL; exL( FOLVar( "y_1" ) ); negL
    ; axiomLog }
quantEx: at.logic.gapt.proofs.lk.LKProof =
[p10] ∀x.(P(x)∧∃y.¬P(y)) :- (ContractionLeftRule(p9, Ant(0), Ant(1)))
[p9] ∀x.(P(x)∧∃y.¬P(y)), ∀x.(P(x)∧∃y.¬P(y)) :- (ForallLeftRule(p8, Ant(0), (P(x)∧
    ∃y.¬P(y)), c, x))
[p8] (P(c)∧∃y.¬P(y)), ∀x.(P(x)∧∃y.¬P(y)) :- (AndLeftRule(p7, Ant(0), Ant(1)))
[p7] P(c), ∃y.¬P(y), ∀x.(P(x)∧∃y.¬P(y)) :- (WeakeningLeftRule(p6, P(c)))
[p6] ∃y.¬P(y), ∀x.(P(x)∧∃y.¬P(y)) :- (ExistsLeftRule(p5, Ant(0), y_0, y))
[p5] ¬P(y_0), ∀x.(P(x)∧∃y.¬P(y)) :- (NegLeftRule(p4, Suc(0)))
[p4] ∀x.(P(x)∧∃y.¬P(y)) :- P(y_0) (ForallLeftRule(p3, Ant(0), (P(x)∧∃y.¬P(y)),
    y_0, x))
[p3] (P(y_0)∧∃y.¬P(y)) :- P(y_0) (AndLeftRule(p2, Ant(1), Ant(0)))
[p2] ∃y.¬P(y), P(y_0) :- P(y_0) (WeakeningLeftRule(p1, ∃y.¬P(y)))
[p1] P(y_0) :- P(y_0) (LogicalAxiom(P(y_0)))
```

Implication, Negation, Disjunction and Conjunction

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. When no label is provided, the first applicable formula is chosen.

```
gapt> val propEx = Lemma( Sequent( Seq( "initAnt" -> parseFormula("A -> B") ), Seq(
     "initSuc" -> parseFormula("(A & B) | -A") ) ) ) { orR( "initSuc" ); negR( "
    initSuc_1" ); andR( "initSuc_0" ); axiom; impL; axiom; axiom }
propEx: 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(0), Ant(2)))
[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))
[p1] A :- A (LogicalAxiom(A))
```

Cut

The `cut` tactic is used to introduce a cut rule. The first argument is the cut formula and the second argument is the (unique) label for the cut formula. In the case where a non-unique label is provided the tactic simply fails.

```
gapt> val cutEx = Lemma( Sequent( Seq( "A" -> parseFormula( "A" ) ), Seq( "C" ->
    parseFormula( "(exists x exists y ( -x=y & f(x)=f(y) ))" ) ) ) ) { cut(
    parseFormula( "I(1)" ), "I1" ); cut( parseFormula( "I(0)" ), "I0" ) }
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are
    still 3 unproved sub goals:
A: A
:-
C: ∃x.∃y.(¬x=y∧f(x)=f(y))
I1: I(1)
I0: I(0)

I0: I(0)
A: A
:-∃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))
```

3.2 Meta Tactics

The meta tactics work on a higher level of abstraction, i.e. by combining existing tactics to form a new tactic.

Proof insertion

The `insert` tactic works as the name suggests by inserting a (partial) proof as a solution to a sub goal. The proof can be inserted if its end sequent subsumes the sub goal. When this is possible, the appropriate substitution is applied to the insertion proof along with a series of weakening rules, such that the insertion proof is correctly merged into the current proof segment to solve the sub goal.

```
gapt> val drinker = Lemma( Sequent( Nil, Seq( "D" -> parseFormula( "(exists x (P(x)
     -> (all y P(y))))" ) ) ) ) { exR( parseTerm( "c" ) ); impR; allR; exR(
    parseTerm( "y" ) ); impR; allR; axiom }
drinker: at.logic.gapt.proofs.lk.LKProof =
[p9] :- ∃x.(P(x)→∀y.P(y)) (ContractionRightRule(p8, Suc(0), Suc(1)))
[p8] :- ∃x.(P(x)→∀y.P(y)), ∃x.(P(x)→∀y.P(y)) (ExistsRightRule(p7, Suc(1), (P(x)→∀
    y.P(y)), c, x))
[p7] :- ∃x.(P(x)→∀y.P(y)), (P(c)→∀y.P(y)) (ImpRightRule(p6, Ant(0), Suc(1)))
[p6] P(c) :- ∃x.(P(x)→∀y.P(y)), ∀y.P(y) (WeakeningLeftRule(p5, P(c)))
[p5] :- ∃x.(P(x)→∀y.P(y)), ∀y.P(y) (ForallRightRule(p4, Suc(0), y, y))
[p4] :- P(y), ∃x.(P(x)→∀y.P(y)) (ExistsRightRule(p3, Suc(1), (P(x)→∀y.P(y)), y, x)
    )
[p3] :- P(y), (P(y)→∀y0.P(y0)) (ImpRightRule(p2, Ant(0), Suc(1)))
[p2] P(y) :- P(y),∀y0.P(y0) (WeakeningRightRule(p1, ∀y0.P(y0)))
[p1] P(y) :- P(y) (LogicalAxiom(P(y)))

gapt> val insertEx = Lemma( Sequent( Nil, Seq( "D" -> parseFormula( "(exists y (P(y
    ) -> (all z P(z))))" ) ) ) ) { insert( drinker ) }
insertEx: at.logic.gapt.proofs.lk.LKProof =
[p9] :- ∃x.(P(x)→∀y.P(y)) (ContractionRightRule(p8, Suc(0), Suc(1)))
[p8] :- ∃x.(P(x)→∀y.P(y)), ∃x.(P(x)→∀y.P(y)) (ExistsRightRule(p7, Suc(1), (P(x)→∀
    y.P(y)), c, x))
[p7] :- ∃x.(P(x)→∀y.P(y)), (P(c)→∀y.P(y)) (ImpRightRule(p6, Ant(0), Suc(1)))
[p6] P(c) :- ∃x.(P(x)→∀y.P(y)), ∀y.P(y) (WeakeningLeftRule(p5, P(c)))
[p5] :- ∃x.(P(x)→∀y.P(y)), ∀y.P(y) (ForallRightRule(p4, Suc(0), y, y))
[p4] :- P(y0), ∃x.(P(x)→∀y.P(y)) (ExistsRightRule(p3, Suc(1), (P(x)→∀y.P(y)), y, x
    ))
[p3] :- P(y0), (P(y0)→∀y1.P(y1)) (ImpRightRule(p2, Ant(0), Suc(1)))
[p2] P(y0) :- P(y0),∀y1.P(y1) (WeakeningRightRule(p1, ∀y1.P(y1)))
[p1] P(y0) :- P(y0) (LogicalAxiom(P(y0)))
```

Repeating a tactical

The `repeat` tactical takes as input a tactical and repeatedly applies it as long as it succeeds. It is generally not required for the input tactical to ever succeed, in which case it has no effect on the proof state.

```
gapt> val repeatEx = Lemma( Sequent( Seq( "initAnt" -> parseFormula("A -> B") ),
    Seq( "initSuc" -> parseFormula("(A & B) | -A") ) ) ) { orR; negR; andR; repeat(
     axiom ); impL; repeat( axiom ) }
repeatEx: 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(0), Ant(2)))
[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))
[p1] A :- A (LogicalAxiom(A))
```

Using an alternative tactical on failure

Tacticals can be combined with the `orElse` method that allows a tactical to try another tactical in case it fails itself. This can even be combined with the `repeat` tactical as seen in the following example:

```
gapt> val orElseEx = Lemma( Sequent( Seq( "initAnt" -> parseFormula("A -> B") ),
    Seq( "initSuc" -> parseFormula("(A & B) | -A") ) ) ) { repeat( orR orElse negR
    orElse andR orElse impL orElse axiom ) }
orElseEx: 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(0), Ant(2)))
[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))
[p1] A :- A (LogicalAxiom(A))
```

3.3 Complex Tactics

The complex tactics provide features that are goal directed and more efficient than simple backwards rule application, i.e. there is one single tactic for decomposing a formula.

Destruct

The `destruct` tactic applies once the applicable LK rule to break down the formula. All the propositional rules are included along with the strong quantifier rules. The weak quantifier rules can of course not be applied without a term as input. It optionally takes a label as input, and is applied to the first *destructible* formula if nothing is specified.

```
gapt> val destructEx = Lemma( Sequent( Seq( "label1" -> parseFormula( "a & (b & c)"
    ) ), Seq( "label2" -> parseFormula( "a | (b | c)" ) ) ) ) { destruct( "label1"
    ); destruct( "label2" ); axiom }
destructEx: at.logic.gapt.proofs.lk.LKProof =
[p5] (a∧(b∧c)) :- (a∨(b∨c)) (AndLeftRule(p4, Ant(1), Ant(0)))
[p4] (b∧c), a :- (a∨(b∨c)) (WeakeningLeftRule(p3, (b∨c)))
[p3] a :- (a∨(b∨c)) (OrRightRule(p2, Suc(0), Suc(1)))
[p2] a :- a, (b∨c) (WeakeningRightRule(p1, (b∨c)))
[p1] a :- a (LogicalAxiom(a))
```

Decompose

The `decompose` tactical has some similarities with `destruct`, but it only applies the unary rules. Furthermore, it repeats itself on the current sub goal as many times as possible. This is also means that it does not take any label input.

```
gapt> val decomposeEx = Lemma( Sequent( Seq( "label1" -> parseFormula( "(all x (p(x
    ) & q(x)))" ) ), Seq( "label2" -> parseFormula( "(all y (p(y) -> (q(y) | r(y)))
    )" ) ) ) ) { decompose; allL( FOLVar( "y" ) ); decompose; axiom }
decomposeEx: at.logic.gapt.proofs.lk.LKProof =
[p9] ∀x.(p(x)∧q(x)) :- ∀y.(p(y)→(q(y)∨r(y))) (ForallRightRule(p8, Suc(0), y, y))
[p8] ∀x.(p(x)∧q(x)) :- (p(y)→(q(y)∨r(y))) (ImpRightRule(p7, Ant(0), Suc(0)))
[p7] p(y), ∀x.(p(x)→q(x)) :- (q(y)∨r(y)) (WeakeningLeftRule(p6, p(y)))
[p6] ∀x.(p(x)→q(x)) :- (q(y)∨r(y)) (OrRightRule(p5, Suc(0), Suc(1)))
[p5] ∀x.(p(x)→q(x)) :- q(y), r(y) (WeakeningRightRule(p4, r(y)))
[p4] ∀x.(p(x)→q(x)) :- q(y) (ForallLeftRule(p3, Ant(0), (p(x)→q(x)), y, x))
[p3] (p(y)→q(y)) :- q(y) (AndLeftRule(p2, Ant(0), Ant(1)))
[p2] p(y), q(y) :- q(y) (WeakeningLeftRule(p1, p(y)))
[p1] q(y) :- q(y) (LogicalAxiom(q(y)))
```

Using solvers

The GAPT system has support for a number of solvers, some of which currently interfaces with the tactic-language. The available solving features are the resolution provers Escargot and Prover9, along with the built-in tableaux prover. The tactics are `escargot`, `prover9` and `prop`, respectively.

```
gapt> val solverEx = Lemma( Sequent( Seq( "initAnt" -> parseFormula("A -> B") ),
    Seq( "initSuc" -> parseFormula("(A & B) | -A") ) ) ) { prop }
solverEx: 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(1)))
[p4] (A→B), A, A :- (A∧B) (ImpLeftRule(p1, Suc(0), p3, Ant(1)))
[p3] A, B :- (A∧B) (AndRightRule(p1, Suc(0), p2, Suc(0)))
[p2] B :- B (LogicalAxiom(B))
[p1] A :- A (LogicalAxiom(A))
```

Backwards chain

A universally quantified implication chain is commonly encountered problem where the sub goal has the following structure:

$$\forall x.\Psi_1 \to \Psi_2 \to \cdots \to \Psi_k \to \varphi,\ \Delta \ \vdash\ \varphi[x := t],\ \Gamma$$

From here we want to have new the sub goals $\forall x.\Psi_1 \to \Psi_2 \to \cdots \to \Psi_k \to \Psi_i[x := t],\ \Delta$ for $i = 1 \ldots k$. This is usually referred to as backwards chaining, and is available through the `chain` tactic. The tactic takes as argument always the label of the universally quantified formula. Furthermore, the label of φ can be specified. In the case that it is not specified, it will by default try the first matching formula. It should be noted that the following equivalent structure is also handled:

$$\forall x.(\Psi_1 \wedge \Psi_2 \wedge \cdots \wedge \Psi_k \to \varphi),\ \Delta \ \vdash\ \varphi[x := t],\ \Gamma$$

```
gapt> val chainEx = Lemma( Sequent( Seq( "a" -> parseFormula( "r(f(c))" ), "b" ->
    parseFormula( "q(f(c))" ), "c" -> parseFormula( "w(f(c))" ), "hyp" ->
    parseFormula( "(all x ((r(x) & q(x) & w(x)) -> p(f(x))))" ) ), Seq( "target" ->
     parseFormula( "p(f(f(c)))" ) ) ) ) { chain( "hyp" ); }
at.logic.gapt.proofs.gaptic.QedFailureException: Proof not completed. There are
    still 3 unproved sub goals:
a: r(f(c))
b: q(f(c))
c: w(f(c))
hyp: ∀x.((r(x)∧(q(x)∧w(x)))→p(f(x)))
:-
hyp_0: r(f(c))

a: r(f(c))
b: q(f(c))
c: w(f(c))
hyp: ∀x.((r(x)∧(q(x)∧w(x)))→p(f(x)))
:-
hyp_0: q(f(c))

a: r(f(c))
b: q(f(c))
c: w(f(c))
hyp: ∀x.((r(x)∧(q(x)∧w(x)))→p(f(x)))
:-
hyp_0: w(f(c))

gapt> val chainEx2 = Lemma( Sequent( Seq( "hyp" -> parseFormula( "(all x (p(f(x))))
    " ) ), Seq( "target" -> parseFormula( "p(f(f(c)))" ) ) ) ) { chain( "hyp" ) }
[p2] ∀x.p(f(x)) :- p(f(f(c))) (ForallLeftRule(p1, Ant(0), p(f(x)), f(c), x))
[p1] p(f(f(c))) :- p(f(f(c))) (LogicalAxiom(p(f(f(c)))))
```

4 Further Work

The amount of possible features of a tactic-language is numerous, and we therefore only seek to cover a handful of possible future developments.

Forward chain

As described in Section 3.3, the `chain` tactic currently only supports backwards chaining, and will simply fail in the case of a forward chain. Consider as an example a sub goal with the following sequent:

$$\forall x.(q(x) \to p(f(x))),\ q(c),\ \Delta \ \vdash\ \Gamma$$

From here we want to reach the following sequent as a sub goal by use of forward chaining:

$$\forall x.(q(x) \to p(f(x))),\ p(f(c)),\ \Delta \ \vdash\ \Gamma$$

The pattern is in many ways like that of backwards chaining, and it is as such a natural extension to the `chain` tactic.

Higher-order logic

As briefly mentioned, the tactic-language is developed to support higher-order logic, but the features are only tested on first-order logic formalizations. Therefore, it is to be expected to require an extensive amount of testing and and possibly some modifications to fully support higher-order logic as well.

User-defined tactics

The tactics `destruct` and `decompose` are examples of tactics that are definable through the available meta tactics. As such, combinations of the available tactics combined with the meta tactics can be used to form new tactics that streamlines the proof formalization. Since the needed tactics will differ from different proof contexts, the possibility for the user to define new tactics for the given formalization would prove particularly useful. One obvious approach is to allow the user to define a new tactic as a combination of existing ones. While this is clearly limited by the available built-in features, it does however ensure that any user-defined tactic works correctly. Another approach is to allow the user to define tactics more freely. That is, to allow the sub goal to be replaced by any fitting proof segment. It must be fitting in the sense that the sequent at the root of the proof segment must still be equal to the sequent of the sub goal that it replaces. The latter approach of course has the obvious drawback that more work is required to provide such a feature, since it would require an advanced sub-language for defining tactics within the current tactic-language.

References

[1] Buss, Samuel R., *Handbook of proof theory*, Chapter I, 1998.

[2] Troelstra, A. S. and Schwichtenberg H., *Basíc Proof Theory*, 2000.

[3] Theory And Logic Group, GAPT: General Architecture for Proof Theory, User Manual, `http://www.logic.at/gapt/downloads/gapt-user-manual.pdf` Technical University of Vienna.