Contents

Chapter 1

Package logic.proof.builder.proof

Package Contents *Page*

Classes

1.1 Classes

1.1.1 Class **Proof**

Stores all data necessary to construct a proof. Simple Methods are provided to manipulate the proof such as adding or deleting lines.

Declaration

public class Proof
extends java.lang.Object

Fields

- public List predicates
  - A list of all the named predicates in the proof. Used to populate the predicate list.

Constructors

- *Proof*
  `public` **Proof**`( )`
  - **Usage**
    - Default constructor. Constructs an empty proof

Methods

- *addStepAsEndOfSubproof*
  `public ProofStep` **addStepAsEndOfSubproof**`( logic.proof.builder.parser.SimpleNode` **node**`, java.lang.String` **formula** `)`
  - **Usage**
    - Adds a new proofstep that is the last line of a subproof
  - **Parameters**
    - `node` - Root node of the sentence of the proofstep
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *addStepAsNewLine*
  `public ProofStep` **addStepAsNewLine**`( logic.proof.builder.parser.SimpleNode` **node**`, java.lang.String` **formula** `)`
  - **Usage**
    - The default method to add a new proofstep to the proof

    - `node` - Root node of the sentence of the proofstep
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *addStepAsStartOfSubproof*
  `public ProofStep` **addStepAsStartOfSubproof**`(` `logic.proof.builder.parser.SimpleNode` **node**`,` `java.lang.String` **formula** `)`
  - **Usage**
    - Adds a new proofstep that is the start of a subproof
  - **Parameters**
    - `node` - Root node of the sentence of the proofstep
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *addVar*
  `public ProofStep` **addVar**`(` `java.lang.String` **var** `)`
  - **Usage**
    - Add a new proofstep which introduces a boxed variable
  - **Parameters**
    - `var` - The name of the variable being introduced
  - **Returns** - Returns the proofstep that has been added

- *addVar*
  `public ProofStep` **addVar**`(` `java.lang.String` **introducedVariable**`,` `logic.proof.builder.parser.SimpleNode` **rootNode**`,` `java.lang.String` **formula** `)`
  - **Usage**
    - Add a new proofstep which introduces a boxed variable alongside an assumption
  - **Parameters**
    - `introducedVariable` - The name of the variable being introduced
    - `node` - Root node of the sentence
    - `formula` - String representation of the sentence
  - **Returns** - Returns the proofstep that has been added

- *getCurrentLevel*
  `public int` **getCurrentLevel**`( )`
  - **Usage**
    - Returns the number of subproofs currently open
  - **Returns** - the number of subproofs currently open

- *getLines*
  `public ArrayList` **getLines**`( )`
  - **Usage**
    - Returns the ordered list of proofsteps
  - **Returns** - the ordered list of proofsteps

- *removeStep*
  `public void` **removeStep**`( )`
  - **Usage**
    - Removes the most recent line from the proof

1.1.2 Class **ProofStep**

Declaration

public class ProofStep
extends java.lang.Object

Fields

- public ProofStep parent
  –
- public List subproofs
  –
- public ProofStep next
  –
- public SimpleNode node
  –
- public Integer lineNumber
  –
- public int level
  –
- public String formula
  –
- public String justification
  –
- public boolean endOfSubproof
  –
- public HashMap freeVariables
  –
- public String introducedVariable

1.1.3 Class **RulesOfInference**

Contains methods for the rules of inference of first-order logic

Declaration

public final class RulesOfInference
extends java.lang.Object

Constructors

- *RulesOfInference*
  `public` **RulesOfInference**`( )`

Methods

- *andElimination1*
  `public static SimpleNode` **andElimination1**`(`
  `logic.proof.builder.parser.SimpleNode` **premise** `)`
  - **Parameters**
    - `premise` - The root node of a conjunctive sentence of FOL
  - **Returns** - leftChild The left child node of the given conjunction
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

- *andElimination1*
  `public static void` **andElimination1**`( logic.proof.builder.parser.SimpleNode` **premise**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `premise` - The root node of a conjunction
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

- *andElimination2*
  `public static SimpleNode` **andElimination2**`(`
  `logic.proof.builder.parser.SimpleNode` **premise** `)`
  - **Parameters**
    - `premise` - The root node of a conjunction

- **Exceptions**
  - `logic.proof.builder.exceptions.PremiseException` - If conclusion does not follow from using rule on given arguments

---

- *andElimination2*

  `public static void` **andElimination2**`( logic.proof.builder.parser.SimpleNode` **premise**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `premise` - The root node of a conjunction
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *andIntroduction*

  `public static SimpleNode` **andIntroduction**`( logic.proof.builder.parser.SimpleNode` **p**`, logic.proof.builder.parser.SimpleNode` **q** `)`

  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `q` - The root node of a sentence of FOL
  - **Returns** - conjunction The root node of a conjunction of the given parameters

---

- *andIntroduction*

  `public static void` **andIntroduction**`( logic.proof.builder.parser.SimpleNode` **p**`, logic.proof.builder.parser.SimpleNode` **q**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `q` - The root node of a sentence of FOL
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *bottomElimination*

  `public static void` **bottomElimination**`( logic.proof.builder.parser.SimpleNode` **p**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `premise` - Bottom only
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *compareEqualsElim*
  public static boolean **compareEqualsElim**(logic.proof.builder.parser.SimpleNode **a**, logic.proof.builder.parser.SimpleNode **b**, logic.proof.builder.parser.Variable **subVariable**, java.lang.String **newName**)

- *copy*
  public static boolean **copy**(logic.proof.builder.parser.SimpleNode **p**, logic.proof.builder.parser.SimpleNode **conclusion**)
  - **Parameters**
    - `premise` - The root node of a sentence of FOL
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

- *doubleNegationElimination*
  public static SimpleNode **doubleNegationElimination**(logic.proof.builder.parser.SimpleNode **p**)
  - **Parameters**
    - `p` - The root node of a sentence of FOL starting with two negations
  - **Returns** - The premise without the first two negations
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

- *doubleNegationElimination*
  public static void **doubleNegationElimination**(logic.proof.builder.parser.SimpleNode **p**, logic.proof.builder.parser.SimpleNode **conclusion**)
  - **Parameters**
    - `premise` - The root node of a sentence of FOL starting with two negations
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

- *doubleNegationIntroduction*
  public static SimpleNode **doubleNegationIntroduction**(logic.proof.builder.parser.SimpleNode **p**)
  - **Parameters**
    - `p` - The root node of a sentence of FOL
  - **Returns** - the premise with two negations appended to the start

- *doubleNegationIntroduction*
  `public static void` **doubleNegationIntroduction**`(`
  `logic.proof.builder.parser.SimpleNode` **p**`,`
  `logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `conclusion` - The root node of the sentence being justified, should start with double negation
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

- *equalsElimination*
  `public static void` **equalsElimination**`( logic.proof.builder.parser.SimpleNode` **equals**`, logic.proof.builder.parser.SimpleNode` **statement**`,`
  `logic.proof.builder.parser.Variable` **variable**`,`
  `logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `equals` - A sentence of the form t1 = t2
    - `statement` - A sentence of FOL, should contain t1
    - `variable` - The free variable t1
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

- *equalsIntroduction*
  `public static void` **equalsIntroduction**`( logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `conclusion` - The root node of the sentence being justified, must have the form t = t
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

- *existsElimination*
  `public static void` **existsElimination**`( logic.proof.builder.parser.SimpleNode` **p**`, java.util.ArrayList` **subproof**`, java.lang.String` **variableName**`,`
  `logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `p` - The root node of a sentence of FOL, should be existentially quantified
    - `subproof` - a Subproof starting with a sentence in which the existentially quantified variable of the premise is named and ends in any sentence of FOL
    - `variableName` - The name of the variable that is introduced

  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *existsIntroduction*
  `public static void` **existsIntroduction**`( logic.proof.builder.parser.SimpleNode` **p**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `premise` - The root node of a sentence of FOL
    - `conclusion` - The root node of the sentence being justified, should begin with an existential quantifer
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *forAllElimination*
  `public static void` **forAllElimination**`( logic.proof.builder.parser.SimpleNode` **forAll**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `forAll` - The root node of universally quantified sentence of FOL
    - `conclusion` - The root node of the sentence being justified, should be the unquantified version of the premise
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *forAllIntroduction*
  `public static void` **forAllIntroduction**`( logic.proof.builder.parser.SimpleNode` **p**`, logic.proof.builder.parser.Variable` **variable**`, logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `p` - The final line of the subproof, should contain the introduced variable
    - `variable` - The introduced variable
    - `conclusion` - The universally quantified version of the premise
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *impliesIntroduction*
  `public static SimpleNode` **impliesIntroduction**`( java.util.List` **subproof** `)`

  - **Parameters**
    - `subproof` - Any subproof
  - **Returns** - an implication where the LHS is the first line of the given subproof and the

---

- *impliesIntroduction*
  `public static void` **impliesIntroduction**`( java.util.List` **subproof**, `logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `premise` - The root node of a sentence of FOL
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *modusPonens*
  `public static SimpleNode` **modusPonens**`(` `logic.proof.builder.parser.SimpleNode` **p**, `logic.proof.builder.parser.SimpleNode` **implication** `)`

  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `implication` - The root node of a material implication sentence. The LHS should be the previous argument.
  - **Returns** - the RHS of the implication
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *modusPonens*
  `public static void` **modusPonens**`( logic.proof.builder.parser.SimpleNode` **p**, `logic.proof.builder.parser.SimpleNode` **implication**, `logic.proof.builder.parser.SimpleNode` **conclusion** `)`

  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `implication` - The root node of a material implication sentence. The LHS should be the previous argument.
    - `conclusion` - The root node of the sentence being justified
  - **Exceptions**
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *negationElimination*
  `public static SimpleNode` **negationElimination**`(` `logic.proof.builder.parser.SimpleNode` **p**, `logic.proof.builder.parser.SimpleNode` **notP** `)`

  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `notP` - The negation of the previous argument

- **Exceptions**
  - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *negationElimination*
  `public static void` **negationElimination**`(`
  `logic.proof.builder.parser.SimpleNode` **p**,
  `logic.proof.builder.parser.SimpleNode` **notP**,
  `logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `p` - The root node of a sentence of FOL
    - `notP` - The negation of the previous argument
    - `conclusion` - The root node of the sentence being justified, should only be bottom
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *negationIntroduction*
  `public static SimpleNode` **negationIntroduction**`( java.util.List` **subproof** `)`
  - **Parameters**
    - `subproof` - A list of proofsteps ending with bottom
  - **Returns** - the negation of the first line of the subproof
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule

---

- *negationIntroduction*
  `public static void` **negationIntroduction**`( java.util.List` **subproof**,
  `logic.proof.builder.parser.SimpleNode` **conclusion** `)`
  - **Parameters**
    - `premise` - The root node of a sentence of FOL
    - `conclusion` - The root node of the sentence being justified, should start with a negation
  - **Exceptions**
    - `logic.proof.builder.exceptions.PremiseException` - If premises are not of the correct form for this rule
    - `logic.proof.builder.exceptions.ConclusionException` - If conclusion does not follow from using rule on given arguments

---

- *orElimination*
  `public static SimpleNode` **orElimination**`( java.util.List` **subproof1**,
  `java.util.List` **subproof2**, `logic.proof.builder.parser.SimpleNode`
  disjunction `)`
  - **Parameters**
    - `subproof1` - A subproof starting with the LHS of the disjunction

  - subproof2 - A subproof starting with the RHS of the disjunction and ending with the same sentence as the previous subproof
  - disjunction - The root node of a disjunction of sentences
- **Returns** - chi The root node of the sentence that both subprrofs end with
- **Exceptions**
  - logic.proof.builder.exceptions.PremiseException - If premises are not of the correct form for this rule

- *orElimination*

  public static void **orElimination**(java.util.List **subproof1**, java.util.List **subproof2**, logic.proof.builder.parser.SimpleNode **disjunction**, logic.proof.builder.parser.SimpleNode **conclusion**)

  - **Parameters**
    - subproof1 - A list of
    - subproof2 - The root node of the sentence being justified
    - disjunction - The root node of a disjunction of sentences
    - conclusion - The root node of the sentence being justified
  - **Exceptions**
    - logic.proof.builder.exceptions.ConclusionException - If conclusion does not follow from using rule on given arguments
    - logic.proof.builder.exceptions.PremiseException - If premises are not of the correct form for this rule

- *orIntroduction1*

  public static void **orIntroduction1**(logic.proof.builder.parser.SimpleNode **premise**, logic.proof.builder.parser.SimpleNode **conclusion**)

  - **Parameters**
    - premise - The root node of a sentence of FOL
    - conclusion - The root node of the sentence being justified, must be a disjunctive sentence
  - **Exceptions**
    - logic.proof.builder.exceptions.ConclusionException - If conclusion does not follow from using rule on given arguments
    - logic.proof.builder.exceptions.PremiseException - If premises are not of the correct form for this rule

- *orIntroduction2*

  public static void **orIntroduction2**(logic.proof.builder.parser.SimpleNode **premise**, logic.proof.builder.parser.SimpleNode **conclusion**)

  - **Parameters**
    - premise - The root node of a sentence of FOL
    - conclusion - The root node of the sentence being justified, must be a disjunctive sentence
  - **Exceptions**
    - logic.proof.builder.exceptions.ConclusionException - If conclusion does not follow from using rule on given arguments
    - logic.proof.builder.exceptions.PremiseException - If premises are not of the correct form for this rule