Bachelor Thesis

Proofs in Equational Logic

Suat Coşkun
info@suatcoskun.com

14 October 2012

Supervisor: Dr. Harald Zankl

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Datum	Unterschrift

Abstract

Equational logic can be seen as predicate logic where the only predicate symbol is equality.
The first aim of this bachelor project is the formulation of a calculus for equational logic. The second aim is to design an input-format for proofs and the third aim is to develop a program which transforms the given sequences of equalities with respect to the input-format to LaTeX proofs in this calculus.
The used programming language for transforming is java.

Acknowledgments

Thanks to Allah that I believe in his existence but cannot prove his existence. Thanks to my supervisor Dr. Harald Zankl for guiding me during this project. Thanks to all people.

Contents

1 Introduction

A logic has a certain language in which formulas of the logic can be formulated. The language is usually given by an inductive definition, involving one or more kinds of variables, connectives, quantifiers, etc. There are different kinds of logics e.g., classical propositional logic, intuitionistic propositional logic, classical first order logic, intuitionistic first order logic, first order logic with equality, other logics like minimal logic, linear logic, modal logics, horn logic, rewrite logic etc., and finally *equational logic.*

Equational logic is a sub logic of classical first order logic, expresses equalities between terms that may refer to individual variables. Equational logic expresses simple equational reasoning, terms are still composed of function symbols and variables, formulas are always equations of the form $s \approx t$ when s and t are terms.
The main aims of this report are to

1. develop an easy text format where equational logic proofs are easy to typeset
2. develop transformations to LaTeX code such that equational logic proofs are nicely displayed.

Typesetting such proofs directly in LaTeX is time consuming and error-prone. Therefore there was a need for this project. The basic but main questions about this bachelor project can be seen as follows:

1. Why to do this project?
    - Because it is more cumbersome to write the proofs in LaTeX than writing in any plain-text file.
2. What are the advantages of this project?
    - The simplicity of writing of the proof in a simple text format.
    - Display the same proof in various formats e.g., linear-format, tree-format.
3. Where to use this project?
    - At universities in lectures

In this project we have tree keywords such as *given, transformed, displayed.*

- A proof is *given.*
- The given proof is *transformed.*
- The transformed proof is *displayed.*

Given is for example the proof in Figure 1.1 below in a file. The program, which has been developed in this thesis, transforms the given proof in LaTeX code in Figure 1.2 and Figure 1.3 .

Example 1.1. : Given is the proof below.

$$1; a = b; prem$$
$$2; b = c; prem$$
$$3; a = c; trans 1, 2$$

Figure 1.1: proof

Displayed in linear – view

1	$a \approx b$	prem
2	$b \approx c$	prem
3	$a \approx c$	trans 1, 2

Figure 1.2: proof_linear.pdf

Displayed in tree – view

$$\dfrac{\dfrac{}{a \approx b}\,(app) \quad \dfrac{}{b \approx c}\,(app)}{a \approx c}\,(trans)$$

Figure 1.3: proof_tree.pdf

Above we have the different representations of the same proof. The left one is in linear-shape while the right one is in tree-shape. The written program helps to typeset in LaTeX. This project allows easy typesetting in simple text format. We write the proof into a file according to the input-format. And the proof in this file is transformed automatically into different LaTeX sources for nice display.

I would like to give an overview of structure of thesis. In the next chapter, i.e., Chapter 2 we recall equational logic. In the Chapter 3 we present the input-format used for typesetting equational logic proofs as plain-text files. In the Chapter 4 and Chapter 5 we denote to transformations into LaTeX source code. In Chapter 6 we present a user manual before Chapter 7 concludes.

2 Equational Logic

2.1 Definition of Logic and Equational Logic

This page was taken mostly from the lecture script by Xiemaisi, University of Sinica in Taiwan [9].

Definition 2.1. What is a Logic?

- A logic has a certain language in which formulas of the logic can be formulated.
- The language is usually given by an inductive definition, involving one or more kinds of variables, connectives, quantifiers, etc.
- Formulas of the language have some sort of intended meaning.

Examples of Logics

- Classical propositional logic (CPL): formulas express true or false propositions.
- Intuitionistic propositional logic (IPL): same language as CPL, formulas express abstract problems or statements to be proved.
- Classical first order logic (CFOL): same intended meaning as CPL, more expressive formula language with quantification over individuals.
- Intuitionistic first order logic (IFOL): relates to IPL like CFOL to CPL
- Equational logic (EL): a sub logic of classical first order logic, expresses equalities between terms that may refer to individual variables.
- First order logic with equality (FOLE): a variety of CFOL/IFOL that gives special status to the equality symbol.
- Other logics: minimal logic, linear logic and its varieties, modal logics, temporal logic, Horn logic, rewrite logic, ...

Definition 2.2. What is Equational Logic?

Equational logic consists of terms and the symbol equality which should be proved whether the both sides are equal or not equal.

- Equational Logic expresses simple equational reasoning.
- Terms are still composed of function symbols and variables.

- Formulas are always equations of the form $s \approx t$ (no connectives, no quantifiers).
- We can view equational logic either from a semantic or a proof theoretic point of view.

2.2 Terms

This section and the sections Section 2.3 and Section 2.5 were taken from the lecture script of Term Rewriting by Prof. Aart Middeldorp from the University of Innsbruck [5].

In this section we discuss the syntax of the terms.

Definition 2.3. A *signature* is a set $\mathcal{F}$ of *function symbols.* Associated with every $f \in \mathcal{F}$ is a natural number denoting its *arity,* i.e., the number of arguments it is supposed to have. A function symbol of arity n is called n-ary. We use unary for 1-ary, binary for 2-ary, and ternary for 3-ary function symbols. Function symbols of arity 0 are called *constants.*

Definition 2.4. Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a countably infinite set of *variables* disjoint from $\mathcal{F}$. The set $\mathcal{T}(\mathcal{F},\mathcal{V})$ of terms built from $\mathcal{F}$ and $\mathcal{V}$ is the smallest set such that every variable is a term, every constant is a term, and if $f \in \mathcal{F}$ is a function symbol of arity $n > 0$ and $t_1,\dots,t_n$ are terms then $f(t_1,\dots,t_n)$ is a term.

Function symbols of arity greater than 0 are typically denoted by f, g, h, constants by a, b, c variables by x, y, z and terms by s, t, u (and their derivatives, like s' and t_1).

2.3 Algebras

In the previous section we introduced the syntax of the terms. In this section we are concerned with their semantics.

Definition 2.5. Let $\mathcal{F}$ be a signature. An $\mathcal{F}-$*algebra* $\mathcal{A}$ is a set A equipped with operations $f_{\mathcal{A}} : A^n \to A$ for every $n-$ary function symbol $f \in \mathcal{F}$. The underlying set A is called the *carrier* of $\mathcal{A}$.

Consider a signature $\mathcal{F}$ consisting of a constant 0, a unary function symbol s, and a binary function symbol $+$. The set $\mathbb{N}$ of natural numbers can be turned into an $\mathcal{F}$-algebra $\mathcal{A}$ by defining $0_{\mathcal{A}} = 0$, $s_{\mathcal{A}}$ is the successor function, and $+_{\mathcal{A}}$ is addition. The carrier of the $\mathcal{F}$-algebra $\mathcal{B}$ is the set $\{\oplus,\ominus,\otimes,\oslash\}$.

Constant 0 is interpreted $\oplus$ and the interpretation $s_{\mathcal{B}}$ and $+_{\mathcal{B}}$ of the function symbols s and $+$ are given in the following tables (the first argument of $t_{\mathcal{B}}$ is written in the left column and the second argument in the top row):

$s_{\mathcal{B}}$	
$\oplus$	$\ominus$
$\ominus$	$\otimes$
$\otimes$	$\oslash$
$\oslash$	$\oplus$

Table 2.1: Interpretation of $s_{\mathcal{B}}$

$+_{\mathcal{B}}$	$\oplus$	$\ominus$	$\otimes$	$\oslash$
$\oplus$	$\oplus$	$\ominus$	$\otimes$	$\oslash$
$\ominus$	$\ominus$	$\ominus$	$\oplus$	$\oplus$
$\otimes$	$\otimes$	$\oslash$	$\otimes$	$\ominus$
$\oslash$	$\oslash$	$\oplus$	$\otimes$	$\oslash$

Table 2.2: Interpretation of $+_{\mathcal{B}}$

These two $\mathcal{F}$-algebras will be used to illustrate subsequent developments. In the following we make the notational convention that A (B, C, …) denotes the carrier of the $\mathcal{F}$-algebra A (B, C, …). Furthermore, if the signature $\mathcal{F}$ can be inferred from the context or is irrelevant, we often write algebra instead of $\mathcal{F}$-algebra.

Let $\mathcal{A}$ be an arbitrary algebra. Every ground term t can be interpreted in $\mathcal{A}$ by simply replacing every function symbol f in t by its interpretation $f_{\mathcal{A}}$ and evaluating the resulting expression. For instance, in the algebra $\mathcal{A}$ defined above the ground term $s(0+s(0))$ is interpreted as $s_{\mathcal{A}}(0_{\mathcal{A}} +_{\mathcal{A}} s_{\mathcal{A}}(0_{\mathcal{A}})) = 2$. Its interpretation in the algebra $\mathcal{B}$ is $s_{\mathcal{B}}(0_{\mathcal{B}} +_{\mathcal{B}} s_{\mathcal{B}}(0_{\mathcal{B}})) = \otimes$. This is formalized below.

Definition 2.6. Let $\mathcal{A}$ be an arbitrary algebra. We inductively define a mapping $[.]_{\mathcal{A}}$ from the set of ground terms to A as follows:
$[f(t_1,\ldots,t_n)]_{\mathcal{A}} = f_{\mathcal{A}}([t_1]_{\mathcal{A}},\ldots,[t_n]_{\mathcal{A}})$. In particular, if t is a constant then $[t]_{\mathcal{A}} = t_{\mathcal{A}}$.

So $[s(0+s(0))]_{\mathcal{A}} = 2$ and $[s(0+s(0)]_{\mathcal{B}} = \otimes$ in the example algebras. The interpretation of non-ground terms depends on the values that we assign to the variables. Consider for instance the terms $s(x+s(y))$ and the example algebra $\mathcal{A}$. If we assign 2 to x and 3 to y the we get $s_{\mathcal{A}}(2 +_{\mathcal{A}} s_{\mathcal{A}}(3))$, which evaluates to 7. Assigning the value 49 to both x and y results in $s_{\mathcal{A}}(49 +_{\mathcal{A}} s_{\mathcal{A}}(49)) = 100$.

2.4 Rules for Equational Logic

2.4.1 Reflexivity

The reflexivity rule

$$\frac{}{t \approx t}\ (ref)$$

For all terms t reflexivity allows us to prove two identical terms equivalent.

Example 2.7. : t is equal t .

$$\frac{}{t \approx t}\,(ref)$$

Figure 2.1: Tree-view

1	$t \approx t$	ref

Figure 2.2: Linear-view

2.4.2 Symmetry

The symmetry rule

$$\frac{s \approx t}{t \approx s}\,(sym)$$

Symmetry allows us to conclude the equation in reverse order, we conclude the equation below the inference-line by the equation above the inference-line.

Example 2.8. : Given is $a \approx b$ and prove $b \approx a$.

$$\cfrac{\cfrac{}{a \approx b}\,(app)}{b \approx a}\,(sym)$$

Figure 2.3: Tree-view

1	$a \approx b$	prem
2	$b \approx a$	sym 1

Figure 2.4: Linear-view

2.4.3 Transitivity

The transitivity rule

$$\frac{s \approx t,\ \ t \approx u}{s \approx u}\,(trans)$$

Transitivity allows us to prove $s \approx u$ if s is equivalent to t and t is equivalent to u.

Example 2.9. : Given are $a \approx b$ and $b \approx c$, prove $a \approx c$.

$$\cfrac{\cfrac{}{a \approx b}\,(app) \quad \cfrac{}{b \approx c}\,(app)}{a \approx c}\,(trans)$$

Figure 2.5: Tree-view

1	$a \approx b$	prem
2	$b \approx c$	prem
3	$a \approx c$	trans 1, 2

Figure 2.6: Linear-view

2.4.4 Application

The application rule

$$\frac{}{l\sigma \approx r\sigma}\,(app)$$

For all $l \approx r \in \mathcal{E}$ and substitutions σ we can prove any instance of an equation.

Example 2.10. : Given is $f(x) + 0 \approx f(x)$, so prove $f(1) + 0 \approx f(1)$.

$$\frac{}{f(1) + 0 \approx f(1)}\,(app)$$

Figure 2.7: Tree-view

1	$f(x) + 0 \approx f(x)$	prem
2	$f(1) + 0 \approx f(1)$	app 1

Figure 2.8: Linear-view

2.4.5 Congruence

The congruence rule

$$\frac{s_1 \approx t_1, \ \ldots, \ s_n \approx t_n}{f(s_1, \ \ldots, \ s_n) \approx f(t_1, \ \ldots, \ t_n)} \ (cong)$$

For all $n-$ary function symbols f, we can extend terms with any function symbols.

Example 2.11. : Given is $a \approx b$, prove $f(a) \approx f(b)$.

$$\dfrac{\dfrac{}{a \approx b}\ (app)}{f(a) \approx f(b)}\ (cong)$$

Figure 2.9: Tree-view

1	$a \approx b$	prem
2	$f(a) \approx f(b)$	cong 1

Figure 2.10: Linear-view

Example 2.12. : Given are $a \approx b$ and $b \approx c$ prove $f(a,b) \approx f(b,c)$.

$$\dfrac{\dfrac{}{a \approx b}\ (app) \quad \dfrac{}{b \approx c}\ (app)}{f(a,b) \approx f(b,c)}\ (cong)$$

Figure 2.11: Tree-view

1	$a \approx b$	prem
2	$b \approx c$	prem
3	$f(a,b) \approx f(b,c)$	cong 1, 2

Figure 2.12: Linear-view

Example 2.13. : Given are $a \approx b$, $b \approx c$ and $c \approx d$, prove $f(a,b,c) \approx f(b,c,d)$.

$$\dfrac{\dfrac{}{a \approx b}\ (app) \quad \dfrac{}{b \approx c}\ (app) \quad \dfrac{}{c \approx d}\ (app)}{f(a,b,c) \approx f(b,c,d)}\ (cong)$$

Figure 2.13: Tree-view

1	$a \approx b$	prem
2	$b \approx c$	prem
3	$c \approx d$	prem
4	$f(a,b,c) \approx f(b,c,d)$	cong 1, 2, 3

Figure 2.14: Linear-view

2.4.6 Summary of Rules

The rules below state that if the formula above the line is a theorem formally deduced from axioms by application of the syllogisms, then the formula below the line is also a formal theorem. Usually, some finite set E of identities is given as axiom schemata.

Equational logic can be combined with first-order logic. In this case, the congruence rule is extended onto predicate symbols as well, and the application rule is omitted. Syllogisms can be turned into axiom schemata having the form of implications to which Modus Ponens can be applied. Major results of first-order logic hold in this extended theory.

Equational logic is complete, if algebra A is a model for E, i.e., all identities from E hold in algebra A (cf. universal algebra) , then $s \approx t$ holds in A iff it can be deduced in the equational logic defined by E. This theorem is sometimes known as Birkhoff's theorem [6].

Reflexivity	$\dfrac{}{t \approx t}$(ref)	for all terms t
Symmetry	$\dfrac{s \approx t}{t \approx s}$(sym)	
Transitivity	$\dfrac{s \approx t,\ t \approx u}{s \approx u}$(trans)	
Application	$\dfrac{}{l\sigma \approx r\sigma}$(app)	for all $l \approx r \in \mathcal{E}$ and substitutions σ
Congruence	$\dfrac{s_1 \approx t_1,\ \dots,\ s_n \approx t_n}{f(s_1,\ \dots,\ s_n) \approx f(t_1,\ \dots,\ t_n)}$(cong)	for all n−ary function symbols f

Table 2.3: Summary of inference rules

2.5 Equational Reasoning

Definition 2.14. An *equation* is a pair (s, t) of terms, written as $s \approx t$.

Equations are interpreted by comparing the meaning of the two constituent terms. Consider for instance the example algebras $\mathcal{A}$ and $\mathcal{B}$ defined in Section 2.3 and the equation $s(0) + s(0) \approx s(s(0 + 0))$. Both terms have the same value 2 in $\mathcal{A}$. We say the equation is *valid* in $\mathcal{A}$. Then equation $s(0) + s(0) \approx s(s(0 + 0))$ is not valid in $\mathcal{B}$ because the values of the two terms differ: $[s(0) + s(0)]_{\mathcal{B}} = \ominus$ and $[s(s(0 + 0))]_{\mathcal{B}} = \otimes$. To determine the validity of equations involving non-ground terms, we have to take all possible values for the variables into consideration.

Definition 2.15. An equation $s \approx t$ is valid in an algebra $\mathcal{A}$, denoted by $\mathcal{A} \vDash s \approx t$ or $s =_{\mathcal{A}} t$, if $[\alpha]_{\mathcal{A}}(s) = [\alpha]_{\mathcal{A}}(t)$ for every assignment $\alpha \in \mathcal{A}^{\mathcal{V}}$. We also say that $\mathcal{A}$ is a model of the equation $s \approx t$.

The equation $x+x \approx x$ is not valid in $\mathcal{A}$ because $[\alpha]_{\mathcal{A}}(x+x) = 2 \neq 1 = [\alpha]_{\mathcal{A}}(x)$ for any assignment α satisfying $\alpha(x) = 1$. Because $[\beta]_{\mathcal{B}}(x + x) = [\beta]_{\mathcal{B}}(x)$ for every assignment β from $\mathcal{V}$ to $\{\oplus, \ominus, \otimes, \oslash\}$, the equation $x + x \approx x$ is valid in $\mathcal{B}$.

Definition 2.16. An *equational system* (ES for short) is a pair $(\mathcal{F}, \mathcal{E})$ consisting of a signature $\mathcal{F}$ and a set $\mathcal{E}$ of equations between term in $\mathcal{T}(\mathcal{F}, \mathcal{E})$.

Both $\mathcal{F}$ and $\mathcal{E}$ may be infinite. An ES $(\mathcal{F}, \mathcal{E})$ is said to be *finite* if both $\mathcal{F}$ and $\mathcal{E}$ are finite. We often present an ES as a set of equations, without making explicit its signature, assuming that the signature consists of the function symbols occurring in the equations. In the next definition we give semantics to ESs.

Definition 2.17. An algebra $\mathcal{A}$ is a *model* of an ES $\mathcal{E}$, denoted by $\mathcal{A} \vDash \mathcal{E}$, if every equation in $\mathcal{E}$ is valid in $\mathcal{A}$. The *variety* $\mathcal{M}od(\mathcal{E})$ of an ES $\mathcal{E}$ in the class of all algebras that are models of $\mathcal{E}$. We write $\mathcal{E} \vDash s \approx t$ or simply $s \approx_{\mathcal{E}} t$ if $s \approx t$ is valid in all models of $\mathcal{E}$.

Consider the ES $\mathcal{E} = \{0 + x \approx x,\ s(x) + y \approx s(x + y)\}$. The equation $s(x) + y \approx s(x+y)$ is valid in the example of algebra $\mathcal{A}$ since for all natural numbers m and n we have $(m+1)+n = (m+n)+1$. Since the equation $0 + x \approx x$ is also valid in $\mathcal{A}(0 + n = n$ for all natural numbers $n)$, $\mathcal{A}$ is a model of the ES $\mathcal{E}$. The equation $s(x) + y \approx s(x + y)$ is not valid in the example algebra $\mathcal{B}:\ s_{\mathcal{B}}(\oplus) +_{\mathcal{B}} \ominus = \ominus \neq \otimes = s_{\mathcal{B}}(\oplus +_{\mathcal{B}} \ominus)$. Hence $\mathcal{B} \notin \mathcal{M}od(\mathcal{E})$.

Definition 2.18. Let $\mathcal{E}$ be an ES. We write $\mathcal{E} \vdash s \approx t$ or simply $s \approx_{\mathcal{E}} t$ if the equation $s \approx t$ derivable from the inference rules of Table 2.3 .

The following *proof tree* shows that $s(s(0)+s(0)) \approx_{\mathcal{E}} s(s(s(0)))$ with respect to the example ES $\mathcal{E}$ above:

$$\dfrac{\dfrac{\dfrac{}{s(0)+s(0) \approx s(0+s(0))}\,(\mathit{app}) \qquad \dfrac{\dfrac{}{0+s(0) \approx s(0)}\,(\mathit{app})}{s(0+s(0)) \approx s(s(0))}\,(\mathit{cong})}{s(0)+s(0) \approx s(s(0))}\,(\mathit{trans})}{s(s(0)+s(0)) \approx s(s(s(0)))}\,(\mathit{cong})$$

It turns out that the equation $s(s(0) + s(0)) \approx_{\mathcal{E}} s(s(s(0)))$ is valid in the example algebras $\mathcal{A}$ and $\mathcal{B}$: $[s(s(0) + s(0))]_{\mathcal{A}} = 3 = [s(s(s(0)))]_{\mathcal{A}}$ and $[s(s(0) + s(0))]_{\mathcal{B}} = \otimes = [s(s(s(0)))]_{\mathcal{B}}$. This is not a coincidence. Focus on contribution of this thesis that equational reasoning is *sound*. This means that every equation $s \approx t$ deducible from an ES $\mathcal{E}$ by equational reasoning is valid in all models of $\mathcal{E}$.

Theorem 2.19. *Let $\mathcal{E}$ be an ES and s and t be terms. If $s \approx_{\mathcal{E}} t$ then $s \approx_{\mathcal{E}} t$.*

Proof. An easy induction on the structure of the proof tree of $s \approx t$.

The final result of this section, which is known as the *completness* of equational reasoning, states that the reverse of the Theorem 2.19 also holds.

Corollary 2.20. *Let $\mathcal{E}$ be an ES. The relations $=_{\mathcal{E}}$ and $\approx_{\mathcal{E}}$ coincide.*

$((S.x).y).z \approx (x.z).(y.z)$
$(K.x).y \approx x$
$I.x \approx x$

Table 2.4: Combinatory logic.

Definition 2.21. The *equational theory* of an *ES* $\mathcal{E}$ is the set of all equations $s \approx t$ such that $s \approx_{\mathcal{E}} t$. The *validity problem* for a given *ES* $\mathcal{E}$ is the question whether an arbitrary equation $s \approx t$ belongs to the equational theory of $\mathcal{E}$. If we only consider equations $s \approx t$ between ground terms s and t then we speak of the *word problem* for $\mathcal{E}$.

Rephrased in syntactical terms, the validity problem for $\mathcal{E}$ is the question whether the relation $\approx_{\mathcal{E}}$ is computable and the word problem for $\mathcal{E}$ amounts to the computability of the restriction of $\approx_{\mathcal{E}}$ to ground terms. The word and validity problem are undecidable in general. A concrete example of an *ES* with undecidable word problem (and hence undecidable validity problem) is *combinatory logic*, presented in Table 2.4 . The signature of combinatory logic consists of the tree constants S, K, and I, and a binary function symbol ., called *application*. So there is no algorithm that decides, given two ground term s and t, whether s and t can be proved equal using the equations of the Table 2.4 .

3 Input-Format of the Proof

In order to transform a proof we need a given file which is written according to the syntax below. A valid-input-format corresponds to the regular expression as defined in the example below.

Example 3.1. : Regular expression of a valid input-format.

```
\d+;.+=.+;(cong\d+(,\d+)*|trans\d+,\d+|sym\d+|app\d+|ref|prem)
```

1. A line in the file has three parts and these parts are separated with semicolons e.g., part1; part2; part3
2. The first part expresses the index number of the line and is also used for referencing earlier proof steps in the next lines.
3. The second part is the deduced equality and ends with semicolon.
4. Finally the third part is the applied rule and contains the number or numbers of the line(s) which we applied the rule to. And the most important thing is here arguments are separated with a comma.
5. White-spaces in or between any of these parts will be ignored. Blank-lines will be also ignored.

The following examples show well-formed input proofs.

3.1 Valid Input-Format

Example 3.2. : An example for a valid input-format which doesn't contain any whites-spaces and blank-line.

```
1;b=a;prem
2;c=b;prem
3;c=d;prem
4;d=x;prem
5;x=y;prem
6;y=f;prem
7;a=b;sym1
8;b=c;sym2
9;a=c;trans7,8
10;a=d;trans9,3
11;x=f;trans5,6
12;d=f;trans4,11
13;g(a,d)=g(d,f);cong10,12
14;g(d,f)=g(a,d);sym13
15;h(g(d,f))=h(g(a,d));cong14
```

Example 3.3. : Another example for a valid input-format is the example below which contains white-spaces and blank-line. This example of the proof contains white-spaces and blank-line while the other Example 3.2 doesn't contains.

```
 1; a =       b;prem




2;                           b = c;prem
3; c = d;            prem
4; d= e;prem

         5; e= f;prem
6; f= g;prem

8; h(a,b,c,d,e,f) = h(b,c,d,e,f,g);cong    1,2 , 3, 4,5,6
```

3.2 Invalid Input-Format

The program automatically discovers the syntax issues where you type wrong or forget to type the line-number, the semicolons, the symbol equality, the abbreviations of the rules, the arguments of the rules, and the comma(s) between the arguments. Recall the regular expression which we defined in Example 3.1. We can read the valid-input-format of a proof by that example.

Example 3.4. : An example for invalid input-format is the example below. In this case we have a file whose content contains the wrong typesetting.

```
1    a = b;prem


    2 ; c =         d;   premise

 3; b = c    prem
     4 ; f(b) =         f(c);   cong
 5; a c ;    trans1,3
       ; c =         a;  5
   7; f(a)=f(c);
   8; f(c)=f(a); sym
   9; f(b)=f(a); trans48
   10; f(a)=f(b);sym 9
```

A correct line begins with a number followed by a semicolon and then some character followed by the equality symbol and again some character followed by a semicolon then the abbreviation of the rules. If the rule is a rule which can have a number or numbers as its arguments, these have also to write after the abbreviation of these rules. In case more than one argument they are separated with a comma.

The first line “$1a = b; prem$” . Here we forgot to write the first semicolon. After correction has the structure as “$1; a = b; prem$” .

The second line “$2; c = d; premise$” . Here we didn't write the abbreviation of the rule premise. This looks after correction as “$2; c = d; prem$” .
So one can see the whole corrected-format on the right-hand-side in the table below. The corrected character or characters are displayed in the color of darkblue.

Line	Wrong-format	Corrected-format
1	1a=b;prem	1;a=b;prem
2	2;c=d;premise	2;c=d;prem
3	3;b=cprem	3;b=c;prem
4	4;f(b)=f(c);cong	4;f(b)=f(c);cong3
5	5;ac;trans1,3	5;a=c;trans1,3
6	;c=a;5	6;c=a;sym5
7	7;f(a)=f(c);	7;f(a)=f(c);cong 5
8	8;f(c)=f(a);sym	8;f(c)=f(a);sym 7
9	9;f(b)=f(a);trans48	9;f(b)=f(a);trans4,8

Table 3.1: The corrected input-format

4 Proof Tree

Proofs in tree-view use the package proof.sty [8]. The proof-tree may have many nodes and these nodes may have again many others nodes as its children. If a node has only a child then we call it as *unary*-node, with two children *binary*-node and with $n-$children n-ary node. If a node has no child then it is a leaf expect the application rule.

The child corresponds to the argument of the rule. A node with one child means a rule with one argument, a node with two children means a rule with two arguments, and a node with n-children means a rule with n-arguments e.g., "sym 1" means both a node with one child/argument and a *unary*-node and "cong 1,2,3" means both a node with tree children/arguments and a *ternary*-node. The rules reflexivity, premise are leaf, application is also a leaf although it has a child.

Next we describe an algorithm which transforms proofs in the format from Chapter 3 into proofs with respect to proof.sty.

4.1 Algorithm

The algorithm of proving in tree-view took most of time of this thesis. At first I implemented the program according to the first algorithm as in the next section. But in this algorithm we had the restriction that the congruence rule may have at most two arguments. Then I improved the first algorithm in Section 4.1.1 so that the congruence rule may have many argument.

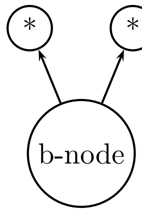
4.1.1 The first Algorithm: The rule congruence has at most 2 arguments

The congruence rule causes that a node may be a *ternary*-node or n-ary node. In this algorithm we have the restriction such that the rule congruence can have nodes which are a *unary*-node and *binary*-node.

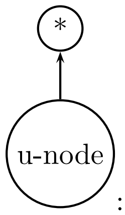
We distinguish between two trees. The first one is the basic tree which has only a *unary*-node while the other one is the complex tree which may have any n-ary-node but in the first algorithm *binary*-node and in the second algorithm to the 9-ary-node.

- The basic tree: A tree whose nodes have only one child/ *unary*-node.
- The complex tree: A tree whose nodes may have many children i.e., *unary*-nodes, *binary*-nodes.

Abbreviations: unary-node: u-node, binary-node: b-node, ternary-node: t-node, n-ary-node: n-node.



Binary-node : Every binary-node corresponds either to the rule of transitivity or congruence with two arguments.



Unary-node : Every unary-node corresponds one of to the rules symmetry or congruence with only one argument. Reflexivity, premise and application are leafs. Application is a leaf although it has a child.
Any arbitrary node of the complex tree has four cases below that we distinguish.

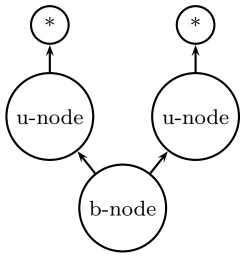


Figure 4.1: The children of a binary-node are unary-nodes

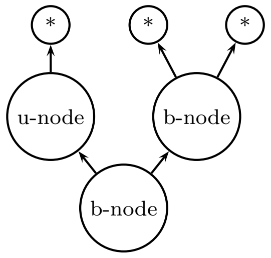


Figure 4.2: The right hand-side of the b-node is a b-node while the left hand-side is a u-node

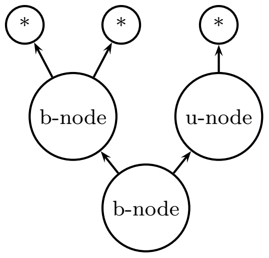


Figure 4.3: The left hand-side of a b-node is a b-node while the right hand-side is a u-node

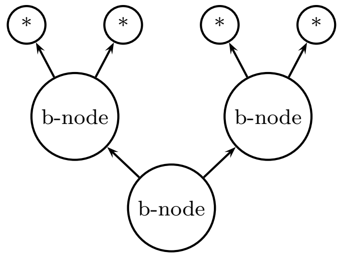


Figure 4.4: The both children are b-node

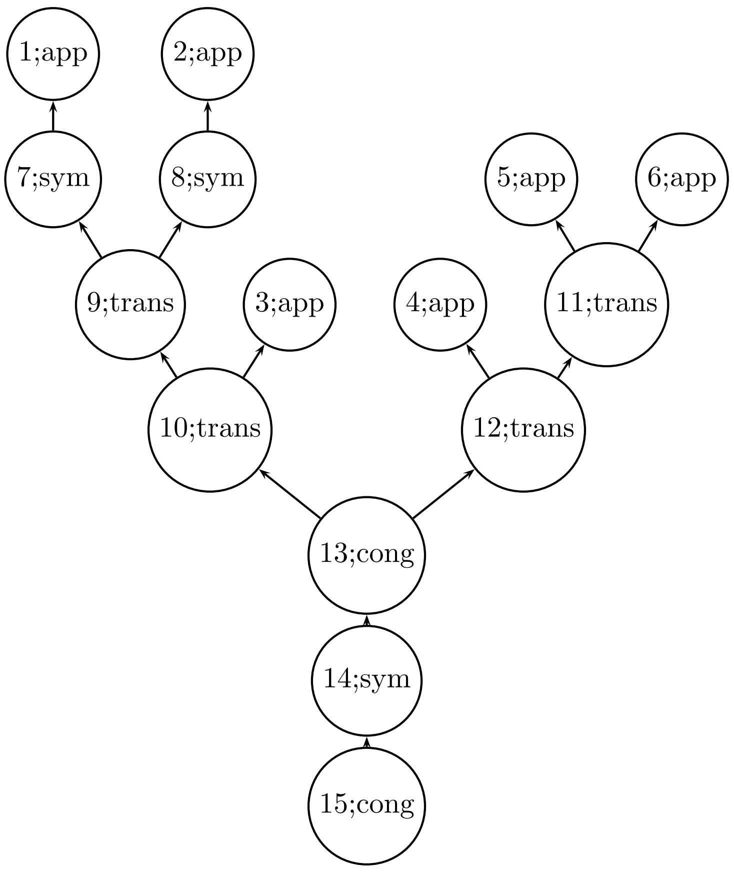


Figure 4.5: A tree with respect to the package proof.sty

The tree in the Figure 4.5 is the tree of the Example 3.2 . In the tree we clearly see the behaviours of the rules. In every node you can see the proof-line-number and the name of the rule, e.g., 15;cong means the line 15 and the applied rule congruence.

The basic tree: a tree whose nodes have only a child.
There is no binary-node in the basic tree. The lines in the given (see Figure 4.6) proof have the same color in the tree (see Figure 4.7) . Thus we can imagine the tree structure of this proof.

The first step is reading of the given file. After reading the file we add the whole content of the given file in a list. Then we check if the given file corresponds to the basic tree or to the complex tree. If it were a the basic tree, it is very easy to transform it. We begin here to transform from the last line backwards, and it is displayed as in the Figure 4.8 .

Example 4.1. : Given were a proof below in Figure 4.6 which only contains *unary*-nodes.

$1; b = a; prem$

$2; a = b; sym1$

$3; f(a) = f(b); cong2$

Figure 4.6: Given proof

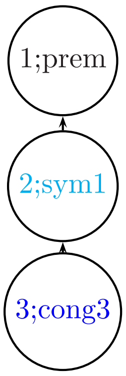


Figure 4.7: Given proof's tree

$$\dfrac{\dfrac{\dfrac{}{b \approx a}\,(app)}{a \approx b}\,(sym)}{f(a) \approx f(b)}\,(cong)$$

Figure 4.8: Proof's tree-view

The complex tree: a tree whose nodes may have many children i.e., *binary*-node, *ternary*-node till $9{-}ary$-node. $9{-}ary$-node means a node with 9 children.

First we read the given proof file (see Section Valid-Input-Format Example 3.2) and add its content in a list. Then we check if the given file corresponds to the complex tree. If this were the case, we filter all *binary*-nodes in the list which we already added the content of the given file in. After filtering we add the filtered nodes in another list (see Figure 4.9) . And this list contains only *binary*-nodes.

1. We take the first *binary*-node i.e., $9; a = c; trans7, 8$ (see Figure 4.11) in the list and transform it with its left child and continuing the right child. The first *binary*-node doesn't contain any *binary*-node in its block (see Figure 4.11) and (see Figure 4.13) . We start transforming with it left child after finishing with the left child continuing with the right child. So we are done with this *binary*−node and add the transformed TEX code. in its map. Its map has as key its line-number and as content the its TEX code.

2. The next *binary*-node is $10; a = d; trans9, 3$ (see Figure 4.12) . We start with the left child and meet the *binary*-node and stop transforming steps. *Because its child is a binary-node and must have been already transformed before.* And we add the TEX code its child i.e., its *binary*-node in the actual map and we are don with left child. Now we transform the right child which is in this are the *unary*-node application(app). So by doing this we are done with this actual *binary*-node. i.e., 10;a=d;trans9,3 .

3. The next is $11; x = f; trans5, 6$ (see Figure 4.13) , this step is like the first step.

4. The next is $12; d = f; trans4, 11$ (see Figure 4.14) , this step is like the second step.

5. And finally $13; f(a, d) = f(d, f); cong10, 12$ (see Figure 4.10) , the last *binary*-node. This forth case its both children are *binary*-node. Hence we only add the TEX code of its children in its map. This is our root-*binary*-node. And this *binary*-node has all information i.e., TEX code of all nodes in its block.

6. We are still not done. This root-*binary*-node is not the last one in the list hence we start from the last one in the list and transform till this node. And their TEX code are the completed code of the given proof. Now we are done.

$$9; a{=}\ c; trans7, 8$$
$$10; a{=}\ d; trans9, 3$$
$$11; x{=}\ f; trans5, 6$$
$$12; d{=}\ f; trans4, 11$$
$$13; g(a, d){=}\ g(d, f); cong10, 12$$

Figure 4.9: After filtering to *binary*-node

Above we have the filtered proof, it only contains the rule transitivity and congruence with two arguments.

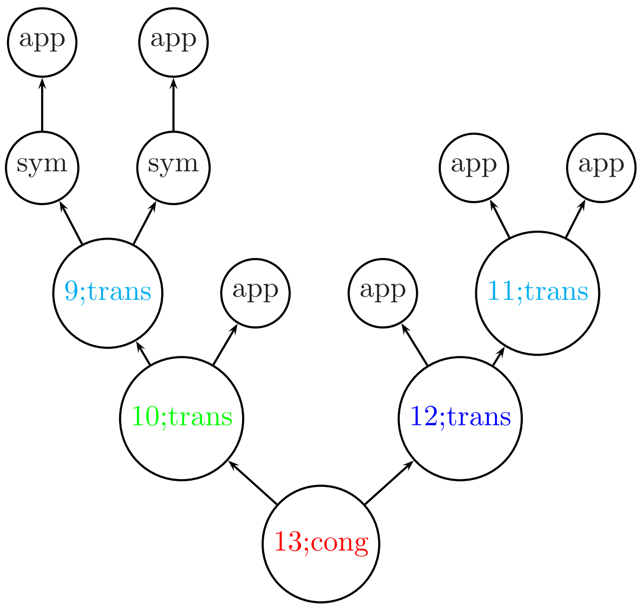


Figure 4.10: The filtered proof's tree

Above we have the filtered proof's tree. It corresponds also to the case (see Figure 4.4)

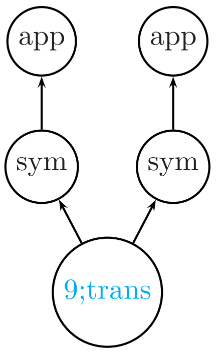


Figure 4.11: The childen of *binary*-node are *unary*-node

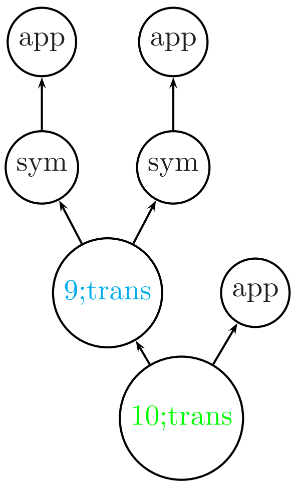


Figure 4.12: The left child of *binary*-node is again a *binary*-node while the right one is *unary*-node

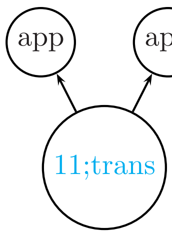


Figure 4.13: The childen of *binary*-node are *unary*-node

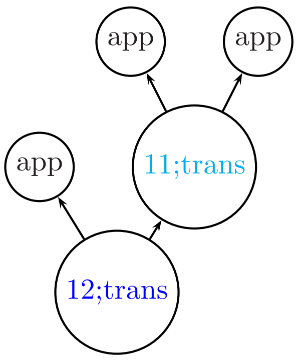


Figure 4.14: The right child is a *binary*-node while the left one is a *unary*-node

However, this algorithm is not default in the implementation because its restriction to at most two children, hence it was disabled in the file TreeView.java. If you want to use this algorithm you have to set the variable switchAlgorithm on false. For changing to default algorithm set the variable switchAlgorithm on true.

4.1.2 The second Algorithm: The rule congruence may have at most 9 arguments

This algorithm is based largely on the algorithm which we mentioned before. The second algorithm has not the four cases (see page 16) which cause that the rule congruence may not have more than two arguments. Instead of the four cases we have here a loop which repeats the number of the arguments of the rule congruence. The *unary*-nodes of the $n-ary$-node are added iteratively in the loop. The $n-ary$-node and *unary*-node have the same meaning as before. The basic tree is the same and the complex tree is the same but it can contain here till 9 children i.e., $9-ary$-node.

Here we filter all nodes from *binary*-node till $9-ary$-node.

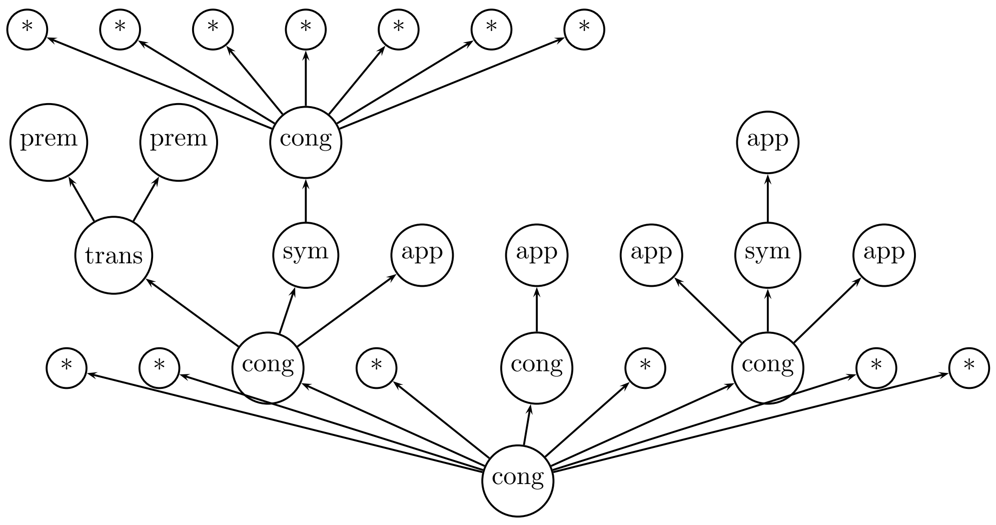


Figure 4.15: The tree structure of a proof which contains a $9 - ary$-node

The structure of a proof in tree-view is like a tree above. The tree above has a node which we call as a $9 - ary$-node. The $9 - ary$-node is the root-node of the tree. And this $9 - ary$-node may have any node like *unary*-node, *binary*-node, *ternary*-node, . . . , $8 - ary$-node or again a $9 - ary$-node. This root node contains all information i.e., source-codes.

However, this algorithm is the default in the implementation. If you want to change the other algorithm, you have to set the variable switchAlgorithm on false in the file TreeView.java. For resetting you have to set the variable switchAlgorithm on true.

Details of the implementation:
In tree-view we have the restriction that the rule congruence (see congruence 2.4.5) doesn't have more than nine arguments because if we select "\tiny" for font size and the both sides of the equality have only a character which is minimum still fits an A4-sized page (see Figure 4.19) . Hence the arguments of the congruence are limited to nine.

Example 4.2. : We recall the example on the left-hand-side which we mentioned before as Example 3.2 let's draws its tree (see Figure 4.16) and show how it looks in tree-view (see Figure 4.17) .

$1; b = a; prem$
$2; c = b; prem$
$3; c = d; prem$
$4; d = x; prem$
$5; x = y; prem$
$6; y = f; prem$
$7; a = b; sym1$
$8; b = c; sym2$
$9; a = c; trans7, 8$
$10; a = d; trans9, 3$
$11; x = f; trans5, 6$
$12; d = f; trans4, 11$
$13; g(a, d) = g(d, f); cong10, 12$
$14; g(d, f) = g(a, d); sym13$
$15; h(g(d, f)) = h(g(a, d)); cong14$

In this example we have the given proof above. The *binary*-nodes are colored in different colors. Hence the uncolored nodes are *unary*-nodes. On the right-hand-side we have proof's tree structure.

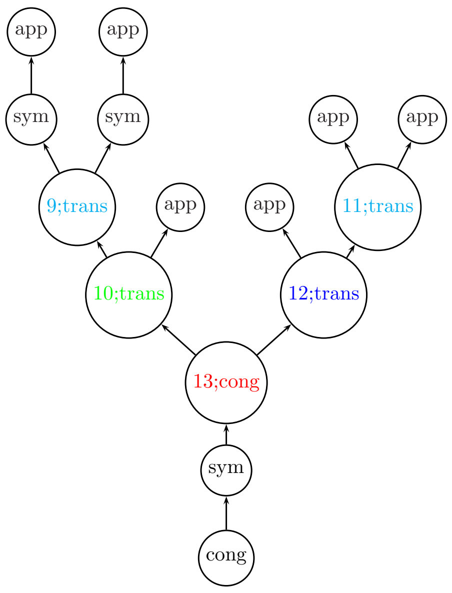


Figure 4.16: The tree of the Example 4.2

Here is the drawn tree of the proof on the left-hand-side. The same color corresponds to the same proof-line in the proof i.e., 13;cong expresses 13;g(a,d)=g(d,f);cong10,12

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{b\approx a}(app)}{a\approx b}(sym)\quad \dfrac{\dfrac{}{c\approx b}(app)}{b\approx c}(sym)}{a\approx c}(trans)\quad \dfrac{}{c\approx d}(app)}{a\approx d}(trans) \quad \dfrac{\dfrac{}{d\approx x}(app)\quad \dfrac{\dfrac{}{x\approx y}(app)\quad\dfrac{}{y\approx f}(app)}{x\approx f}(trans)}{d\approx f}(trans)}{g(a,d)\approx g(d,f)}(cong)}{g(d,f)\approx g(a,d)}(sym)}{h(g(d,f))\approx h(g(a,d))}(cong)$$

Figure 4.17: Tree-view of the given Example 4.2

And above in the Figure 4.17 we see the tree-view of the given file above in Example 4.2 .

Example 4.3. : Let's see another example from the lecture script of Term Rewriting by Prof. Aart Middeldorp [5]. Below we see the proof and in the Figure 4.18 how it displays in tree-view.

$$\begin{aligned}
&1; s(x)+s(y)=s(x+s(y)); prem\\
&2; 0+s(y)=s(y); prem\\
&3; s(0)+s(0)=s(0+s(0)); app1\\
&4; 0+s(0)=s(0); app2\\
&5; s(0+s(0))=s(s(0)); cong4\\
&6; s(0)+s(0)=s(s(0)); trans3,5\\
&7; s(s(0)+s(0))=s(s(s(0))); cong6
\end{aligned}$$

Above we have the given proof, and below its tree-view with respect to the package proof.sty.

$$\cfrac{\cfrac{\cfrac{}{s(0)+s(0)\approx s(0+s(0))}\,(app) \qquad \cfrac{\cfrac{}{0+s(0)\approx s(0)}\,(app)}{s(0+s(0))\approx s(s(0))}\,(cong)}{s(0)+s(0)\approx s(s(0))}\,(trans)}{s(s(0)+s(0))\approx s(s(s(0)))}\,(cong)$$

Figure 4.18: Tree-view of the given Example 4.3

Example 4.4. : An example for our boundary value, the rule congruence has 9 arguments, and we select "\tiny" for font-size and every term consisting of only a character, so it fits the page.

$$\begin{aligned}
&1; a=b; prem\\
&2; b=c; prem\\
&3; c=d; prem\\
&4; d=e; prem\\
&5; e=f; prem\\
&6; f=g; prem\\
&7; g=h; prem\\
&8; h=i; prem\\
&9; i=j; prem\\
&10; s(a,b,c,d,e,f,g,h,i)=s(b,c,d,e,f,g,h,i,j); cong1,2,3,4,5,6,7,8,9
\end{aligned}$$

Above we have the given proof, and below its tree-view in Figure 4.19 with respect to the package proof.sty.

$$\dfrac{\dfrac{}{a \approx b}\,(app) \quad \dfrac{}{b \approx c}\,(app) \quad \dfrac{}{c \approx d}\,(app) \quad \dfrac{}{d \approx e}\,(app) \quad \dfrac{}{e \approx f}\,(app) \quad \dfrac{}{f \approx g}\,(app) \quad \dfrac{}{g \approx h}\,(app) \quad \dfrac{}{h \approx i}\,(app) \quad \dfrac{}{i \approx j}\,(app)}{s(a,b,c,d,e,f,g,h,i) \approx s(b,c,d,e,f,g,h,i,j)}\,(cong)$$

Figure 4.19: Tree-view of the given Example 4.4

Properties of my algorithm

- The algorithm based on the strategy divide and conquer. I divide the tree in $n - ary$-node where n is in the range $[2-9]$. And then I conquer these filtered nodes.
- The algorithm works top-to-bottom. It is to transform the nodes in the top of the tree so by the way their parent-nodes already transformed if the parent-node has these two n-ary nodes as its children.
- The complexity is linear.
- *It is to transform at most one node once or no one.*

4.2 Implementation

The package which I chose for tree-view is proof.sty [8] written by Prof. Makoto Tatsuta, National Institute of Informatics from Japan.[1]
The strategy which is used here is bottom-to-top which means we begin with the last line of given proof and solve it until we reach a leaf.
The other strategy which I tried before was top-to-bottom strategy uses the package bussproofs.sty [2] written by Prof. Samuel R. Buss.[2] The reason why I changed to the package proof.sty from bussproofs.sty is that the package bussproofs.sty only supports nodes till *ternary*-node while proof.sty supports $n-ary$-node. Next we describe the commands (in proof.sty) that allow to typeset proof trees.

1. \infer takes as argument the name of the rule and displays it.
2. \approx corresponds to the equality and is written instead of the = in the proof
3. The curly brackets { and } are used for determining the scopes.
4. The & is used for concatenation of the equalities by the rules transitivity and congruence with two or more arguments.
5. The double $$ sign for beginning and ending the proof.

[1]`http://research.nii.ac.jp/~tatsuta/index-e.html`
[2]`http://www.math.ucsd.edu/~sbuss/ResearchWeb/bussproofs/index.html`

Example 4.5. : Given the proof in Example 4.2 . Below we see its TEX code with respect to the package proof.sty. We can see how it displays in tree-view in Figure 4.17 .

```
$$
\infer[(cong)]{h(g(d,f)) \approx h(g(a,d))}
{
\infer[(sym)]{g(d,f) \approx g(a,d)}{
\infer[(cong)]{g(a,d) \approx g(d,f)}{
\infer[(trans)]{a \approx d}{
\infer[(trans)]{a \approx c}{
\infer[(sym)]{a \approx b}{
\infer[(app)]{b \approx a}{}}
&
\infer[(sym)]{b \approx c}{
\infer[(app)]{c \approx b}{}}}
&
\infer[(app)]{c \approx d}{}}
&
\infer[(trans)]{d \approx f}{
\infer[(app)]{d \approx x}{}
&
\infer[(trans)]{x \approx f}{
\infer[(app)]{x \approx y}{}
&
\infer[(app)]{y \approx f}{}}}}}
}
$$
```

Example 4.6. Given the proof in Example 4.4 . Below we see its TEX code with respect to the package proof.sty. We can see how it displays in tree-view in Figure 4.19 .

```
$$
\infer[(cong)]{s(a,b,c,d,e,f,g,h,i)\approx s(b,c,d,e,f,g,h,i,j)}
{
\infer[(app)]{ a\approx b}{}  &
\infer[(app)]{b\approx c}{}   &
\infer[(app)]{ c\approx d}{}  &
\infer[(app)]{d\approx e}{}   &
\infer[(app)]{e\approx f}{}   &
\infer[(app)]{ f\approx g}{}  &
\infer[(app)]{g\approx h}{}   &
\infer[(app)]{h\approx i}{}   &
\infer[(app)]{ i\approx j}{}
}
$$
```

Example 4.7. : Given the proof in Example 4.3 . Below we see its TEX code with respect to the package proof.sty. We can see how it displays in tree-view in Figure 4.18 .

```
$$
\infer[(cong)]{ s(s(0)+s(0)) \approx s(s(s(0)))}
{
\infer[(trans)]{ s(0)+s(0) \approx s(s(0))}{
\infer[(app)]{ s(0)+s(0) \approx s(0+s(0))}{}
&
\infer[(cong)]{ s(0+s(0)) \approx s(s(0))}{
\infer[(app)]{ 0+s(0) \approx s(0)}{}}}
}
$$
```

4.2.1 Implementation using Java

1. Create an ArrayList <String >listIn = new ArrayList<String >(); The first step is adding the given proof (see Example 3.2) from the given file into an ArrayList. If the given proof would contain an invalid input-format then an error message displayed. In other case we add the given proof in listIn. But before adding the the first line I add a new blank line into Arraylist in order to get to first line with an equation e.g., listIn.add(0) is a "\n" and listIn.get(1) is an equation which is the starting line of the given proof file.

2. If the given proof doesn't contain a congruence rule with more than nine arguments then tree = new TreeView(listIn);, in other case error message is displayed.

3. After calling the constructor of the class TreeView, we convert the given proof line by line in TEX code and save it in the list in class TreeTex tex.infer(listIn);. The method infer(ArrayList $< String >$ In) in TreeTex is responsible for transforming line by line in TEX code.

4. The basic tree: please read (page 17) .

5. The complex tree: please read (page 18) .

6. After these steps the given proof is transformed in TEX code.

5 Natural Deduction

How do we go about constructing a calculus for reasoning about propositions, so that we can establish the validity of Example 5.1 . Clearly, we would like to have a set of rules each of which allows us to draw a conclusion given a certain arrangement of premises. In natural deduction, we have such a collection of proof rules (see Section 5.2) . They allow us to infer formulas from other formulas. By applying these rules in succession, we may infer a conclusion from a set of premises. Let's see how this works.

Suppose we have a set of formulas ϕ_1, ϕ_2, ϕ_3, $\ldots$, ϕ_n, which we will call premises, and another formula, ψ which we will call a conclusion. By applying proof rules to the premises, we hope to get some more formulas, and by applying more proof rules to those, to eventually obtain the conclusion. This intention we denote by ϕ_1,ϕ_2,ϕ_3,$\ldots$,$\phi_n \vdash \psi$.

This expression is called a sequent; it is valid if a proof for it can be found.

Example 5.1. :

$$a \approx b,\ b \approx c \vdash a \approx \mathrm{c}$$

Constructing such a proof is a creative exercise, a bit like programming. It is not necessarily obvious which rules to apply, and in what order, to obtain the desired conclusion. Additionally, our proof rules should be carefully chosen; otherwise, we might be able to prove invalid patterns of argumentation [3].

5.1 Deduction Systems

In order to define the logics and proof theories we are interested in, we shortly present the notion of deduction systems that will be intensively used.

Definition 5.2. A *Deduction system* is a 4-tuple $(\sum, \Phi, A, R)$ where:

- $\sum$ is a countable *alphabet*,
- Φ is a set of $formulas$ $\phi_0, \phi_1, \dots$ that is a decidable language over $\sum$,
- A is a set of *axioms* $a_0, a_1, \dots$ that is a decidable subset of Φ,
- R is a finite set of $inference\ rules$ $r_0, r_1, \dots, r_n$ that are computable predicates over Φ.

Infinite sets of axioms are allowed and specified by axiom schemes. An inference rule is written as:

$$\textbf{Name} \quad \begin{array}{l} \phi_0, \dots, \phi_{n-1} \vdash \phi_n \\ \qquad\qquad \text{if } \textit{condition} \end{array}$$

and means:

"Given $\phi_0, \dots, \phi_{n-1}$ deduce ϕ_n if *condition* holds. "

A *derivation* d of the conclusion $c \in \Phi$ from the *premises* $P = \{p_0, \dots, p_{n-1}\} \subseteq \Phi$ is a finite non-empty *sequence* $(d_0, \dots, d_m)$ such that $d_i \in \Phi, d_m = c$ and either $d_i, \in A(d_i$ is an axiom), or $d_i \in P(d_i$ is a premise), or d_i has been obtained by applying some inference rule in R to a set $d_j, \dots, d_k$ of formulas such that $d_j, \dots, d_k \in d$ and $j, \dots, k < i.$ This is written as:

$$p_0, \dots, p_{n-1} \vdash c.$$

A *theorem th* is a derivation from the empty set of premises, written:

$$\vdash th.$$

A derivation of a theorem is called a $proof$. For a deduction system, the set of all proofs is decidable. If the set of all theorems is decidable, the deduction system is called *decidable.* If the set of all theorems is undecidable but semi-decidable, the deduction system is said $semi-decidable.$ If the set of all theorems is not even semi-decidable, the deduction system is said *undecidable.* An algorithm that computes a decidable set of theorems is called a *decision precedure* for the deduction system [4].

5.2 Rules for Natural Deduction

As rules are used the rules which we mentioned before (see Chapter 2.4.6) and additionally the rule premise.

5.2.1 Premise

Premise allows us to conclude the premise itself, we conclude the theory below the inference-line by the theory above the line.

The rule premise

$$\frac{s \approx t}{s \approx t}\ (prem)$$

for all terms s and t.

5.3 Algorithm

The algorithm of proving in so called linear-view is here to transform the lines in a given file line by line i.e., every line is transformed in its TEX code immediately.

5.4 Implementation

For implementing the given proof in linear-view we used here the package boxproof.sty [7] which is written by Christian Sternagel from the University of Innsbruck.

And next we describe the commands (in boxproof.sty) that allow to typeset proof linear-view:

1. Every proof begins with \begin{boxproof} and ends with \end{boxproof}.
2. Every proof-line is identified with \pline followed by its number of reference line.
3. The name of the rule is written by \text .
4. The proof-line-reference is indicated by \pref .

Example 5.3. : Given were the proof in Example 4.2 . Below we see its TEX code with respect to the package boxproof.sty .

```
\begin{boxproof}
\pline[1]{b \approx a}{\text{prem}}
\pline[2]{c \approx b}{\text{prem}}
\pline[3]{c \approx d}{\text{prem}}
\pline[4]{d \approx x}{\text{prem}}
\pline[5]{x \approx y}{\text{prem}}
\pline[6]{y \approx f}{\text{prem}}
\pline[7]{a \approx b}{\text{sym \pref{1}}}
\pline[8]{b \approx c}{\text{sym \pref{2}}}
\pline[9]{a \approx c}{\text{{trans} \pref{7}, \pref{8}}}
\pline[10]{a \approx d}{\text{{trans} \pref{9}, \pref{3}}}
\pline[11]{x \approx f}{\text{{trans} \pref{5}, \pref{6}}}
\pline[12]{d \approx f}{\text{{trans} \pref{4}, \pref{11}}}
\pline[13]{g(a,d) \approx g(d,f)}{\text{{cong} \pref{10}, \pref{12}}}
\pline[14]{g(d,f) \approx g(a,d)}{\text{sym \pref{13}}}
\pline[15]{h(g(d,f)) \approx h(g(a,d))}{\text{{cong} \pref{14}}}
\end{boxproof}
```

Example 5.4. : Here we see the displayed-view of the TEX code corresponding to the Example 5.3 .

1	$b \approx a$	prem
2	$c \approx b$	prem
3	$c \approx d$	prem
4	$d \approx x$	prem
5	$x \approx y$	prem
6	$y \approx f$	prem
7	$a \approx b$	sym 1
8	$b \approx c$	sym 2
9	$a \approx c$	trans 7, 8
10	$a \approx d$	trans 9, 3
11	$x \approx f$	trans 5, 6
12	$d \approx f$	trans 4, 11
13	$g(a,d) \approx g(d,f)$	cong 10, 12
14	$g(d,f) \approx g(a,d)$	sym 13
15	$h(g(d,f)) \approx h(g(a,d))$	cong 14

Example 5.5. : Let's see another example from the lecture script of Term Rewriting by Prof. Aart Middeldorp [5]. We use here the structure according to keywords (read on page 2) .

$$1; s(x) + s(y) = s(x + s(y)); prem$$
$$2; 0 + s(y) = s(y); prem$$
$$3; s(0) + s(0) = s(0 + s(0)); app1$$
$$4; 0 + s(0) = s(0); app2$$
$$5; s(0 + s(0)) = s(s(0)); cong4$$
$$6; s(0) + s(0) = s(s(0)); trans3, 5$$
$$7; s(s(0) + s(0)) = s(s(s(0))); cong6$$

Figure 5.1: Input: proof.txt

Given is the proof.txt file above which we transform in TEX code as below Figure 5.2 .

```
\begin{boxproof}
\pline[1]{s(x)+s(y) \approx s(x+s(y))}{\text{prem}}
\pline[2]{0+s(y) \approx s(y)}{\text{prem}}
\pline[3]{s(0)+s(0) \approx s(0+s(0))}{\text{app \pref{1}}}
\pline[4]{0+s(0) \approx s(0)}{\text{app \pref{2}}}
\pline[5]{s(0+s(0)) \approx s(s(0))}{\text{{cong} \pref{4}}}
\pline[6]{s(0)+s(0) \approx s(s(0))}{\text{{trans} \pref{3}, \pref{5}}}
\pline[7]{s(s(0)+s(0)) \approx s(s(s(0)))}{\text{{cong} \pref{6}}}
\end{boxproof}
```

Figure 5.2: TEX-output: proof_linear.tex

The given file in Figure 5.1 is transformed with respect to the package boxproof.sty.

1	$s(x) + s(y) \approx s(x + s(y))$	prem
2	$0 + s(y) \approx s(y)$	prem
3	$s(0) + s(0) \approx s(0 + s(0))$	app 1
4	$0 + s(0) \approx s(0)$	app 2
5	$s(0 + s(0)) \approx s(s(0))$	cong 4
6	$s(0) + s(0) \approx s(s(0))$	trans 3, 5
7	$s(s(0) + s(0)) \approx s(s(s(0)))$	cong 6

Figure 5.3: Pdf-output: proof_linear.pdf

The transformed file in Figure 5.2 file is displayed as above.

5.4.1 Implementation using Java

The content of the given file is read via Scanner and added in an arraylist which is listIn. Then the method linearview(listIn) is called.

The needed TEX code is added here and this method calls an other method which is pline(line), for transforming the lines in TEX according to the package boxproof.sty. And the method pline(line) takes every line as an argument and transforms it.

6 How to Use It

6.1 User Manual

Firstly we need a file for transforming. The input-format is described in Chapter 3. Go to the folder where the class LinearView.java is. And then compile it with “javac LinearTree.java” . After it is compiled for transforming execute:
“java LinearTree [filename] [*linear* | *tree* | *lineartree* | *treelinear*]”
E.g., “java LinearTree proof tree” .

Now the proof in the file has been transformed and written into proof_tree.tex. For simplicity you can type e.g., “lineartree” or “treelinear” for transforming in both views and the TEX files *proof_linear.tex* and *proof_tree.tex* are created.

6.2 Test Cases

Lets have a look the cases after compiling the program with “javac LinearTree.java” . We have the cases with error messages and with success.

6.2.1 Test Cases with Error Messages

1. java LinearTree

```
***************************************************************
**** No argument entered! Type like: *************************
java LinearTree [filename] [linear|tree|lineartree|treelinear]
***************************************************************
```

2. java LinearTree nofile

```
***************************************************************
 File not found! Please write the name of the file correctly!
***************************************************************
```

3. java LinearTree proof.txt

```
**************************************************************
**** Second argument not entered! Type like: *****************
java LinearTree [filename] [linear|tree|lineartree|treelinear]
**************************************************************
```

4. java LinearTree proof.txt invalidargument

```
**************************************************************
** Wrong tex format! Type: [linear|tree|lineartree|treelinear]
**************************************************************
```

5. In this case we have a file whose content contains the wrong typesetting. But the program detects the wrong line in this file.

```
1    a = b;prem


    2 ; c =          d;   premise

 3; b = c    prem
     4 ; f(b) =         f(c);   cong
 5; a c ;    trans1,3
       ; c =          a;  5
   7; f(a)=f(c);
   8; f(c)=f(a); sym
   9; f(b)=f(a); trans48
   10; f(a)=f(b);sym 9
```

We enter the following command line in terminal. “java LinearTree proof.txt tree” . And then the error message is displayed.

```
**************************************************************
**** The line/lines is/are below wrong! **********************
**************************************************************
**** A valid format has the structure ***********************
\d+;.+=.+;cong\d+(,\d+)*|trans\d+,\d+|sym\d+|app\d+|ref|prem
White-spaces and blank-line don't cause error! They are allowed!
**************************************************************
1a=b;prem
2;c=d;premise
3;b=cprem
4;f(b)=f(c);cong
5;ac;trans1,3
;c=a;5
7;f(a)=f(c);
8;f(c)=f(a);sym
9;f(b)=f(a);trans48
```

Above we see the wrong lines. For explanation of the correct-format please (see Table 3.1) .

6. In tree-view at most nine arguments are allowed. Given were the proof file in Figure 6.1. Type the line below in a terminal, than you get the error message as follows.
   ”java LinearTree proof.txt tree”

```
1; a = b;prem
2; b = c;prem
3; c = d;prem
4; d= e;prem
5; e= f;prem
6; f= g;prem
7; g=h;prem
8; h=i;prem
9; i=j;prem
10; j=k;prem
11; s(a,b,c,d,e,f,g,h,i,j) = s(b,c,d,e,f,g,h,i,j,k);cong1,2,3,4,5,6,7,8,9,10
```

Figure 6.1: proof.txt

```
**************************************************************
**************************************************************
**** Cong has more than 9 arguments *************************
**** Not allowed in treeview but in linearview ***************
**** No file created!                          ***************
**************************************************************
```

If we enter “java LinearTree proof.txt treelinear” or “java LinearTree proof.txt lineartree” we get the error message below.

```
*********************************************************
*********************************************************
**** Cong has more than 9 arguments *********************
**** Not allowed in treeview but in linearview **********
**** proof_linear.tex is created!          **************
*********************************************************
```

6.2.2 Test Cases with Success

After reading Chapter 3 you create a file such as given below with file-name “proof” .

Given

$1; a = b; prem$

$2; b = c; prem$

$3; a = c; trans 1, 2$

Figure 6.2: proof

After creating this file:

1. If you want to transform in linear-view you type like: “javac LinearTree.java” and continue with “java LinearTree proof linear” then the file proof_linear.tex Figure 6.3 is created and finally you can convert the transformed TEX file into PDF with “pdflatex proof_linear.tex” Figure 6.4 .

Transformed

```
\begin{boxproof}
\pline[1]{a \approx b}{\text{prem}}
\pline[2]{b \approx c}{\text{prem}}
\pline[3]{a \approx c}{\text{{trans}
\pref{1}, \pref{2}}}
\end{boxproof}
```

Figure 6.3: proof_linear.tex

Displayed

1	$a \approx b$	prem
2	$b \approx c$	prem
3	$a \approx c$	trans 1, 2

Figure 6.4: proof_linear.pdf

2. If you want to transform in tree-view you type like: “javac LinearTree.java” and continue with “java LinearTree proof tree” then the file proof_tree.tex Figure 6.5 is created and finally you can convert the transformed TEX file into PDF using the command “pdflatex proof_tree.tex” Figure 6.6 .

Transformed

```
$$
\infer[(trans)]{a \approx c}{
\infer[(app)]{a \approx b}{}
\infer[(app)]{b \approx c}{}}
$$
```

Figure 6.5: proof_tree.tex

Displayed

$$\dfrac{\dfrac{}{a \approx b}\,(app) \quad \dfrac{}{b \approx c}\,(app)}{a \approx c}\,(trans)$$

Figure 6.6: proof_tree.pdf

7 Conclusion

We have recalled the proof rules of equational logic and have discussed two ways of displaying proofs.

One is a tree-like representation while the other one adheres to a linear-shape.

Using the implemented program we are able to transform the given proof in plain text format in different styles such as linear-view and tree-view. The packages for proving in linear-view and tree-view are not restricted with boxproof.sty and proof.sty. There are also many other packages which you can use.

To conclude we stress the situation before and after this thesis:

1. before this project the lecturer had to write separately the equational logic poof depending on the used packages for linear-view and tree-view
2. after this project the lecturer need not care for writing in TEX. He typesets proofs in an easy text format and use the program for transforming into different LATEX formats

As future work that could have here part of this project, we mention the possibility of a proof checker, i.e., an automated program that verifies each step in an equational logic proof.

We note that proof generation, i.e., finding an equational logic proof is undecidable [1].

Bibliography

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge, 1998.

[2] S. R. Buss. *LaTeX for Logicians, bussproofs.sty.* `www.math.ucsd.edu/~sbuss/ResearchWeb/bussproofs/index.html`

[3] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge, 2004.

[4] C. Kirchner and H. Kirchner. *Rewriting Solving Proving.* Not published, 2006. Campus scientifique 615, rue du Jardin Botanique BP 101 54602 Villers-les-Nancy CEDEX e FRANCE `http://www.loria.fr/~hkirchne/rsp.pdf`

[5] A. Middeldorp. *Term Rewriting.* Lecture Notes, 2011. University of Innsbruck. `http://cl-informatik.uibk.ac.at/teaching/ws11/trs/content.php`

[6] A. Sakharov. Equational logic from mathworld. `http://mathworld.wolfram.com/EquationalLogic.html`

[7] C. Sternagel. *The boxproof Package.* `http://cl-informatik.uibk.ac.at/users/griff/`

[8] M. Tatsuta. *Proof Figure Macros for LaTeX, proof.sty.* National Institute of Informatics from Japan. `http://research.nii.ac.jp/~tatsuta/proof-sty.html`

[9] Xiemaisi. *Equational Logic, lecture3-handout.pdf.* University of Sinica in Taiwan. `www.iis.sinica.edu.tw/`