Imperial College London

Department of Computing

Phometa — a visualised proof assistant that build a formal system and prove its theorems using derivation trees

Author:
Gun Pinyo

Supervisor:
Dr. Krysia Broda

Second Marker:
Prof. Alessio R. Lomuscio

June 6, 2016

Submitted in part of fulfilment of the requirements for the master of engineering

Abstract

Manually drawing a derivation tree usually takes many iterations to be completed due to its layout (its width grows exponentially to its height) and variables being rewrite (by unification when derivation rule is applied). Even when the tree is completed, there are nothing to guarantee that the tree is error free.

This thesis describes *Phometa* which is a proof assistant that allows user to create a formal system and prove its theorems using derivation trees. Fundamentally, Phometa consists of three kinds of node which are *Grammar* (Backus-Naur Form), *Rule* (derivation rule), and *Theorem* (derivation tree).

It can be used as educational platform for students to learn certain formal systems provided in standard library. Alternatively, it also can be used as experimental sandbox where user implements their own formal system and try to reason about it.

Phometa is a web application so components such as terms and derivation trees can be rendered nicely in web browser and users can interact with these components directly by clicking button or pressing keyboard shortcut. Visualisation also allows Phometa to have certain features that text-based proof assistants couldn't have, for example, nested underlines can be used to group terms instead of brackets, input method of terms can be controlled in such a way that ill-from terms couldn't be created, and so on.

Phometa has been designed and been implemented in such a way that it is powerful enough to completely replace derivation-tree's manually-drawing, and easy enough to be used by anyone. Its standard library also include famous formal systems such as *Simple Arithmetic*, *Propositional Logic*, and *Typed Lambda Calculus.* This shows that Phometa is generic enough to handle most of formal systems out there.

Acknowledgements

First and most importantly, I would like to thank my supervisor, Dr Krysia Broda, for her constant dedication to supporting me on both project skill and mental advise.

I also would like to thank my second marker, Prof. Alessio R. Lomuscio, for his invaluable advice on interim report.

I also would like to thank many of Imperial's computing students, for their feedback on my project.

I also would like to thank Evan Czaplicki and the rest of Elm community, for their ambition and effort to develop Elm — a wonderful front-end functional reactive language that Phometa is built on top of.

And finally, I also would like to thank my parents for their unconditional love and continuous support on everything since I was born.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Objectives . . . 5
- 1.3 Achievement . . . 5

2 Related Work **6**
- 2.1 Text-Base Proof Assistants . . . 6
  - 2.1.1 Coq . . . 6
  - 2.1.2 Agda . . . 7
  - 2.1.3 Isabelle . . . 8
- 2.2 Visualised Proof Assistant . . . 9
  - 2.2.1 Logitext . . . 9
  - 2.2.2 Panda . . . 10
  - 2.2.3 Pandora . . . 11

3 Background **12**
- 3.1 Formal System . . . 12
- 3.2 Backus-Naur Form . . . 12
- 3.3 Meta Variables and Pattern Matching . . . 15
- 3.4 Derivation of Formal Systems . . . 16

4 Example Formal System — Simple Arithmetic **19**
- 4.1 First time with Phometa . . . 19
- 4.2 Grammars . . . 21
- 4.3 Rules . . . 22
- 4.4 Theorems and Lemmas . . . 24
- 4.5 More complex theorem . . . 29
- 4.6 Exercises . . . 34

5 Example Formal System — Propositional Logic **35**
- 5.1 Grammars . . . 35
- 5.2 Hypothesis rules . . . 38
- 5.3 Main Rules . . . 40

5.4 Classical Logic 43
5.5 Validity of Proposition 45
5.6 Context manipulation 46
5.7 How to build Grammars and Rules 48
5.7.1 Recreation of `Prop` 48
5.7.2 Recreation of `or-elim` 50
5.7.3 Recreation of `hypothesis` 52
5.8 Exercises 54

6 Example Formal System - Lambda Calculus **56**
6.1 Untyped Lambda Calculus 56
6.2 Simply-typed Lambda Calculus 56

7 Specification **57**
7.1 Overview 57
7.2 Repository 57
7.3 Auto Complete 57
7.4 Node Comment 58
7.5 Node Grammar 58
7.6 Root Term 58
7.7 Node Rule 58
7.8 Node Theorem 58

8 Implementation **59**
8.1 Decision on programming language 59
8.2 Model-Controller-View Architecture 60
8.3 The Anatomy of Phometa Repository 62
8.4 Model, Command, Keymap, and Action 63
8.5 Structure of Proofs Repository 64
8.6 Samples of code — Pattern Matching 68
8.7 Building a Complied Version 74
8.8 Back-End Communication 74
8.9 Testing / Continious Integration 75

9 Evaluation **77**
9.1 Users Feedback — discuss with friends 77
9.2 Users feedback — discuss with junior students 78
9.3 Strengths 78
9.4 Limitation 79

10 Conclusion **81**
10.1 Lesson Learnt 81
10.2 Future Works 82

Chapter 1

Introduction

1.1 Motivation

Proofs are very important to all kinds of Mathematics because they ensure the correctness of theorems. However, it is hard to verify the correctness of a proof itself especially for a complex proof. To tackle this problem, we can prove a theorem on a *proof assistant*, aka *interactive theorem prover*, which provides a rigorous method to construct a proof such that an invalid proof will never occur. Therefore if we manage to complete a proof, it is guaranteed that the proof is valid.

There are many powerful and famous proof assistants such as Coq[12], Agda[5], and Isabelle[4] which are suitable for extreme use case of complex proofs. Nevertheless, they have a steep learning curve and have specific meta-theory behind it, for example, Coq has Calculus of Inductive Construction (CIC), Agda has Unified Theory of Dependent Types[15][14] which are quite hard for newcomers. To solve this problem they should start with something easier than these and come back again later.

One of the easiest starting point to learn about formal proof is to use derivation trees where validity of a term is derived from a derivation rule together with validity of zero or more terms depending on the rule. These prerequisite terms can be proven similarly to the main term. The proving process will happen recursively, this lead to tree-like structure of the final proof, this is why it is called "derivation tree".

The naive way to construct such a derivation tree is to draw it on a paper, however, this has many disadvantages such as

- The width of a derivation tree usually grow exponentially to its height — hard to arrange the layout on a paper.
- We don't know that how much space that each branch requires — need to recreate the tree for many iterations.

- Variables might need to be rewritten by other terms as a result of internal unification when applying a rule — again, need to recreate the tree for many iterations.
- When a derivation tree is completed, there is nothing to guarantee that it doesn't have any errors — conflict with the ambition to use proof assistant at the first place.

So I decided to create a proof assistants called *Phometa* to solve this derivation-tree manually-drawing problem. To be precise, Phometa is proof assistant that allows user to create a formal system and prove theorems using derivation trees.

Phometa fundamentally consists of three kinds[1] of node as the following

- *Grammar* (or Backus-Naur Form) — How to construct a well-form term.
  For example, a simple arithmetic expression can be constructed by a number *or* two expressions adding together *or* two expressions multiply together.
- *Rule* (or derivation rule) — A reason that can be used to prove validity of terms.
  For example, $(u + v) = (x + y)$ is valid if $u = x$ and $v = y$.
- *Theorem* (or derivation tree) — An evidence (proof) showing that
  a particular term is valid. For example,

  $((3 + 4) \times 5) = ((4 + 3) \times 5)$ is valid by rule ADD-INTRO and

  $(3 + 4) = (4 + 3)$ is valid by rule ADD-COMM

  $5 = 5$ is valid by rule EQ-REFL

A formal system will be represented by a set of grammars and rules. Validity of terms will be represented by theorems (derivation trees).

In term of usage, users can Phometa by either

- Learn one of many existing formal systems provided in Phometa's standard library and try to proof some theorem regarding to that formal system.
- Create their own formal system or extend an existing formal system, then do some experiments about it.

In order to make Phometa easy to use, it is designed to be web-based application. Users will interact with Phometa mainly by clicking buttons and pressing keyboard-shortcut. This has advantages over traditional proof assistant because it is easier to read, ill-from terms never occur, and guarantee that the entire system is always in consistent state.

[1]There exists the fourth kind of node which are comment node but I don't include it there since it is not relevant to the fundamental concept.

1.2 Objectives

- To make a construction of derivation tree become more systematic. Hence, users become more productive and have less chance to make an error.
- To encourage users to create their own formal systems and reason about it.
- To show that most of formal systems have a similar meta-structure which can be implemented using common framework.
- To show advantages of visualised proof assistant over traditional one.

1.3 Achievement

- Finished designing Phometa specification in such a way to keep it simple yet be able to produce a complex proof.
- Finished implementing Phometa. All of basic functionality is working.
- Encoded several formal systems such as Simple Arithmetic, Propositional Logic, and Typed Lambda Calculus as standard library in Phometa.
- Wrote a tutorial for newcomers to use Phometa (chapters 3, 4, 5, and 6).

Chapter 2

Related Work

There are many proof assistants available out there, each of them rely on slightly different meta-theory. We can separate proof assistants into 2 categories which are text-base proof assistants and visualised proof assistants.

2.1 Text-Base Proof Assistants

Text-base proof assistants are similar to programming language where user writes everything in text-files and compile it, if the compilation is successful, then the proofs are correct. User can freely manipulate these text-files, hence, easier to write a complex proof. In addition, most of proof assistants have a plug-in to mainstream text editor, so user can use their favourite text editor with full performance.

There are several mainstream text-base proof assistants that worth mentioning

2.1.1 Coq

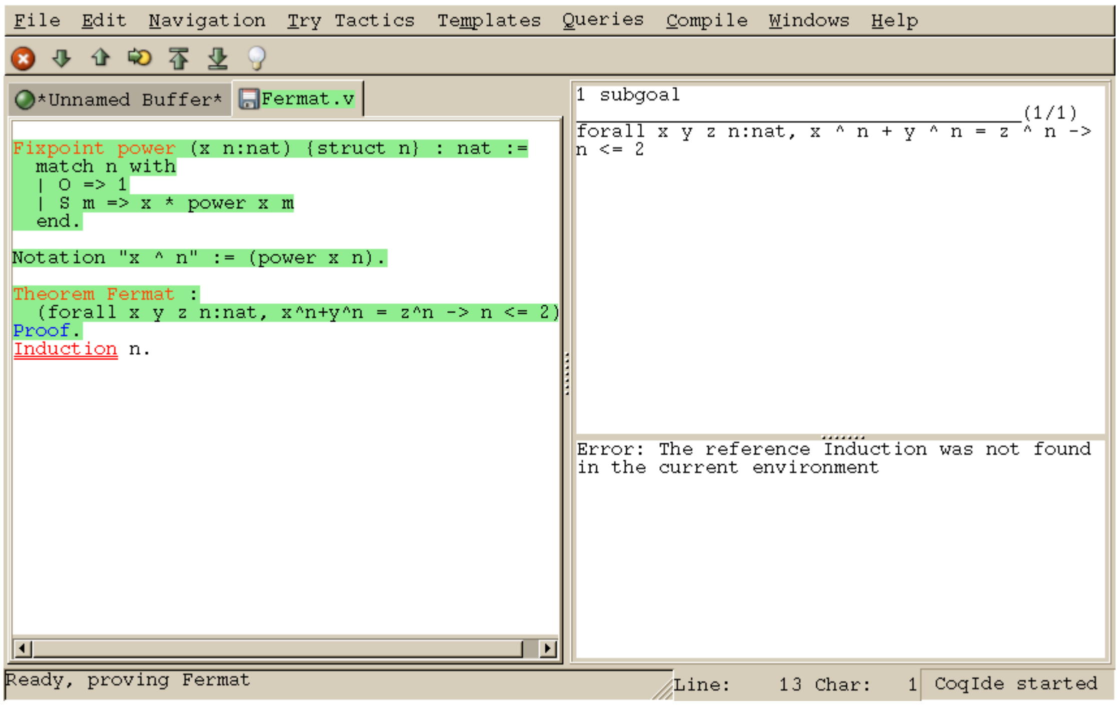
Coq[12] is one the most famous proof assistants. It is based on the Calculus of Inductive Constructions (CIC)[1] developed by Thierry Coquand[6].

Coq has customisable tactics which are commands that transform goal into smaller-sub goal (if any), this makes proving process become faster compared to other proof assistants. In contrast, tactics reduce readability, reader might need to replay each tactic step by step in order to understand a proof completely.

Coq is very mature, it has been developed since 1984. Hence, it is reliable and has lots of libraries supported.

[1]**CIC is itself is developed alongside Coq.**.

In term of editor, most people use Proof General[8] which is a plugin on Emacs[2]. Nevertheless, Coq has its own editor called CoqIde[13] that newcomers can use without learning Emacs.


File Edit Navigation Try Tactics Templates Queries Compile Windows Help
Unnamed Buffer
Fermat.v

```
Fixpoint power (x n:nat) {struct n} : nat :=
  match n with
  | O => 1
  | S m => x * power x m
  end.

Notation "x ^ n" := (power x n).

Theorem Fermat :
  (forall x y z n:nat, x^n+y^n = z^n -> n <= 2)
Proof.
Induction n.
```

```
1 subgoal
______________________________________(1/1)
forall x y z n:nat, x ^ n + y ^ n = z ^ n ->
n <= 2
```

```
Error: The reference Induction was not found
in the current environment
```

```
Ready, proving Fermat                Line:   13 Char:   1  CoqIde started
```

Figure 2.1: Screenshot of Coq (using CoqIde, Credit: [12]) — The left pane is file content and Upper right pane is the current goal which is changed depending on where the cursor point on file content.

2.1.2 Agda

Adga[5] is (dependently typed) functional programming which can be seen as a proof assistant as well. It is based on Unified Theory of Dependent Types[15][14] similar to Martin Lof Type Theory.

Its proving technique is relies on Curry-Howard correspondence which state that there is duality between computer programs and mathematical proofs[18], for example function corresponded to implication, product type corresponded to logical implication.

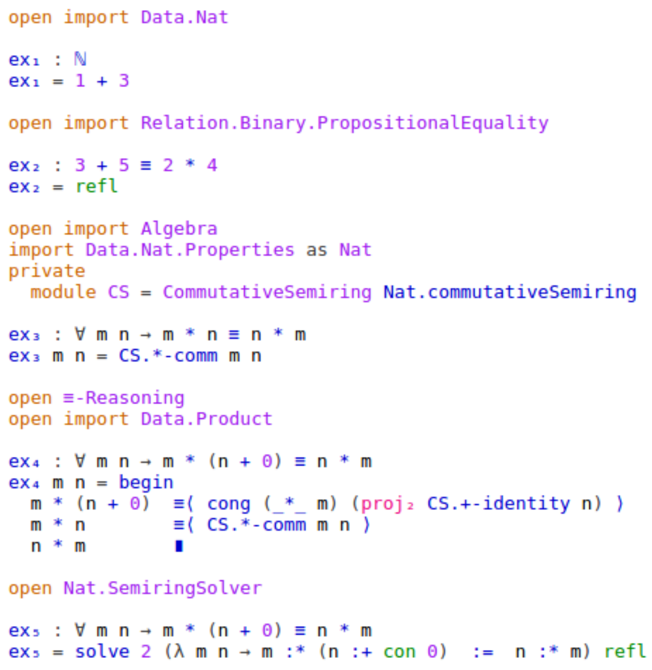
Agda is suitable for reasoning about functional programs because we can write a program and prove that certain properties of a function hold using the same language. This is feasible since a proof is just a function due to Curry-Howard correspondence.

[2]Proof General also other proof assistants such as Isabelle and PhoX

Agda has less steep learning curve compared other proof assistants such as Coq. This is because user doesn't need to learn about proving system since it is the same as programming. In contrast, it doesn't have fancy tactic system so proving process is slower.

In term of popularity, it is less popular than Coq, however, some project such as Homotopy Type Theory[10][11] use Agda as alternative experiments to Coq.

In term of editor, Agda as its own plugin for Emacs which is very nice but user need to be familiar to Emacs before using it. There is no alternative plugin to other editor.



```
open import Data.Nat

ex₁ : ℕ
ex₁ = 1 + 3

open import Relation.Binary.PropositionalEquality

ex₂ : 3 + 5 ≡ 2 * 4
ex₂ = refl

open import Algebra
import Data.Nat.Properties as Nat
private
  module CS = CommutativeSemiring Nat.commutativeSemiring

ex₃ : ∀ m n → m * n ≡ n * m
ex₃ m n = CS.*-comm m n

open ≡-Reasoning
open import Data.Product

ex₄ : ∀ m n → m * (n + 0) ≡ n * m
ex₄ m n = begin
  m * (n + 0)  ≡⟨ cong (_*_ m) (proj₂ CS.+-identity n) ⟩
  m * n        ≡⟨ CS.*-comm m n ⟩
  n * m        ∎

open Nat.SemiringSolver

ex₅ : ∀ m n → m * (n + 0) ≡ n * m
ex₅ = solve 2 (λ m n → m :* (n :+ con 0)  :=  n :* m) refl
```

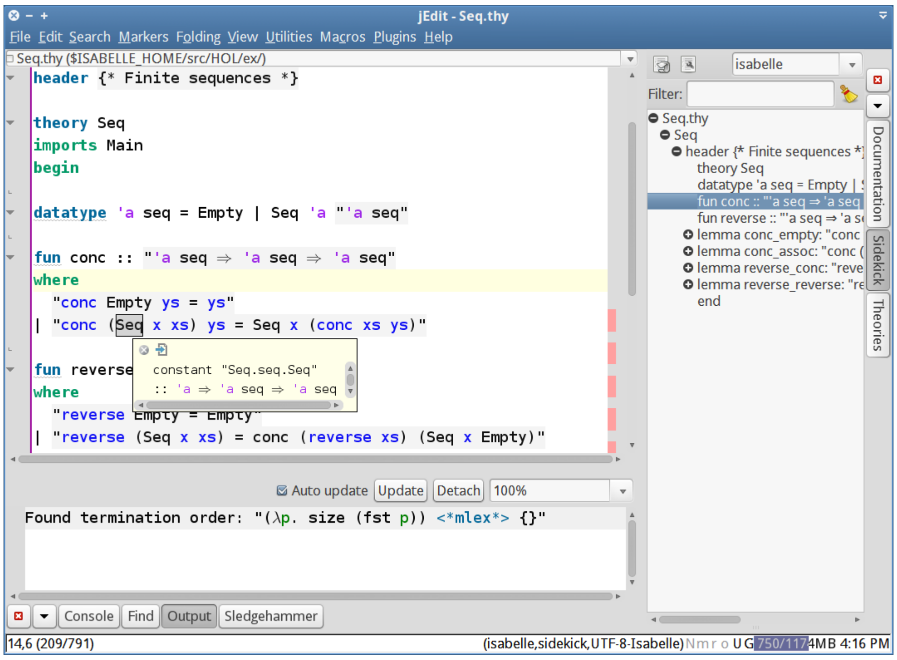
Figure 2.2: Screenshot of Agda — Credit: an example in Agda standard library, removing comment out to save space.

2.1.3 Isabelle

Isabelle[4] is generic proof assistant which allows user to express mathematical formulae in a formal language and provide a tool to prove something about it. There are many systems that Isabelle supports but the most widespread one is *Isabelle/HOL* which provide a higher order logic environment that is ready for a big application.

One of the main advantage of Isabelle is its readability, the proofs will be constructed by a language called *Isar* which is designed in such a way that it could be read easily by both of computers and humans. Another advantage of Isabelle is that some part of a proof can be automatically proven, this improve user productivity.

In term of editor, Isabelle has default user interface and Prover IDE called *Isabelle/jEdit* which is based on jEdit and Isabelle/Scala.



```
header {* Finite sequences *}

theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"
```

```
constant "Seq.seq.Seq"
:: 'a ⇒ 'a seq ⇒ 'a seq
```

```
Seq.thy
  Seq
    header {* Finite sequences *}
      theory Seq
      datatype 'a seq = Empty | S
      fun conc :: "'a seq ⇒ 'a seq
      fun reverse :: "'a seq ⇒ 'a s
    lemma conc_empty: "conc
    lemma conc_assoc: "conc (
    lemma reverse_conc: "reve
    lemma reverse_reverse: "re
      end
```

```
Found termination order: "(λp. size (fst p)) <*mlex*> {}"
```

```
14,6 (209/791)                    (isabelle,sidekick,UTF-8-Isabelle)Nmro UG 750/1174MB 4:16 PM
```

Figure 2.3: Screenshot of Isabelle (using Isabelle/jEdit, Credit: [4]) — The top-left pane shows file content and the right pane shows content structure.

2.2 Visualised Proof Assistant

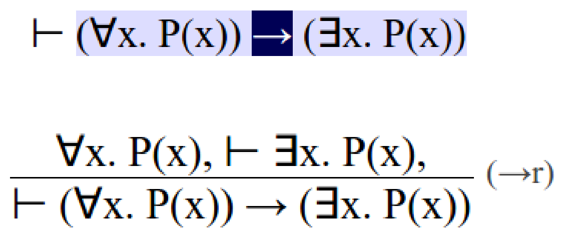
Visualised proof assistants show proofs and related contents in graphical way, it also allow user interact with proofs mainly by clicking which make them easier to use by newcomers, nevertheless, user cannot fully manipulate the proof as they have limited choices of input method, this makes it is harder have some advance features.

There are so visualised proof assistants so there, I will show a few of them to illustrate what visualised proof assistants look like.

2.2.1 Logitext

Logitext[17] is a web-based proof assistant for *first-order classical logic* using the *sequent calculus*. The main intention is to teach students about *Gentzen trees* which one of a way to struct derivation system. Logitext uses Coq internally to check validity of proof steps.

Logitext is very easy use, user usually only need to click a logical connector on the goal itself to construct a derivation tree. However, it only supports sequent calculus for first-order classical logic or propositional intuitionistic logic and user cannot extend the system.



$$\vdash (\forall x.\ P(x)) \to (\exists x.\ P(x))$$

$$\frac{\forall x.\ P(x), \vdash \exists x.\ P(x),}{\vdash (\forall x.\ P(x)) \to (\exists x.\ P(x))}\ (\to r)$$

Figure 2.4: Screenshots of Logitext — You can see that a rule could be invoke by just clicking a logical connector.

2.2.2 Panda

Panda[16][9] (**P**roof **A**ssistant for **N**atural **D**eduction for **A**ll) is graphical proof assistant that can be used to prove first order logic using Gentzen style's natural deduction.

User will interact with it mainly by drag and drop. The main advantage is its tutorial which is well integrate with the actual program itself.

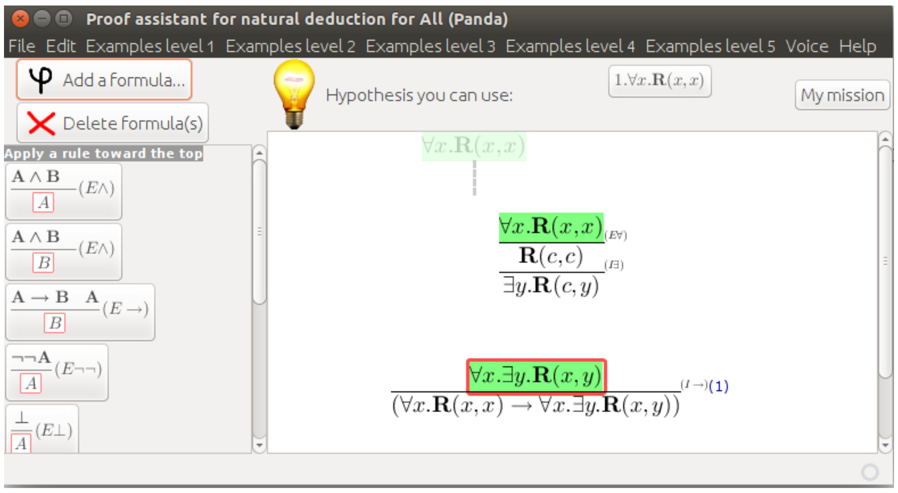
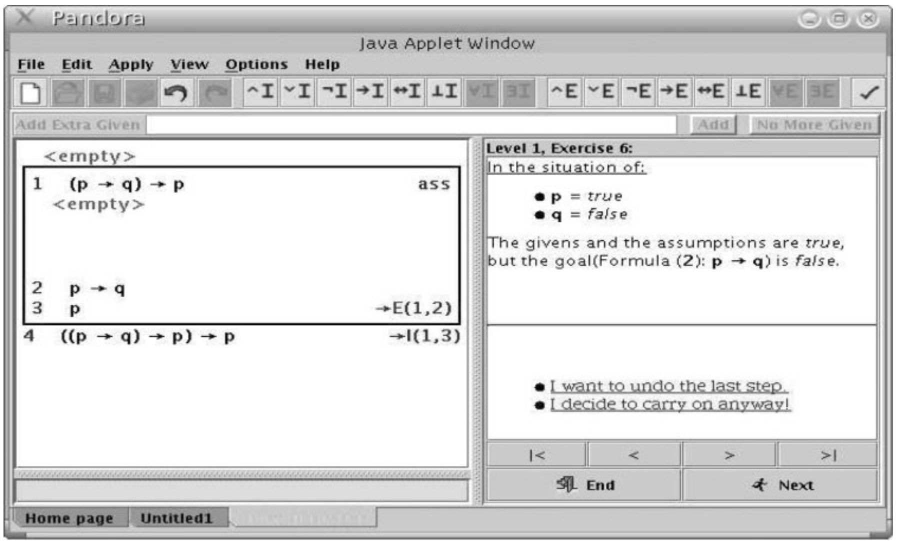

Proof assistant for natural deduction for All (Panda)
File Edit Examples level 1 Examples level 2 Examples level 3 Examples level 4 Examples level 5 Voice Help
Add a formula...
Delete formula(s)
Hypothesis you can use:
1.$\forall x.\mathbf{R}(x,x)$
My mission
Apply a rule toward the top
$\frac{A \wedge B}{A}(E\wedge)$
$\frac{A \wedge B}{B}(E\wedge)$
$\frac{A \to B \quad A}{B}(E\to)$
$\frac{\neg\neg A}{A}(E\neg\neg)$
$\frac{\bot}{A}(E\bot)$
$\forall x.\mathbf{R}(x,x)$
$\frac{\frac{\forall x.\mathbf{R}(x,x)}{\mathbf{R}(c,c)}(E\forall)}{\exists y.\mathbf{R}(c,y)}(I\exists)$
$\frac{\forall x.\exists y.\mathbf{R}(x,y)}{(\forall x.\mathbf{R}(x,x) \to \forall x.\exists y.\mathbf{R}(x,y))}(I\to)(1)$

Figure 2.5: Screenshots of Panda — User can expand the current tree by selecting a rule on the left pane and can drag and drop formula around to connect it to another one.

2.2.3 Pandora

Pandora[7][3] (**P**roof **A**ssistant for **N**atural **D**eduction using **O**rganised **R**ectangular **A**reas) is graphical proof assistant that can be used to prove first order logic using Fetch style's natural deduction.

The usage is quite similar to Panda, but it use boxes (Fetch style) rather than derivation tree (Gentzen style). It also have a comprehensive document for newcomer as well.



```
<empty>
1   (p → q) → p               ass
    <empty>

2   p → q
3   p                         →E(1,2)
4   ((p → q) → p) → p         →I(1,3)
```

Level 1, Exercise 6:
In the situation of:

- **p** = *true*
- **q** = *false*

The givens and the assumptions are *true*, but the goal(Formula (2): **p → q**) is *false*.

- I want to undo the last step.
- I decide to carry on anyway!

Figure 2.6: Screenshots of Pandora (Credit: [3]) — The left pane shows current proof, user can use a rule by clicking on the upper pane.

Chapter 3

Background

In this chapter, we will go thought some required materials needed for later chapters. These can be linked together by an example of Simple Arithmetic explained below.

3.1 Formal System

A formal system is any well-defined system of abstract thought based on mathematical model[19]. Each formal system has a formal language composed of primitive symbols[1] acted by certain formation[2].

Informally, is an abstract system that has precise structures and can be reasoned about. For example, numbers (base 10) and their arithmetic (using $+$ and $\times$) could form a formal system. This is because every term (e.g. 5, $(3+1)$, (3×4)) has explicit structure and we can argue something like "*does* 12 *equal to* (3×4)" or "*for any integers a and b, $(a+b)$ is equal to $(b+a)$*".

3.2 Backus-Naur Form

Backus-Naur Form (BNF) is a way to construct a term, for example, grammars of formal system above can be defined as the following

[1]Phometa will assume that primitive symbols are any Unicode character.

```
<Digit>   ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

<Number> ::= <Digit> | <Number> <Digit>

<Expr>    ::= <Number>
            | ‘(’<Expr> ‘+’ <Expr>‘)’
            | ‘(’<Expr> ‘×’ <Expr>‘)’

<Equation> ::= <Expr> ‘=’ <Expr>
```

Figure 3.1: Backus-Naur Form of Simple Arithmetic

- A term of `<Digit>` can be either 0 or 1 or 2 or . . . or 9 and nothing else.

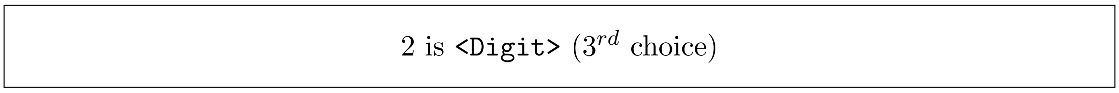


Figure 3.2: This diagram explains that why 2 is a term of `<Digit>`

- A term of `<Number>` can be either
  - `<Digit>`
  - another `<Number>` concatenate with `<Digit>`

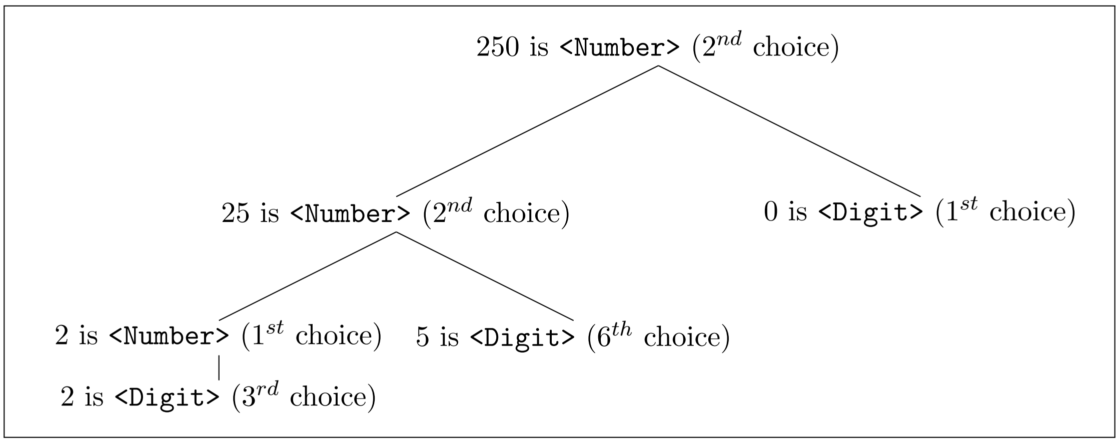

250 is <Number> (2nd choice)
25 is <Number> (2nd choice)
0 is <Digit> (1st choice)
2 is <Number> (1st choice)
5 is <Digit> (6th choice)
2 is <Digit> (3rd choice)

Figure 3.3: This diagram explains that why 250 is a term of `<Number>`

- A term of `<Expr>` can be either
  - `<Number>`
  - other two `<Expr>`s concatenate using '(' '+' ')'
  - other two `<Expr>`s concatenate using '(' '$\times$' ')'

  Please note that we need brackets around '+' and '$\times$' to avoid ambiguity. If we don't have these brackets, $3 + 4 + 5$ could be interpreted as either $(3 + 4) + 5$ or $3 + (4 + 5)$ which is not precise. Moreover, $12 + 0 \times 6$ will be interpreted as $12 + (0 \times 6)$ due to priority of $\times$ over $+$ and it is impossible to encode some thing like $(12 + 0) \times 6$.

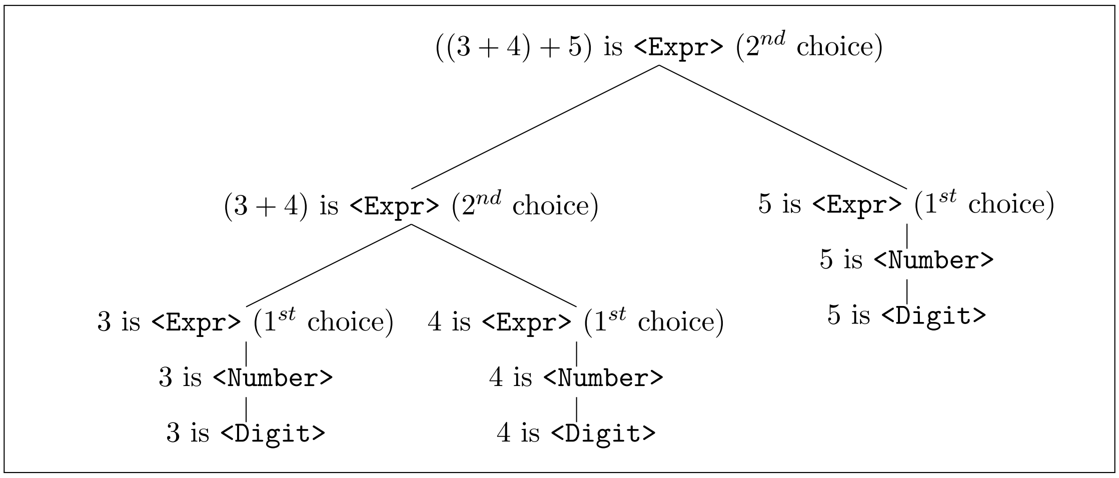


Figure 3.4: This diagram explains that why $((3 + 4) + 5)$ is a term of `<Expr>`

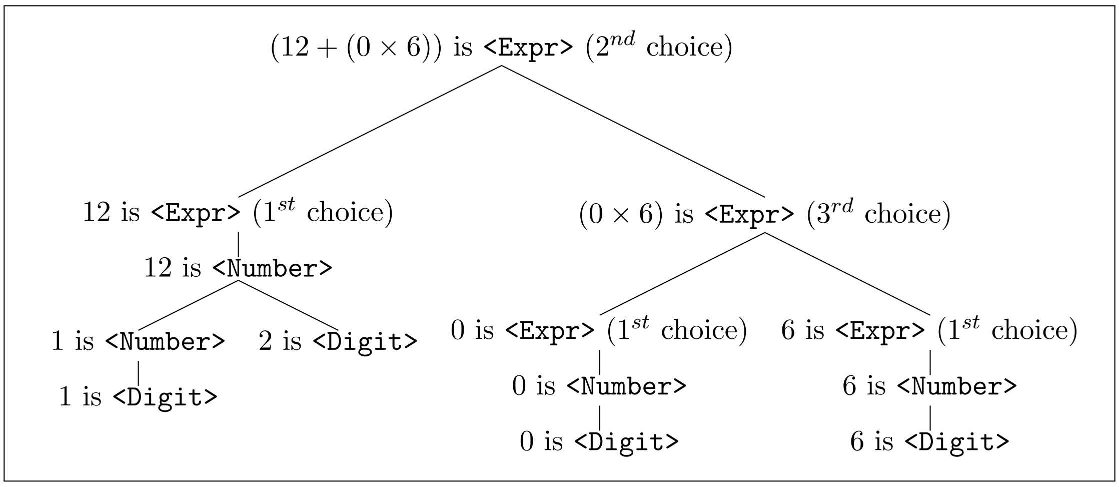


Figure 3.5: This diagram explains that why $(12 + (0 \times 6))$ is a term of `<Expr>`

- A term of `<Equation>` can be only two `<Expr>`s concatenate using '='

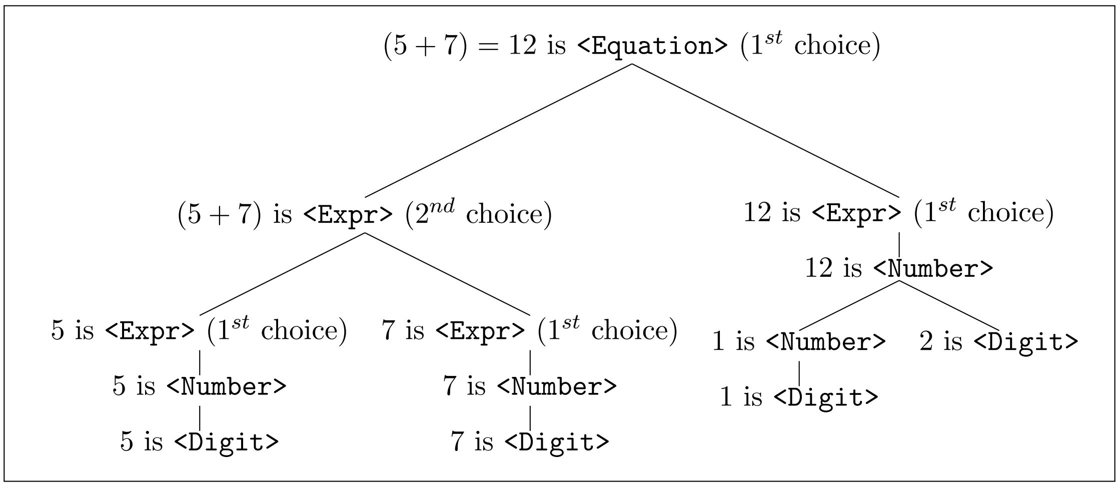

$(5+7)=12$ is <Equation> (1st choice)
$(5+7)$ is <Expr> (2nd choice)
12 is <Expr> (1st choice)
12 is <Number>
5 is <Expr> (1st choice)
7 is <Expr> (1st choice)
1 is <Number>
2 is <Digit>
5 is <Number>
7 is <Number>
1 is <Digit>
5 is <Digit>
7 is <Digit>

Figure 3.6: This diagram explains that why $(5 + 7) = 12$ is a term of `<Equation>`

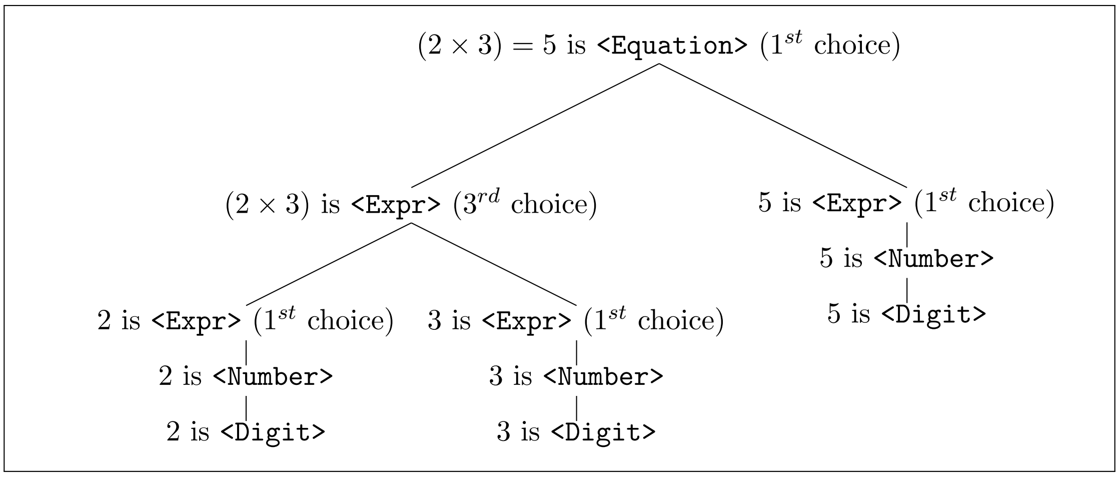

$(2\times 3)=5$ is <Equation> (1st choice)
(2×3) is <Expr> (3rd choice)
5 is <Expr> (1st choice)
5 is <Number>
5 is <Digit>
2 is <Expr> (1st choice)
3 is <Expr> (1st choice)
2 is <Number>
3 is <Number>
2 is <Digit>
3 is <Digit>

Figure 3.7: This diagram explains that why $(2 \times 3) = 5$ is a term of `<Equation>`. Please note that this construction is purely syntactic so wrong equation is acceptable.

3.3 Meta Variables and Pattern Matching

Meta variables are arbitrary sub-terms embedded inside root term. For example, an `<Expr>` $(x + y)$ represents two arbitrary `<Expr>` joined by '+'.

But if we have an `<Equation>` $(x + 7) = 12$, shouldn't x be an unknown variable that needed to be solve rather than being arbitrary `<Expr>`? Well, x still represents arbitrary `<Expr>` but in order make this equation hold, x must be 5. Hence "*variable needed to be solve*" is just spacial form of "*variable as arbitrary term*".

Meta variables help us to represents statement in more general manner. For example, "*the same expressions plus together is the same as 2 times that expression*" could be represented by $(x + x) = (2 \times x)$ rather than $(0 + 0) = (2 \times 0)$ and $(1 + 1) = (2 \times 1)$ and $(2 + 2) = (2 \times 2)$ and so on.

But if we know that $(x+x) = (2\times x)$, how could we derive its instance e.g. $(1+1) = (2\times 1)$ or even $(y \times z) + (y \times z) = (2 \times (y \times z))$? The solution for this is to use *Pattern Matching* which is algorithm that try to substitute pattern's meta variables into more specific form, in order to make pattern identical to target, for example

- $(x + x) = (2 \times x)$ is pattern matchable with $(1 + 1) = (2 \times 1)$ by substitute x with1
- $(x + x) = (2 \times x)$ is pattern matchable with $(y \times z) + (y \times z) = (2 \times (y \times z))$ by substitute x with $(y \times z)$
- $(x + x) = (2 \times x)$ is *not* pattern matchable with $(1 + 1) = 2$, if we try to substitute x with 1 we would get $(1 + 1) = (2 \times 1)$ which is not identical to $(1 + 1) = 2$
- $(1 + 1) = (2 \times 1)$ is *not* pattern matchable with $(x + x) = (2 \times x)$, because pattern $(1 + 1) = (2 \times 1)$ doesn't have any meta variable and it is not identical to $(x + x) = (2 \times x)$. This show that pattern matching doesn't generally holds in opposite direction
- $(x + x) = (2 \times x)$ is pattern matchable to itself by substitute x with x

If pattern matching is successful then the target is instance of the pattern.

3.4 Derivation of Formal Systems

So far, we construct any term based on Backus-Naur Form, this doesn't prevent invalid term, for example, $(2 \times 3) = 5$ is perfectly a term of `<Equation>`. Thus, we need some mechanism to verify a term i.e. *prove* that the particular term holds. One way to deal with this is to use derivation system, first, we have a set of derivation rules that has format as the following

$$\textsc{rule-name}\ \frac{Premise_1 \qquad Premise_2 \qquad Premise_3 \qquad \ldots \qquad Premise_n}{Conclusion}$$

Figure 3.8: Structure of derivation rule.

This say that if we know that $Premise_1$ and $Premise_2$ and $Premise_3$ and ... and $Premise_n$ hold then $Conclusion$ holds. In another word, if we want to prove $Conclusion$ then we can use this derivation rule then proof its premises.

Derivation rules of current example formal system could be shown as the following

$$\textsc{eq-refl}\ \frac{}{x = x} \qquad \textsc{eq-symm}\ \frac{y = x}{x = y} \qquad \textsc{eq-tran}\ \frac{x = z \qquad z = y}{x = y}$$

$$\textsc{add-intro}\ \frac{u = x \qquad v = y}{(u + v) = (x + y)} \qquad \textsc{mult-intro}\ \frac{u = x \qquad v = y}{(u \times v) = (x \times y)}$$

$$\textsc{add-assoc}\ \frac{}{((x + y) + z) = (x + (y + z))} \qquad \textsc{mult-assoc}\ \frac{}{((x \times y) \times z) = (x \times (y \times z))}$$

$$\textsc{add-comm}\ \frac{}{(x + y) = (y + x)} \qquad \textsc{mult-comm}\ \frac{}{(x \times y) = (y \times x)}$$

$$\textsc{dist-left}\ \frac{}{(x \times (y + z)) = ((x \times y) + (x \times z))} \qquad \textsc{dist-right}\ \frac{}{((x + y) \times z) = ((x \times z) + (y \times z))}$$

Figure 3.9: Derivation rules of Simple Arithmetic (not exhaustive, due to limited space).

In order to use a derivation rule, first the conclusion of the rule is pattern match against current goal, if it is pattern matchable then meta variables in premises are substituted respect to the pattern matching (if some meta variables of premises doesn't exist in substitution list then we are free to substitute by anything). These substituted premises will become next goals that we need to prove.

For example if we want to prove $((3+4)*5) = ((4+3)*5)$ we could use rule MULT-INTRO to prove it since $(u \times v) = (x \times y)$ is pattern matchable with $((3+4)*5) = ((4+3)*5)$ by substitute u with $(3+4)$, v with 5, x with $(4+3)$, and y with 5. Then premises $u = x$ and $v = y$ are substituted and become $(3+4) = (4+3)$ and $5 = 5$ respectively. Therefore, $((3+4)*5) = ((4+3)*5)$ can be proven by MULT-INTRO and produce another two sub-goals which are $(3+4) = (4+3)$ and $5 = 5$. This can be shown as instance of MULT-INTRO as the following

$$\dfrac{(3+4)=(4+3) \qquad 5=5}{((3+4)*5)=((4+3)*5)}\;\textsc{add-intro}$$

Figure 3.10: Example of instance of derivation rule.

For the remaining, we could prove $(3+4)=(4+3)$ using ADD-COMM because $(x+y)=(y+x)$ is pattern matchable with $(3+4)=(4+3)$, ADD-COMM doesn't have any premises hence there are no further sub-goal. For $5=5$ we could use EQ-REFL, this also deosn't produce further sub-goal so the entire proof is complete. We can the write the entire proof using *derivation tree* as the following

$$\dfrac{\dfrac{}{(3+4)=(4+3)}\;\textsc{add-comm} \qquad \dfrac{}{5=5}\;\textsc{eq-refl}}{((3+4)*5)=((4+3)*5)}\;\textsc{add-intro}$$

Figure 3.11: Example of derivation tree.

Some rules in figure 3.9 don't have any premises. This is necessary, otherwise, applying rule always generate further sub goals and never terminate. These rules can be seen as *axiom* which is a term that valid by assumption i.e. so need to prove such a term.

For better understanding about derivation system, here is a more complex derivation tree which prove $(((w\times x)+(w\times y))\times z)=(w\times((x\times z)+(y\times z)))$. Reader is encouraged to explore that why this derivation tree is correct.

$$\begin{array}{c}\dfrac{}{((w\times(x+y))\times z)=(w\times((x+y)\times z))}\;\textsc{mult-assoc}\\ \dfrac{\vdots \qquad \dfrac{\dfrac{}{w=w}\;\textsc{eq-refl} \qquad \dfrac{}{((x+y)\times z)=((x\times z)+(y\times z))}\;\textsc{dist-right}}{(w\times((x+y)\times z))=(w\times((x\times z)+(y\times z)))}\;\textsc{mult-intro}}{((w\times(x+y))\times z)=(w\times((x\times z)+(y\times z)))}\;\textsc{eq-tran}\end{array}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(w\times(x+y))=((w\times x)+(w\times y))}\;\textsc{dist-left}}{((w\times x)+(w\times y))=(w\times(x+y))}\;\textsc{eq-symm} \qquad \dfrac{}{z=z}\;\textsc{eq-refl}}{(((w\times x)+(w\times y))\times z)=((w\times(x+y))\times z)}\;\textsc{mult-intro} \qquad \vdots}{(((w\times x)+(w\times y))\times z)=(w\times((x\times z)+(y\times z)))}\;\textsc{eq-tran}$$

Figure 3.12: Example of more complex derivation tree.

Chapter 4

Example Formal System — Simple Arithmetic

As in background chapter, Simple Arithmetic is used as example to explain basic concept of formal systems and its derivations. In order to make the transition goes smoother, this chapter aims to encode Simple Arithmetic and explain basic features and usability of *Phometa* at the same time. Please note that this is just a faction of actual arithmetic modified to make it easier to understand, so it is not as powerful as the actual one.

4.1 First time with Phometa

You can download complied version of Phometa at

`https://github.com/gunpinyo/phometa/raw/master/build/phometa.tar.gz`

Once you unzip this file, you can start Phometa server by execute

```
./phometa-server.py 8080
```

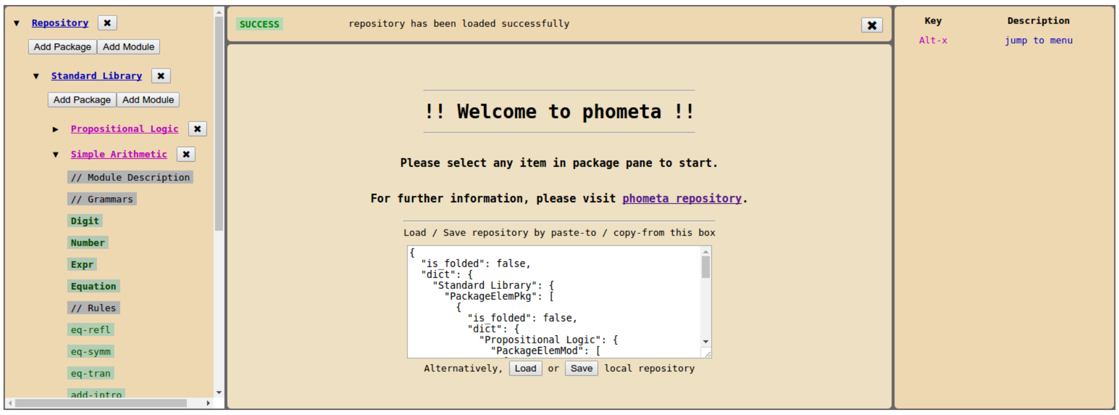
where `8080` is port number, you can change this to another port number if you like. Please note that Python is required for this server.

Then open your favourite web-browser[1] and enter

```
http://localhost:8080/phometa.html
```

The program will look like this

[1]but Google Chrome is recommended



SUCCESS repository has been loaded successfully

!! Welcome to phometa !!

Please select any item in package pane to start.

For further information, please visit phometa repository.

Load / Save repository by paste-to / copy-from this box

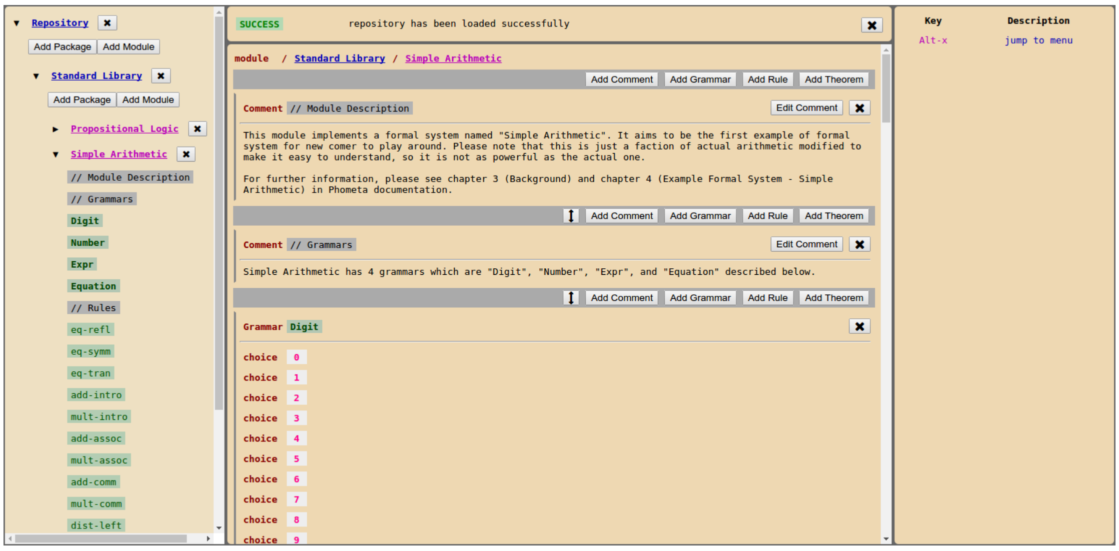
```
{
  "is_folded": false,
  "dict": {
    "Standard Library": {
      "PackageElemPkg": [
        {
          "is_folded": false,
          "dict": {
            "Propositional Logic": {
              "PackageElemMod": [
```

Alternatively, Load or Save local repository

Figure 4.1: Screenshot of Phometa when you open it from web-browser.

Phometa has a repository which consists of packages and modules that store formal systems and its proofs. The left pane of figure 4.1 shows global structure of a repository. The current repository has one package named “Standard Library” which consists of three modules named “Propositional Logic”, “Simple Arithmetic”, and “Typed Lambda Calculus”.

Module in Phometa are analogous to text file. It consists of nodes that could depend on one another. There are four types of node which are *Comment*, *Grammar* (Backus-Naur Form), *Rule* (Derivation Rule), and *Theorem* (Derivation Tree). If you click at a module on the repository pane e.g. “Simple Arithmetic”, you will see the whole content of the module appear on the centre pane. Alternatively, you can click on each node on the repository pane directly to focus on particular node.



SUCCESS repository has been loaded successfully

module / Standard Library / Simple Arithmetic

Comment // Module Description

This module implements a formal system named "Simple Arithmetic". It aims to be the first example of formal system for new comer to play around. Please note that this is just a faction of actual arithmetic modified to make it easy to understand, so it is not as powerful as the actual one.

For further information, please see chapter 3 (Background) and chapter 4 (Example Formal System - Simple Arithmetic) in Phometa documentation.

Comment // Grammars

Simple Arithmetic has 4 grammars which are "Digit", "Number", "Expr", and "Equation" described below.

Grammar Digit

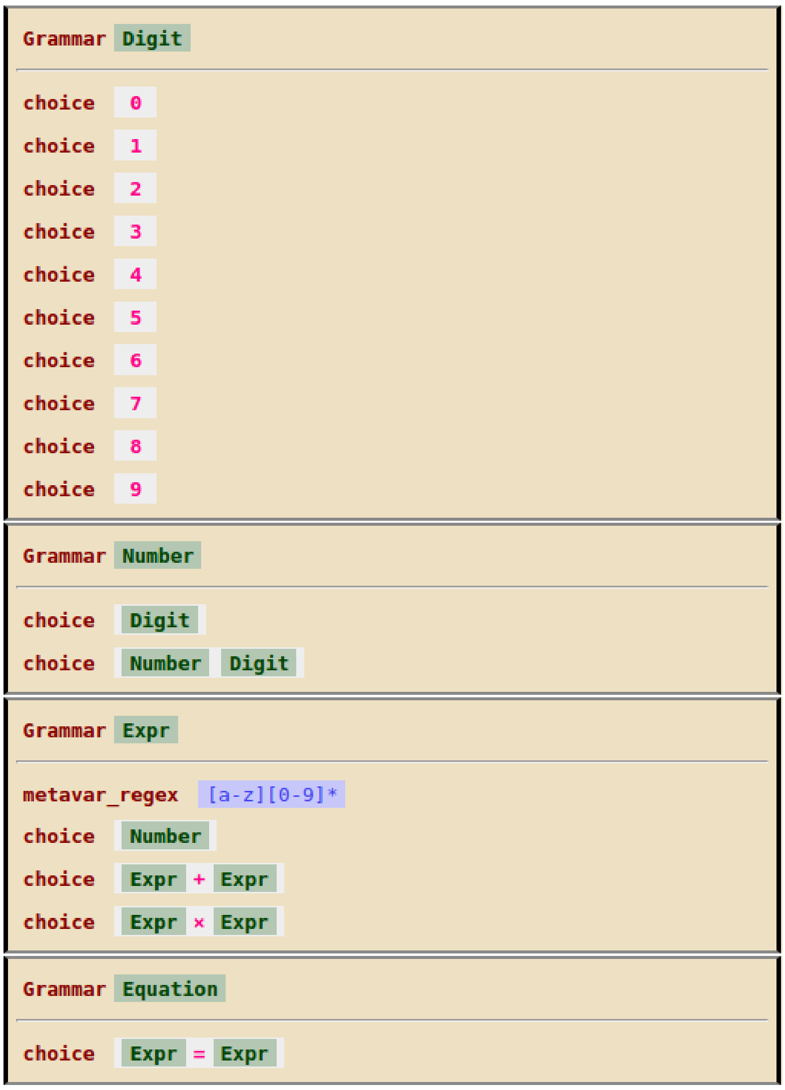
choice 0
choice 1
choice 2
choice 3
choice 4
choice 5
choice 6
choice 7
choice 8
choice 9

Figure 4.2: Screenshot of Phometa when you click “Simple Arithmetic” module.

In order to improve productivity, Phometa has several key-bindings specific to certain state of program. Fortunately, user don't need to remember any of this since the right pane (i.e. keymap pane) shows every possible key-binding with its description on current state. This also allow new-comer to explore new features during using it

4.2 Grammars

The Backus-Naur Form of Simple Arithmetic in figure 3.1 could be transformed in to this four following grammars



```
Grammar Digit

choice  0
choice  1
choice  2
choice  3
choice  4
choice  5
choice  6
choice  7
choice  8
choice  9

Grammar Number

choice  Digit
choice  Number Digit

Grammar Expr

metavar_regex  [a-z][0-9]*
choice  Number
choice  Expr + Expr
choice  Expr × Expr

Grammar Equation

choice  Expr = Expr
```

Figure 4.3: Grammars of Simple Arithmetic

We take advantage of visualisation by replacing brackets with underlines, this should improve readability because reader can see a whole term in a compact way but still able check how they are bounded when needed, for example,

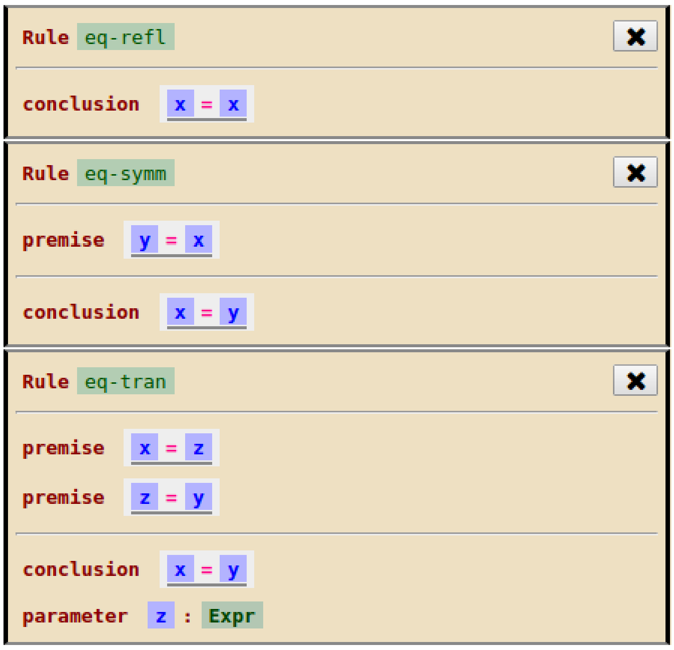
- `<Number>` 250 is transformed to `Number` `2 5 0`
- `<Expr>` $(12 + (0 \times 6))$ is transformed to `Expr` `1 2 + 0 × 6`
- `<Equation>` $(5 + 7) = 12$ is transformed to `Equation` `5 + 7 = 1 2`

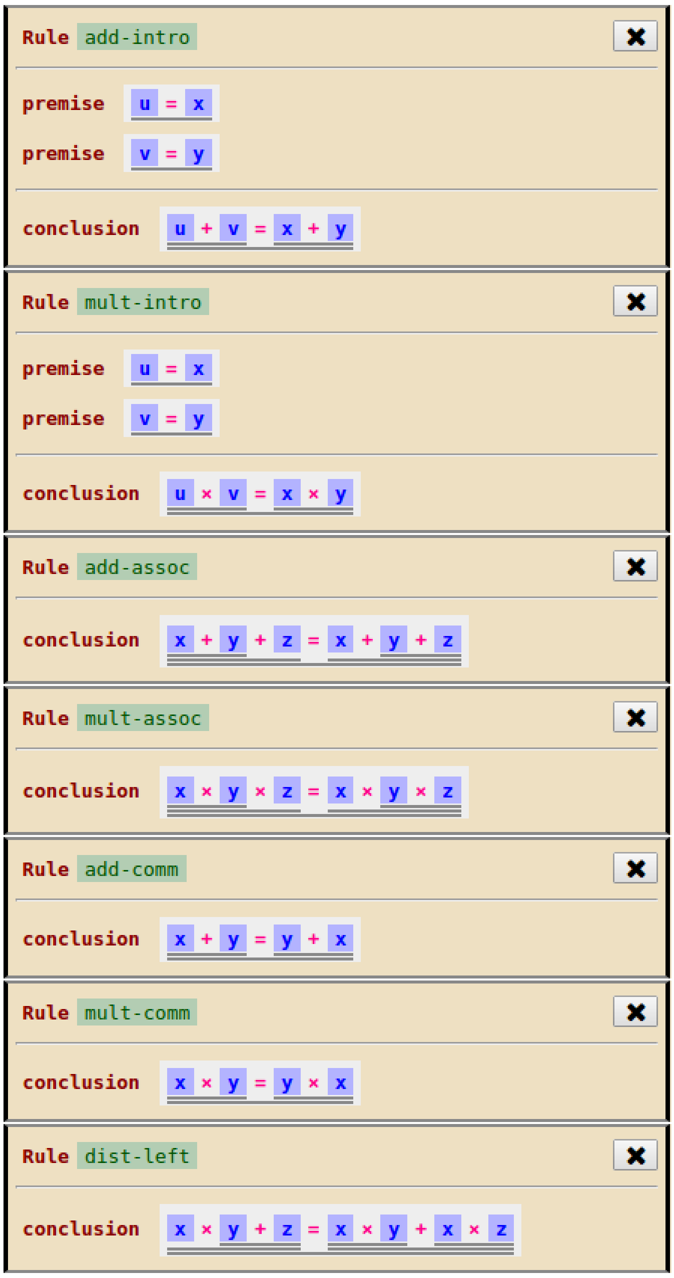
These underline patterns coincide with diagram in figures 3.2, 3.3, 3.5, and 3.6 respectively.

`metavar_regex` is used to control the name meta variables of each grammar. If this property is omitted, the corresponding grammar cannot instantiate meta variables. For example, `Expr` can instantiate meta variables with the name comply to regular expression `/[a-z][0-9]*/` (e.g. `a`, `b`, . . ., `z`, `a1`, `a2`, . . .), whereas `Digit` , `Number` , and `Equation` couldn't instantiate any meta variables, however, it could have meta variables as sub-term e.g. `Equation` `x + x = 2 × x`.

4.3 Rules

The derivation rules of Simple Arithmetic in figure 3.9 can be transformed as the following


```
Rule eq-refl
conclusion  x = x

Rule eq-symm
premise     y = x
conclusion  x = y

Rule eq-tran
premise     x = z
premise     z = y
conclusion  x = y
parameter   z : Expr
```



Rule add-intro

premise u = x

premise v = y

conclusion u + v = x + y

Rule mult-intro

premise u = x

premise v = y

conclusion u × v = x × y

Rule add-assoc

conclusion x + y + z = x + y + z

Rule mult-assoc

conclusion x × y × z = x × y × z

Rule add-comm

conclusion x + y = y + x

Rule mult-comm

conclusion x × y = y × x

Rule dist-left

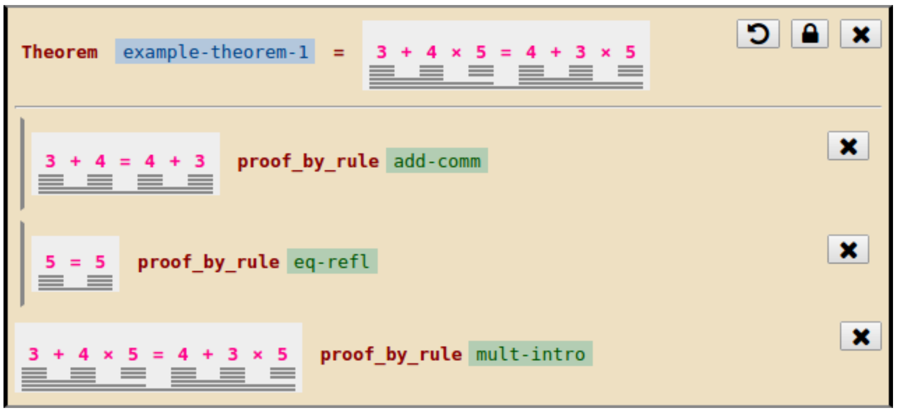
conclusion x × y + z = x × y + x × z

Figure 4.4: Rules of Simple Arithmetic

Most of rules here are self explain but in rule `eq-tran` , there is an additional property named `parameter` (s) which is a meta veritable that appear in premises but not in conclusion, hence user need to give a term when the rule is applied. Please note that `parameter` is automatic i.e. when user define they own rule, it will change automatically depending on premises and conclusion dist-right is not defined here but it will be defined as *lemma* in the next section.

4.4 Theorems and Lemmas

The first example of derivation tree (figure 3.11) could be transformed to theorem


Theorem example-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

3 + 4 = 4 + 3 proof_by_rule add-comm

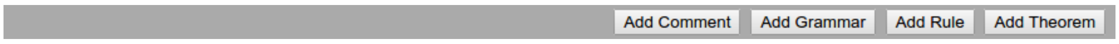
5 = 5 proof_by_rule eq-refl

3 + 4 × 5 = 4 + 3 × 5 proof_by_rule mult-intro

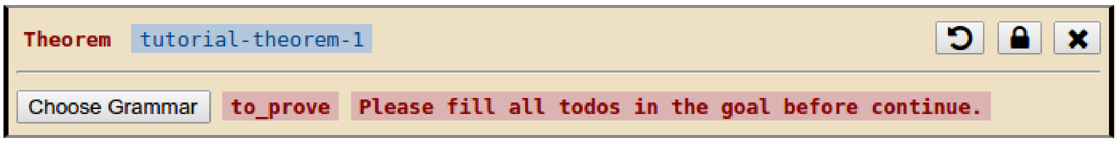
Figure 4.5: A theorem that show that $(3 + 4) \times 5 = (4 + 3) \times 5$.

You can see that the theorem still preserve tree-like structure but the width doesn't grow exponentially like derivation.

Next, I will show you that how was the theorem above constructed. Once we click module "Simple Arithmetic" on the repository pane, we will see the whole context similar to figure 4.2, you will also see that there are adding panel intersperse among each node

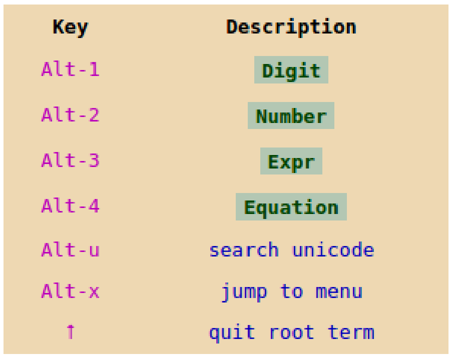


Now, click "Add Theorem", the button will change to input box where you can specify theorem name. Type "tutorial-theorem-1", can you will get empty theorem as the following


Theorem tutorial-theorem-1

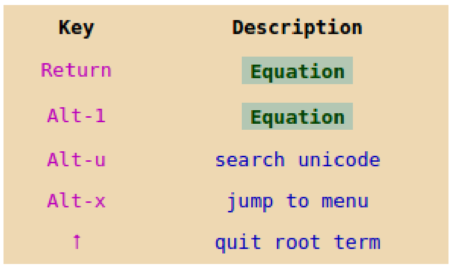
Choose Grammar to_prove Please fill all todos in the goal before continue.

The first thing that we can do is to construct the goal that will be proven. On the picture above you will see button labelled "Choose Grammar" which is, in fact, a term that doesn't know its grammar. We can specify grammar by click the button, which in-tern, will change to input box. Now the keymap pane will look like this



Key	Description
Alt-1	Digit
Alt-2	Number
Alt-3	Expr
Alt-4	Equation
Alt-u	search unicode
Alt-x	jump to menu
↑	quit root term

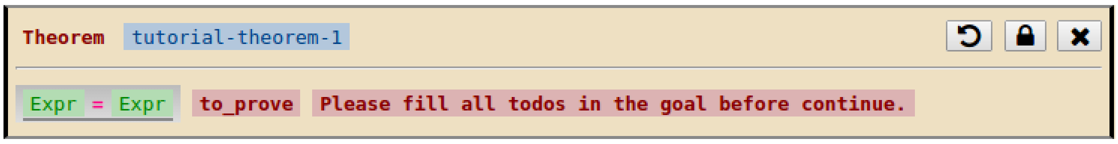
The keymap pane told us that there are 4 grammars available, we can either press `Alt-1..4` or click on the row in keymap pane directly to select grammar. Alternatively, you can search a grammars by type faction on it is input box e.g. "eq"



Key	Description
Return	Equation
Alt-1	Equation
Alt-u	search unicode
Alt-x	jump to menu
↑	quit root term

Now, it is only `Equation` available because it is the only one that has "eq" as (case-insensitively) sub-string. And since it is the only one, you can select it by press `Return` , even though in this case we don't have too many options but still benefit from it in auto complete favour. Please note that this input box support multiple matching separated by space e.g. "eq ti" still match `Equation` because both of "eq" and "ti" are sub-string of it.

Once you select a grammar, the box will change to green colour and waiting for a term of that grammar. In this case, we select `Equation` since it has just one choice and doesn't have meta variable or literal[2] so Phometa automatically click such a choice and the theorem will look like this

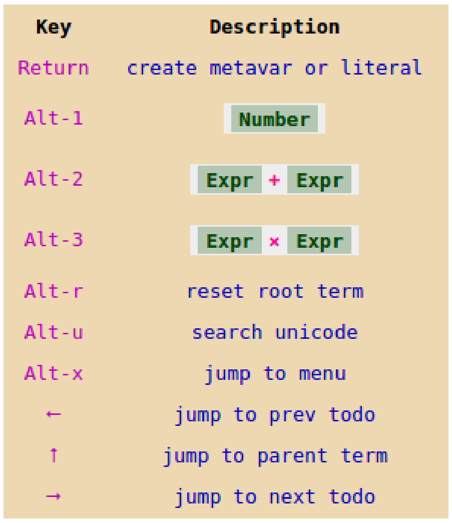


```
Theorem tutorial-theorem-1

Expr = Expr  to_prove  Please fill all todos in the goal before continue.
```

You may notice that the goal term as grey background rather than white as before. This indicate that the term is still modifiable.

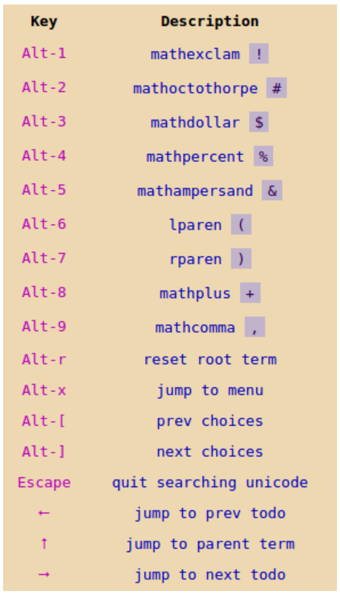
Next, we will continue on the `Expr` term on the left hand side of "=". When the cursor is in it, the keymap will look like this

[2]literal is similar to meta-variable but only match to itself, will be explained in later chapter



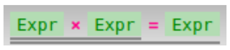
Key	Description
Return	create metavar or literal
Alt-1	Number
Alt-2	Expr + Expr
Alt-3	Expr × Expr
Alt-r	reset root term
Alt-u	search unicode
Alt-x	jump to menu
←	jump to prev todo
↑	jump to parent term
→	jump to next todo

Again, there are 3 choice available which can be selected similar manner when we select grammar. At the this stage you might wonder how to type "×" since it is unicode character. Well, we can go to unicode mode by press `Alt-u` as keymap pane suggest. Then keymap pane will look like this



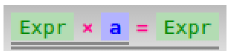
Key	Description
Alt-1	mathexclam !
Alt-2	mathoctothorpe #
Alt-3	mathdollar $
Alt-4	mathpercent %
Alt-5	mathampersand &
Alt-6	lparen (
Alt-7	rparen)
Alt-8	mathplus +
Alt-9	mathcomma ,
Alt-r	reset root term
Alt-x	jump to menu
Alt-[	prev choices
Alt-]	next choices
Escape	quit searching unicode
←	jump to prev todo
↑	jump to parent term
→	jump to next todo

This allow us to search unicode character by using its LaTeX's math-mode name. Now type "times" in the input box, you should see "×" appear on keymap. Once you select it, the unicode mode disappear and put "×" in the input box, which in-tern, filter other choices out so you can hit `Return` for multiplication. The goal will transform to



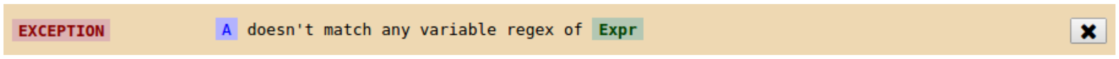
Expr × Expr = Expr

Next we will focus middle `Expr` . If we type string and hit `Return` here it will assume that we enter meta variable or literal (to avoid conflict auto complete similar to choose grammar is disabled here). E.g. if we type "a" and press enter it will become like this



```
Expr × a = Expr
```

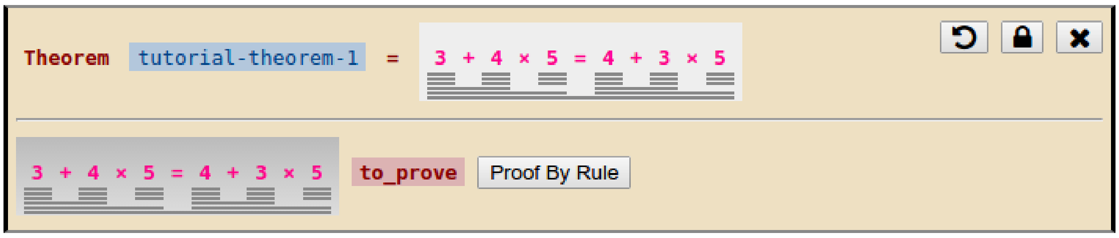
If we enter the name that that doesn't comply to regular expression, it will do nothing and prompt error message above main pane as the following



```
EXCEPTION        A doesn't match any variable regex of Expr
```

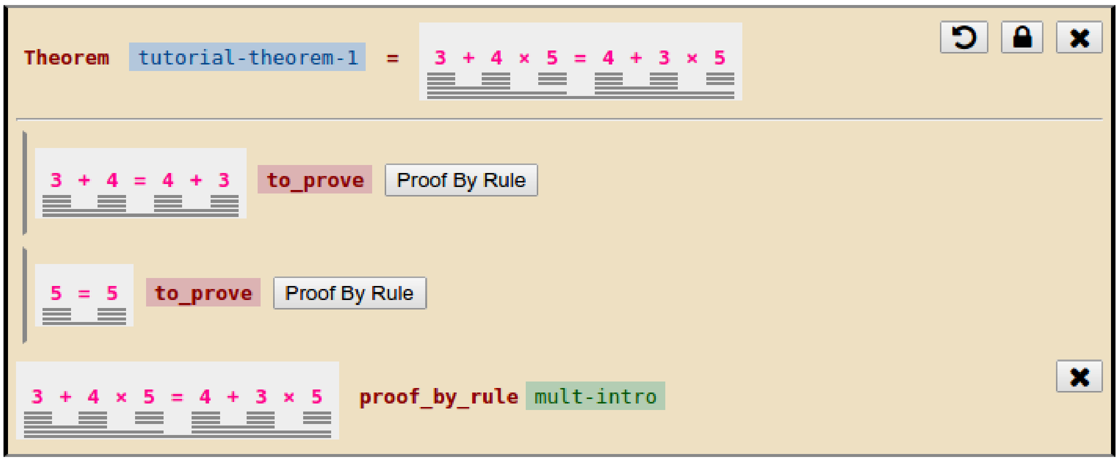
By the way, the goal here doesn't involve any meta variable. We can reset any sub-term (e.g. in this case `a`) by pressing `Alt-t` . Ultimately, you can reset the whole term by pressing `Alt-r` . In addition, you can jump to parent term by pressing `UP` , this is particularly useful when combine with `Alt-t` i.e. you can reset parent term only by keystrokes rather than clicking. You also be able to jump to previous todo or next todo by pressing `LEFT` or `RIGHT` respectively.

By recursively fill the the goal, eventually it will become like this



```
Theorem tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

3 + 4 × 5 = 4 + 3 × 5   to_prove   Proof By Rule
```

Since the goal is complete, it is ready to be proven. You can select a rule by clicking "Proof By Rule" and select `mult-intro` similar manner to choose grammar. Then the theorem will look like this

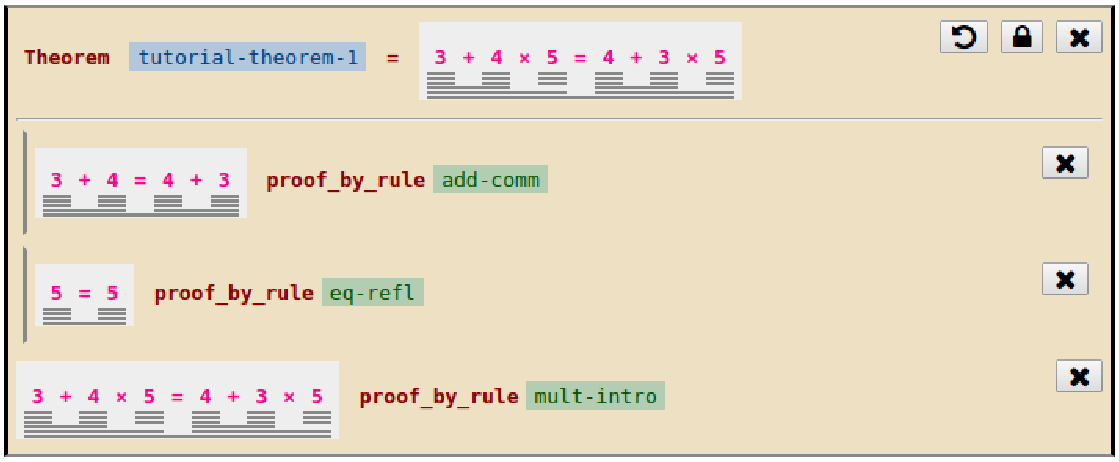


```
Theorem tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

  3 + 4 = 4 + 3   to_prove   Proof By Rule

  5 = 5   to_prove   Proof By Rule

3 + 4 × 5 = 4 + 3 × 5   proof_by_rule mult-intro
```

Once the rule is applied, it will generate further sub-goals.

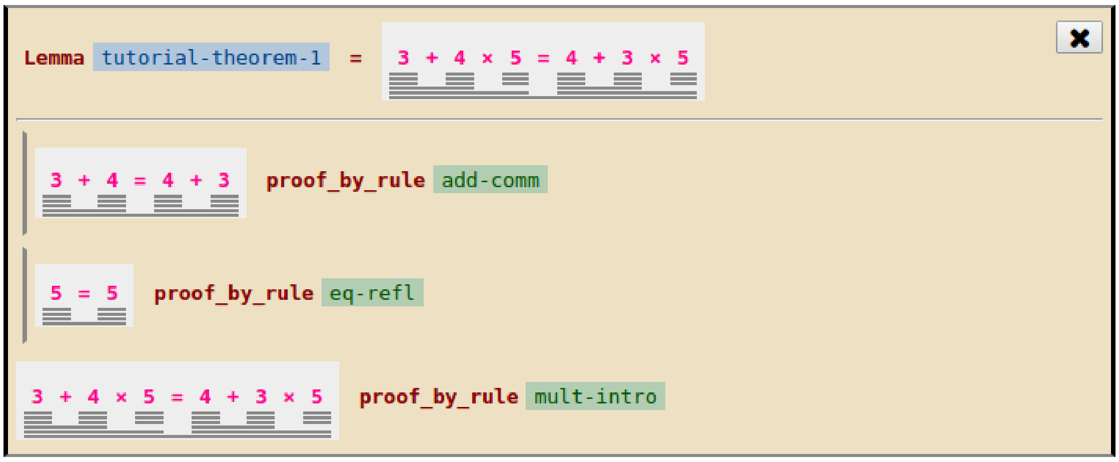
Please notice that the goal background changes to white colour as we can longer modify the goal. However if you made a mistake and want to go back, you can click close button on the bottom right corner of current proof (or press `Alt-t`) to reset current proof then you can modify it again. Ultimately, you can click reset button on the top right corner of the theorem (or press `Alt-r`) to reset entire theorem.

Two remaining sub-goals that have been generated can be proven similar to the process above i.e. select rule `add-comm` for first sub goal and `eq-refl` for second one.



```
Theorem tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

  3 + 4 = 4 + 3  proof_by_rule add-comm
  5 = 5  proof_by_rule eq-refl
3 + 4 × 5 = 4 + 3 × 5  proof_by_rule mult-intro
```

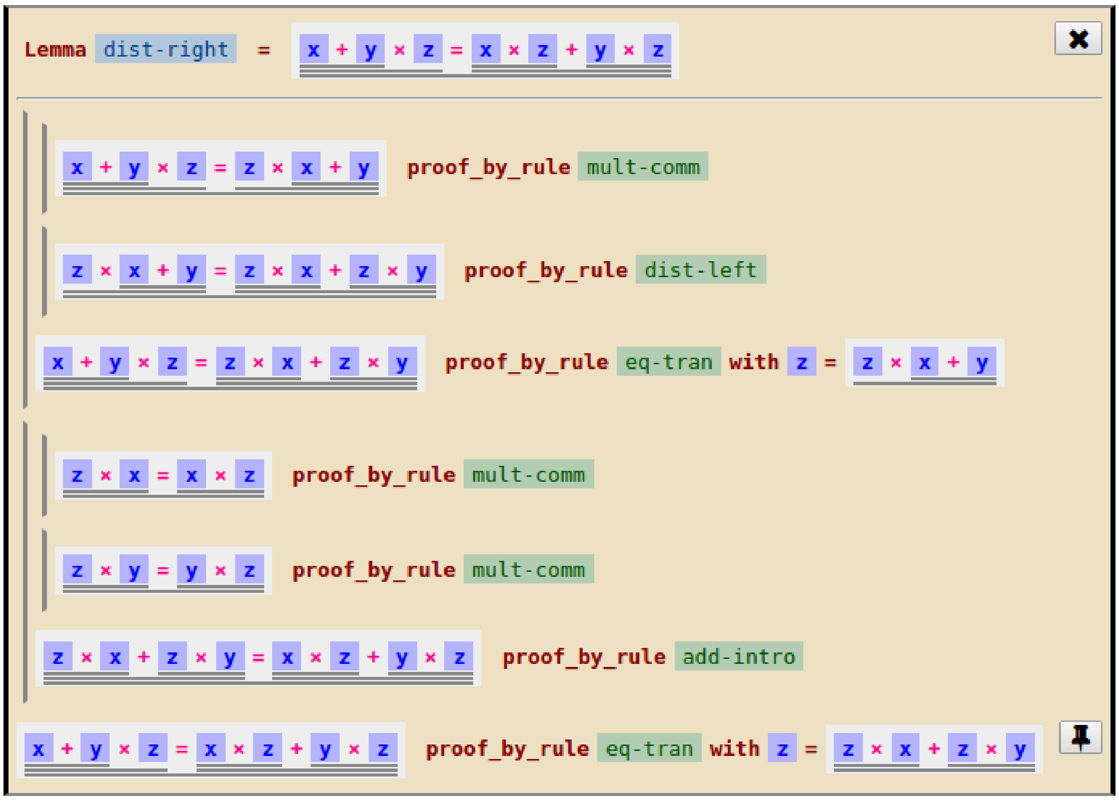
Once the theorem is complete, you can claim validity of the goal. More over you can convert it to lemma that can be used in later theorem by clicking lock button on top-right corner of theorem



```
Lemma tutorial-theorem-1 = 3 + 4 × 5 = 4 + 3 × 5

  3 + 4 = 4 + 3  proof_by_rule add-comm
  5 = 5  proof_by_rule eq-refl
3 + 4 × 5 = 4 + 3 × 5  proof_by_rule mult-intro
```

Because other theorem can use this lemma so it is no longer modifiable, as you can see that close button of each sub-proof and reset button of the main theorem are gone.

Similarly, we can can create lemma `dist-right` as the following



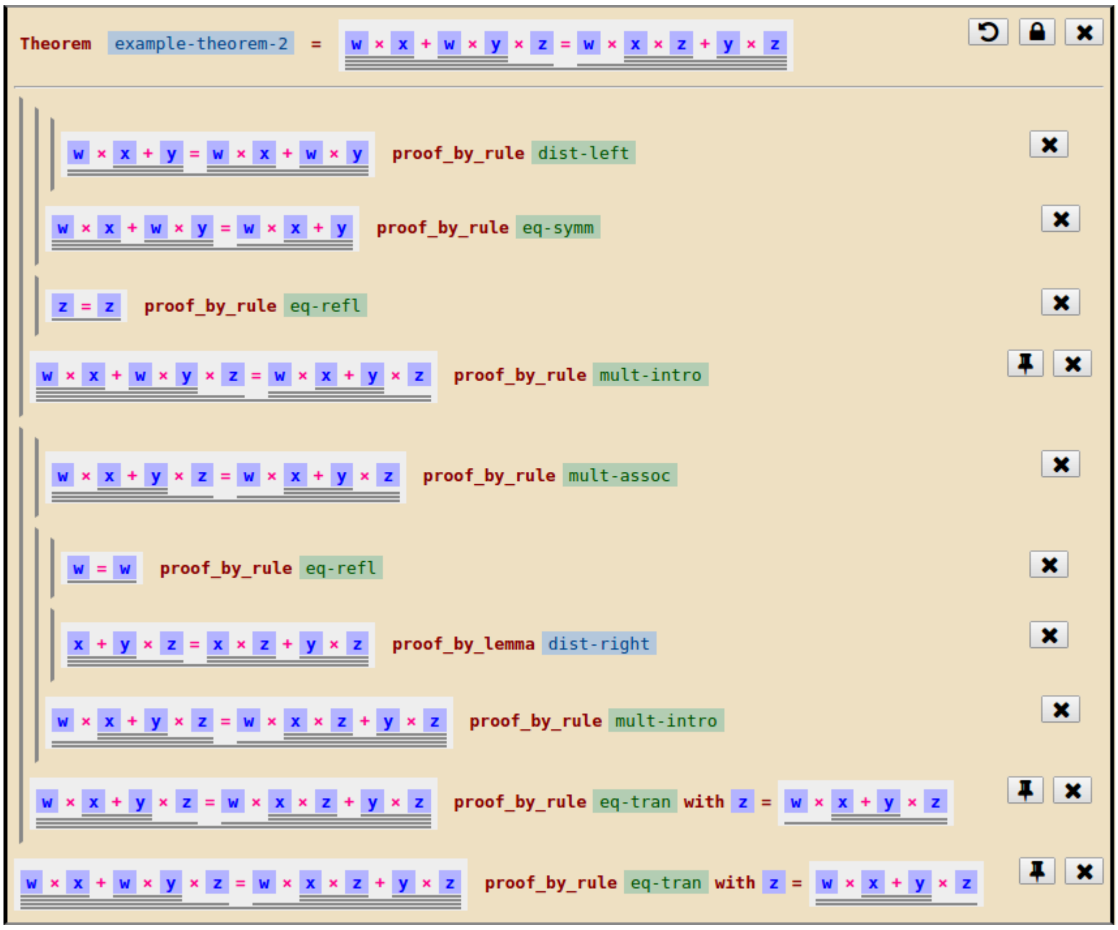
```
Lemma dist-right = x + y × z = x × z + y × z

    x + y × z = z × x + y              proof_by_rule mult-comm
    z × x + y = z × x + z × y          proof_by_rule dist-left
  x + y × z = z × x + z × y            proof_by_rule eq-tran with z = z × x + y
    z × x = x × z                      proof_by_rule mult-comm
    z × y = y × z                      proof_by_rule mult-comm
  z × x + z × y = x × z + y × z        proof_by_rule add-intro
x + y × z = x × z + y × z              proof_by_rule eq-tran with z = z × x + z × y
```

Figure 4.6: An example of lemma obtained by lock a theorem.

It is a good practice to create lots of small lemmas rather than a big theorem. This is because it is easier to read and you can use a lemma multiple time i.e. no need to duplicate sub-proof.

4.5 More complex theorem

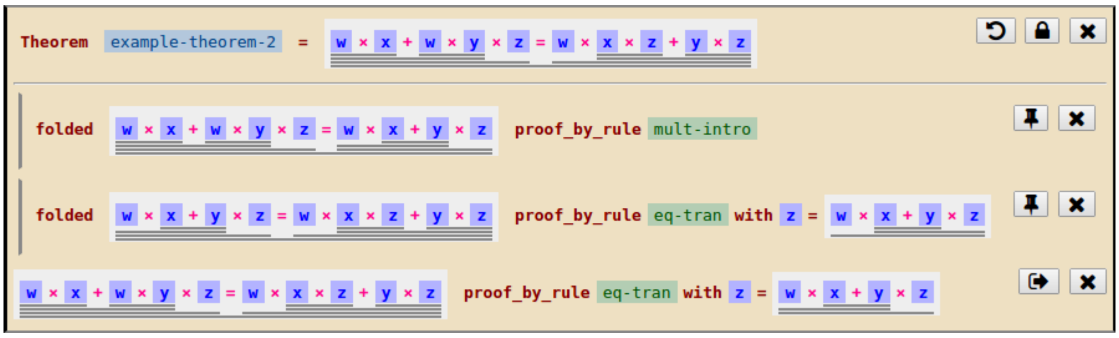
To gain more familiarly on theorem, here is more complex theorem corresponded the second example of derivation tree on figure 3.12



```
Theorem example-theorem-2 = w × x + w × y × z = w × x × z + y × z
w × x + y = w × x + w × y proof_by_rule dist-left
w × x + w × y = w × x + y proof_by_rule eq-symm
z = z proof_by_rule eq-refl
w × x + w × y × z = w × x + y × z proof_by_rule mult-intro
w × x + y × z = w × x + y × z proof_by_rule mult-assoc
w = w proof_by_rule eq-refl
x + y × z = x × z + y × z proof_by_lemma dist-right
w × x + y × z = w × x × z + y × z proof_by_rule mult-intro
w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z
w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z
```

Figure 4.7: The second example theorem of Simple Arithmetic.

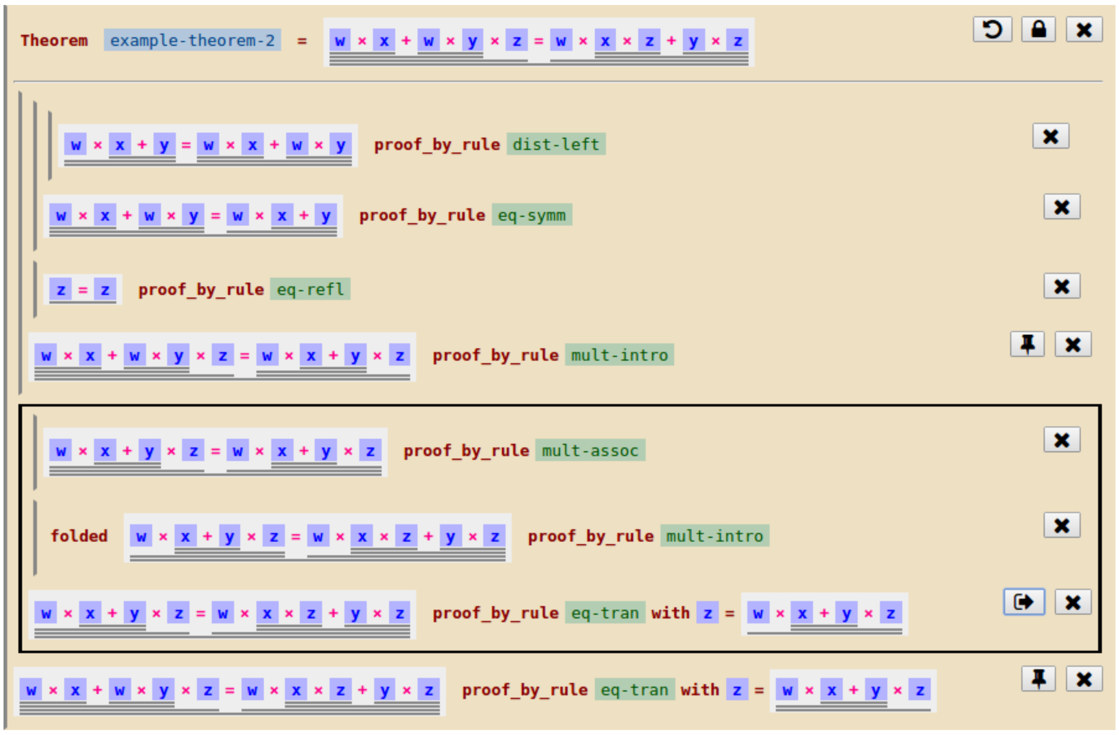
Sub-proofs become more complex and premises of main proof are far away which is harder to read, to avoid this kind of problem, we introduce a focus button that will fold sub-proof of corresponding proof. For example, if we click focus button on the main proof it will look like this



```
Theorem example-theorem-2 = w × x + w × y × z = w × x × z + y × z
folded w × x + w × y × z = w × x + y × z proof_by_rule mult-intro
folded w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z
w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z
```

Two sub-proofs of the main theorem are folded, this allow us to read rule instance of eq-tran easily. You can unfolded sub-proofs by clicking unfocus button at the same position that focus button was there before. And the theorem will look the same as figure 4.7 again.

When you read some proof in Phometa, it is a good idea to click focus on the main proof first so you can read the main rule instance easily. Once you understand main proof you can read one of sub proof by clicking focus button that correspond to that sub-proof. If you click focus button on the first sub-proof it will look like this



Theorem example-theorem-2 = w × x + w × y × z = w × x × z + y × z

w × x + y = w × x + w × y proof_by_rule dist-left

w × x + w × y = w × x + y proof_by_rule eq-symm

z = z proof_by_rule eq-refl

w × x + w × y × z = w × x + y × z proof_by_rule mult-intro

w × x + y × z = w × x + y × z proof_by_rule mult-assoc

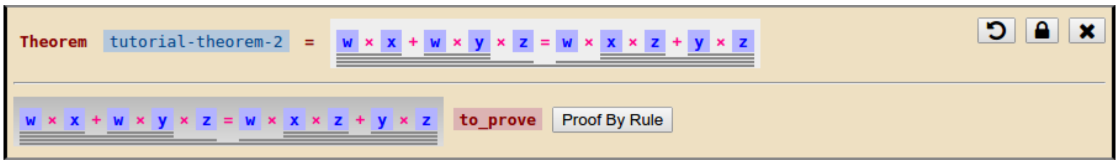
folded w × x + y × z = w × x × z + y × z proof_by_rule mult-intro

w × x + y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

w × x + w × y × z = w × x × z + y × z proof_by_rule eq-tran with z = w × x + y × z

This process automatically unfold the previous one before folding sub-proof of this proof again. Please note that some of deeper proofs might not have focus button at all, this is because its sub-proofs cannot reduce further than original one.

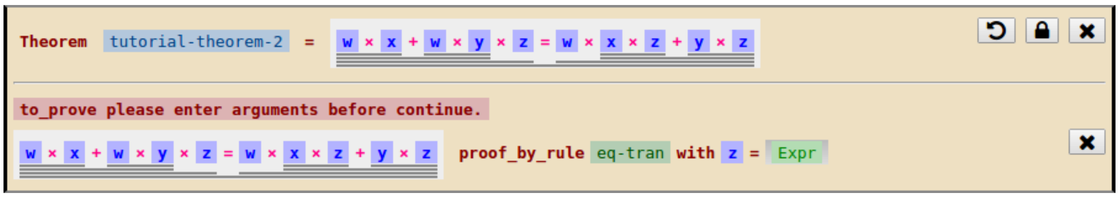
The next thing that I will show are how to use rule parameters and lemma. This can be illustrated by recreate this theorem again. First create create a theorem tutorial-theorem-2 using the same goal as above theorem.



Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

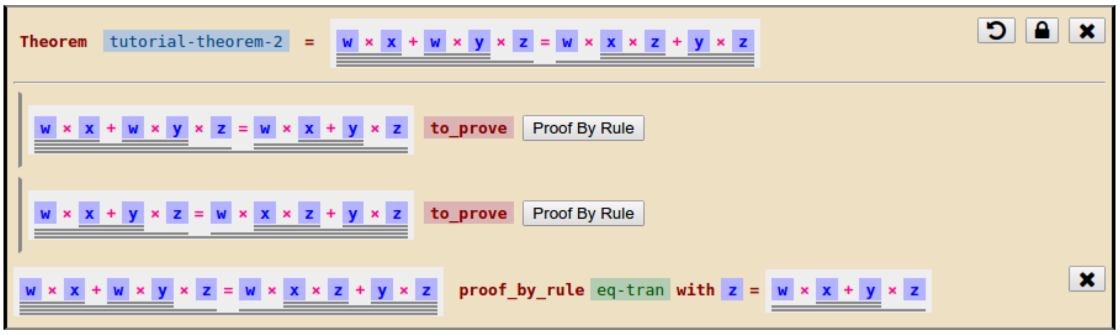
w × x + w × y × z = w × x × z + y × z to_prove Proof By Rule

Then apply rule eq-tran to this goal



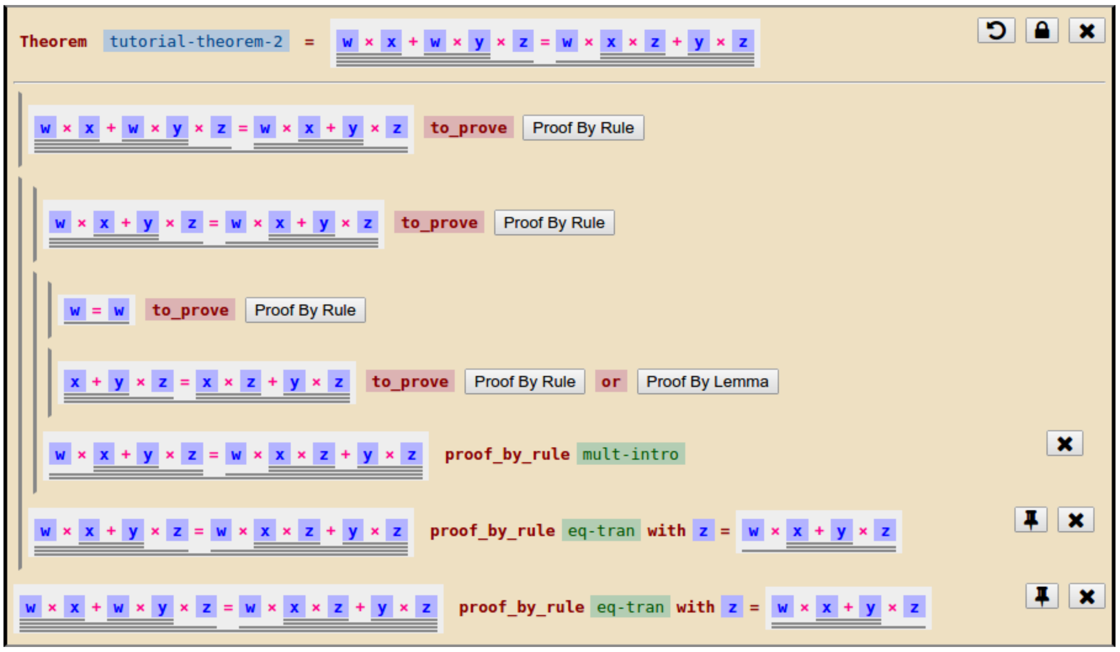
```
Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

to_prove please enter arguments before continue.
w × x + w × y × z = w × x × z + y × z   proof_by_rule eq-tran with z = Expr
```

The rule applying process is not complete because `eq-tran` contains `z` which appear in premises but not in conclusion (i.e. `z` is parameter) so Phometa ask us to fill the term that we want to use. In this case we want `w × x + y × z` so put it there



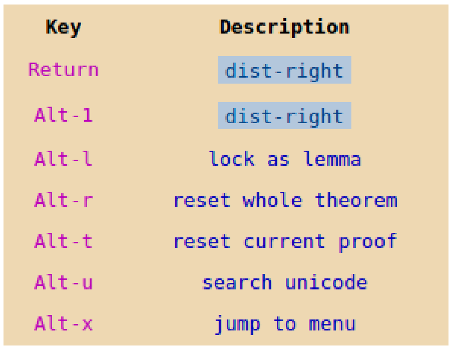
```
Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

  w × x + w × y × z = w × x + y × z   to_prove   Proof By Rule

  w × x + y × z = w × x × z + y × z   to_prove   Proof By Rule

w × x + w × y × z = w × x × z + y × z   proof_by_rule eq-tran with z = w × x + y × z
```

Now, let focus on the second sub-goal, we can apply `eq-tran` again but with `z` = `w × x + y × z`. Then, apply `mult-intro` in the second its sub-goal.



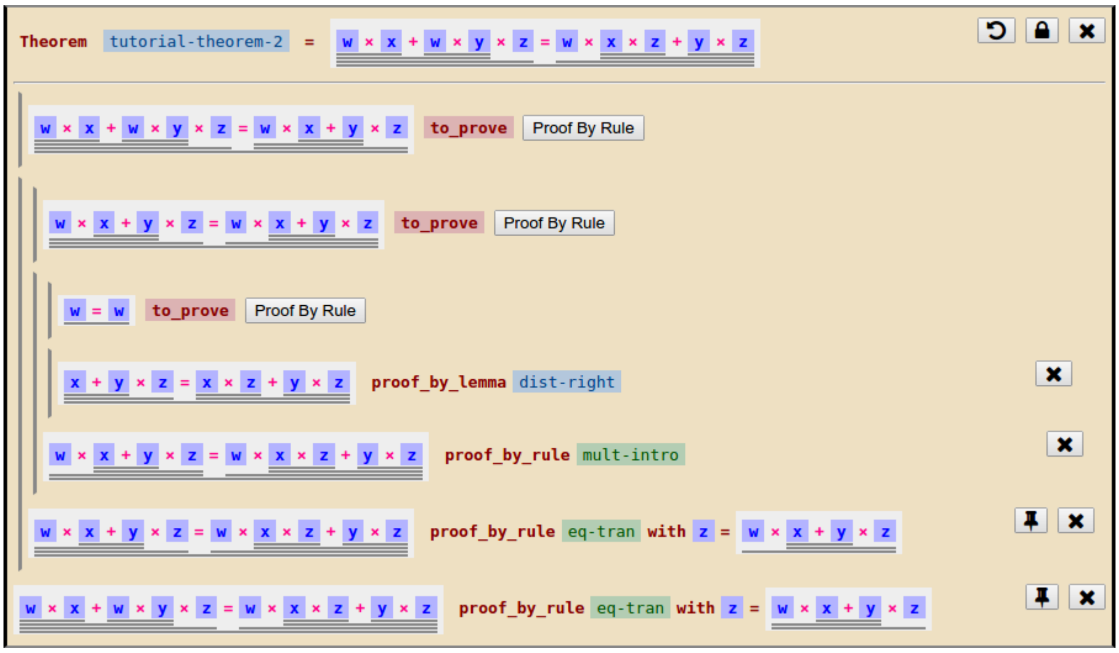
```
Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z

  w × x + w × y × z = w × x + y × z   to_prove   Proof By Rule

    w × x + y × z = w × x + y × z   to_prove   Proof By Rule

      w = w   to_prove   Proof By Rule

      x + y × z = x × z + y × z   to_prove   Proof By Rule   or   Proof By Lemma

    w × x + y × z = w × x × z + y × z   proof_by_rule mult-intro

  w × x + y × z = w × x × z + y × z   proof_by_rule eq-tran with z = w × x + y × z

w × x + w × y × z = w × x × z + y × z   proof_by_rule eq-tran with z = w × x + y × z
```

You can see that there is a sub-goal that has button "Proof By Lemma". This is because there is at least one lemma that is pattern matchable with that sub-goal. If you click "Proof By Lemma" button, the keymap will look like this



Key	Description
Return	dist-right
Alt-1	dist-right
Alt-l	lock as lemma
Alt-r	reset whole theorem
Alt-t	reset current proof
Alt-u	search unicode
Alt-x	jump to menu

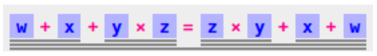
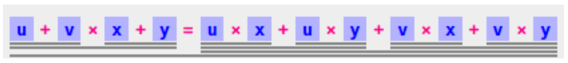
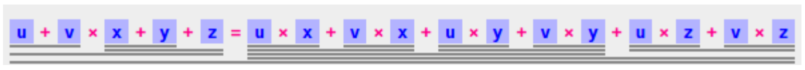
In this case, there is one lemma which is `dist-right` that is pattern matchable to that sub-goal. You can hit `Return` to use this lemma and it will look like this



```
Theorem tutorial-theorem-2 = w × x + w × y × z = w × x × z + y × z
  w × x + w × y × z = w × x + y × z   to_prove   Proof By Rule
    w × x + y × z = w × x + y × z   to_prove   Proof By Rule
      w = w   to_prove   Proof By Rule
      x + y × z = x × z + y × z   proof_by_lemma dist-right
    w × x + y × z = w × x × z + y × z   proof_by_rule mult-intro
  w × x + y × z = w × x × z + y × z   proof_by_rule eq-tran with z = w × x + y × z
w × x + w × y × z = w × x × z + y × z   proof_by_rule eq-tran with z = w × x + y × z
```

The remaining step is easy enough.

4.6 Exercises

- Create a theorem and proof each of the following
  - 
    $w + x + y \times z = z \times y + x + w$
  - 
    $u + v \times x + y = u \times x + u \times y + v \times x + v \times y$
  - 
    $u + v \times x + y + z = u \times x + v \times x + u \times y + v \times y + u \times z + v \times z$
- Create a theorem of your own choice and proof it.
- (Challenge) Extend Simple Arithmetic to support the following
  - addition and multiplication identity.
  - addition and multiplication idempotent.
  - inequality.

  You may need to create a new grammar or rules for this, please see section 5.7 in the next chapter.

Chapter 5

Example Formal System — Propositional Logic

Once you are familiar with some basic features and usability of *Phometa* from the last chapter. This chapter aims to show more advance features on another formal system named *Propositional Logic* which is the most well known logical system[1].

Logic, in general, works so well with traditional derivation system, hence there is spacial name called *Natural Deduction* which is a combination of any kind of Logic together with derivation system.

5.1 Grammars

As usual, the first thing that needed to be defined grammars. Propositional Logic has 4 grammars which are `Prop` , `Atom` , `Context` , and `Judgement` as its grammars.

`Prop` is a proposition, semantically, it is a term that can be evaluated to either true or false (given that there are no meta variables in the term). Grammars of `Prop` can be defined in Phometa as the following

[1]Logical system is a formal system together with semantics[19]

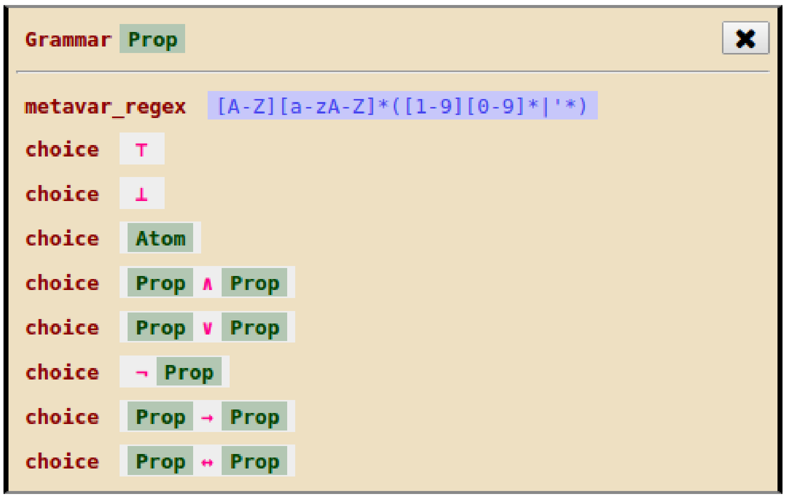

Grammar Prop
metavar_regex [A-Z][a-zA-Z]*([1-9][0-9]*|'*)
choice ⊤
choice ⊥
choice Atom
choice Prop ∧ Prop
choice Prop ∨ Prop
choice ¬ Prop
choice Prop → Prop
choice Prop ↔ Prop

Figure 5.1: Definition of Prop

This grammar is equivalence to the following Backus Normal Form

```
<Prop> ::= ⊤ | ⊥ | <Atom>
         | <Prop> ∧ <Prop>
         | <Prop> ∨ <Prop>
         | ¬ <Prop>
         | <Prop> → <Prop>
         | <Prop> ↔ <Prop>
         | meta-variables comply with regex
           /[A-Z][a-zA-Z]*([1-9][0-9]*|'*)/
```

Figure 5.2: Backus-Naur Form correspond to Prop

On the 3rd choice of Prop depends on Atom which represents primitive truth statement that cannot be broken down any further. It can be defined in Phometa as the following.

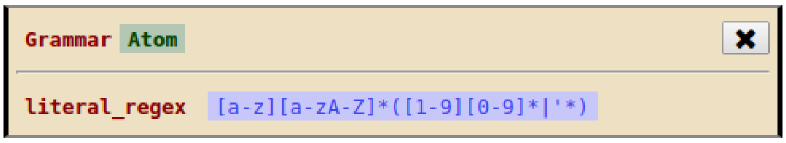

Grammar Atom
literal_regex [a-z][a-zA-Z]*([1-9][0-9]*|'*)

Figure 5.3: Definition of Atom

You can see that literal_regex appears inside Atom definition, this allows Atom be instantiated by literal which is similar to meta variable, the only different is that literal doesn't have ability to be substituted by arbitrary term like meta variable.

At this stage, you might wonder that why `Prop` needs both of meta variables and `Atom` . Well, meta variable will be used when referring to something general and `Atom` will used when referring to particular truth statement. For example, if we can prove that `A ∨ ¬ A` is valid, then terms such as `⊤ ∨ ¬ ⊤`, `⊤ → ⊥ ∨ ¬ ⊤ → ⊥`, `B ∨ ¬ B`, `rainning ∨ ¬ rainning`, and `B ∧ rainning ∨ ¬ B ∧ rainning` are also valid. However, if we can prove that `rainning ∨ ¬ rainning` is valid, we don't want to expose this proof to other terms since it might be proven from specific knowledge.

Now, we have enough ingredient to create a proper proposition, one might say that we can start proving it directly, however, most of proposition that we will dealing with only holds under certain assumptions, hence, a *judgement* should be in the form $A_1, A_2, ..., A_n \vdash B$ where $A_{1..n}$ are assumptions and B is conclusion.

To model a judgement in Phometa, first we need to model assumptions or in the other name, `Context` as the following

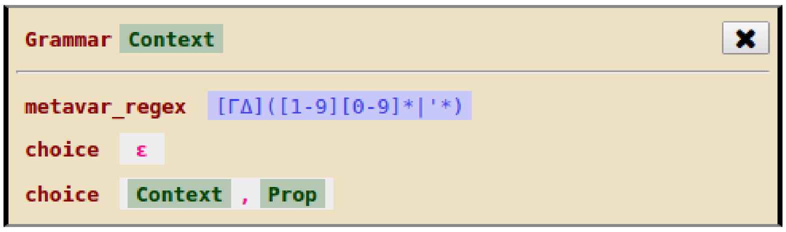

```
Grammar Context
metavar_regex  [ΓΔ]([1-9][0-9]*|'*)
choice  ε
choice  Context , Prop
```

Figure 5.4: Definition of `Context`

So a term of `Context` can be either empty context or another context appended by a proposition. We can see a context as a list of proposition.

Now we are ready to define `Judgement` as the following

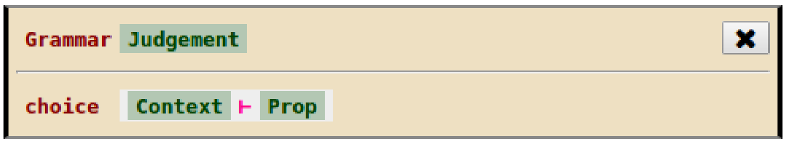

```
Grammar Judgement
choice  Context ⊢ Prop
```

Figure 5.5: Definition of `Judgement`

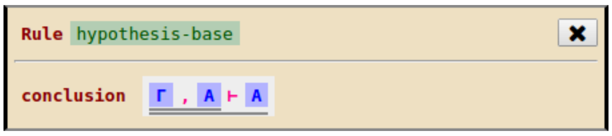
`Judgement` has a meaning of validity. For example, validity of `ε , p , q ∨ r ⊢ p ∧ q ∨ p ∧ r` means, assuming that `p` and `q ∨ r` hold then `p ∧ q ∨ p ∧ r` holds.

Please note that `Judgement` doesn't have field `metavar_regex` nor `literal_regex` so we can't accidentally use meta variable or literal for `Judgement` .

5.2 Hypothesis rules

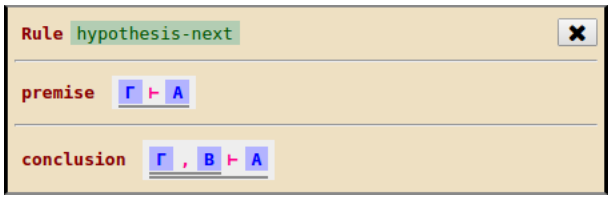
In order to prove any judgement, we want ability to state that for any proposition that is in assumptions, it can be conclusion i.e. $A_{1..n} \vdash A_i$ where $i \in 1..n$

This is achievable by `hypothesis-base` and `hypothesis-next` .



Rule hypothesis-base

conclusion Γ , A ⊢ A

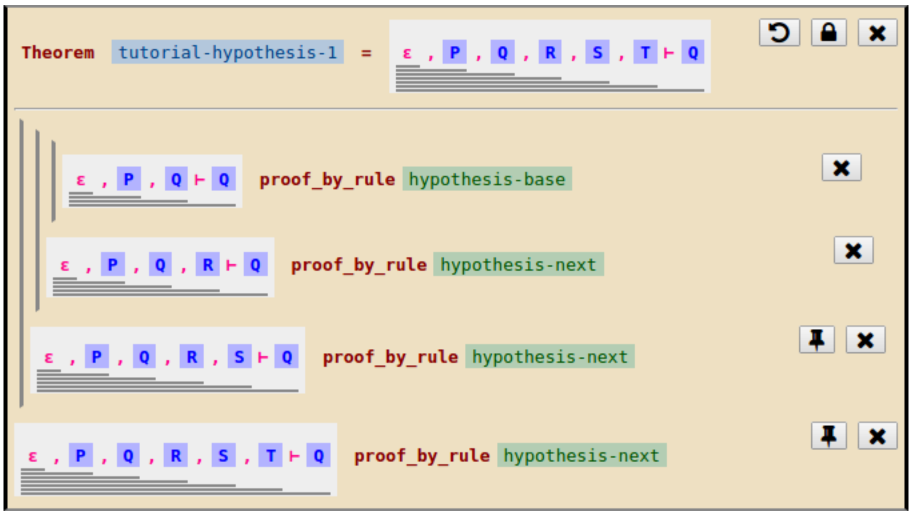


Rule hypothesis-next

premise Γ ⊢ A

conclusion Γ , B ⊢ A

`hypothesis-base` matches the last assumption with the conclusion whereas `hypothesis-next` removes the last assumption and pass on to a premise, this can prove a judgement that has conclusion as assumption like this.



Theorem tutorial-hypothesis-1 = ε , P , Q , R , S , T ⊢ Q

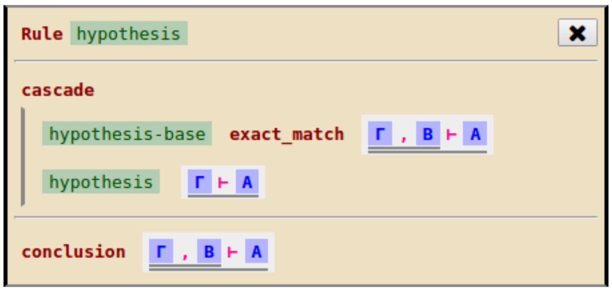
ε , P , Q ⊢ Q proof_by_rule hypothesis-base

ε , P , Q , R ⊢ Q proof_by_rule hypothesis-next

ε , P , Q , R , S ⊢ Q proof_by_rule hypothesis-next

ε , P , Q , R , S , T ⊢ Q proof_by_rule hypothesis-next

This is not efficient as we might need to call `hypothesis-next` $(n-1)$ times where n is the number of assumptions. To solve this problem, we introduce `hypothesis` that is more complex than ordinary derivation rule.



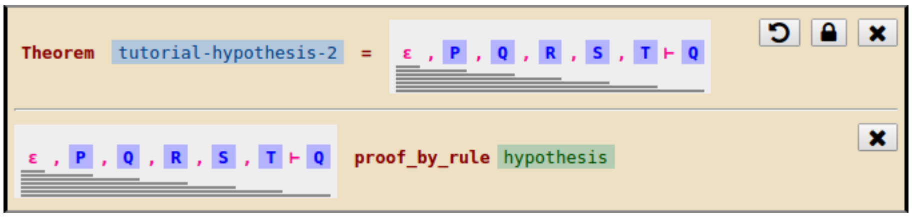
```
Rule hypothesis
cascade
    hypothesis-base exact_match Γ , B ⊢ A
    hypothesis Γ ⊢ A
conclusion Γ , B ⊢ A
```

Figure 5.6: rule Hypothesis

hypothesis uses a cascade premise instead of direct premise. A cascade premise has several sub-rules calling-template that will be tried in order. In this case, hypothesis tries to apply hypothesis-base on its goal,

- If sub-rule is applicable, then use sub-goals generated from sub-rule as its sub-goals, in this case, hypothesis-base doesn't have any premises so this cascade premise has no further sub-goals.
- Otherwise, *cascades* down and tries to apply the next sub-rule which is hypothesis itself[2]. Again, if applicable, use sub-goals of sub-rule, otherwise, the main hypothesis rule fail as the cascade premise fail to match with any ofsub-rules.

hypothesis could solve the last theorem like this



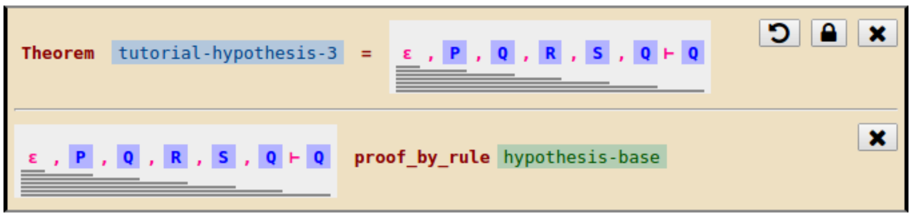
```
Theorem tutorial-hypothesis-2 = ε , P , Q , R , S , T ⊢ Q
ε , P , Q , R , S , T ⊢ Q proof_by_rule hypothesis
```

To show the process, first hypothesis conclusion — Γ , B ⊢ A is pattern match against goal = ε , P , Q , R , S , T ⊢ Q, this results in A = Q, B = T, and Γ = ε , P , Q , R , S. This cascade premise try to apply hypothesis-base with goal[3] = ε , P , Q , R , S , T ⊢ Q.

hypothesis-base , as sub-rule, let Γ , A ⊢ A pattern match against ε , P , Q , R , S , T ⊢ Q and get A = Q, A = T, and Γ = ε , P , Q , R , S. When a meta variable is matched

[2]Yes, it supports recursive call
[3]This is result from substitution to that sub-rule goal template. Coincidentally, it is the same as hypothesis conclusion

with two or more terms, those terms will be unified to make pattern match still possible. So T will be replaced by Q and this sub-rule will success. Here is the same result if hypothesis-base is applied directly on the theorem.



```
Theorem tutorial-hypothesis-3 = ε , P , Q , R , S , Q ⊢ Q

ε , P , Q , R , S , Q ⊢ Q   proof_by_rule hypothesis-base
```

However, this is not what we want, to avoid this problem, we cat put flag exact_match to this sub-rule, this flag will prevent further unification. Now, T cannot unify with Q so this sub-rule fail. So, the cascade premise will move to second sub-rule and try to apply hypothesis with ε , P , Q , R , S ⊢ Q.

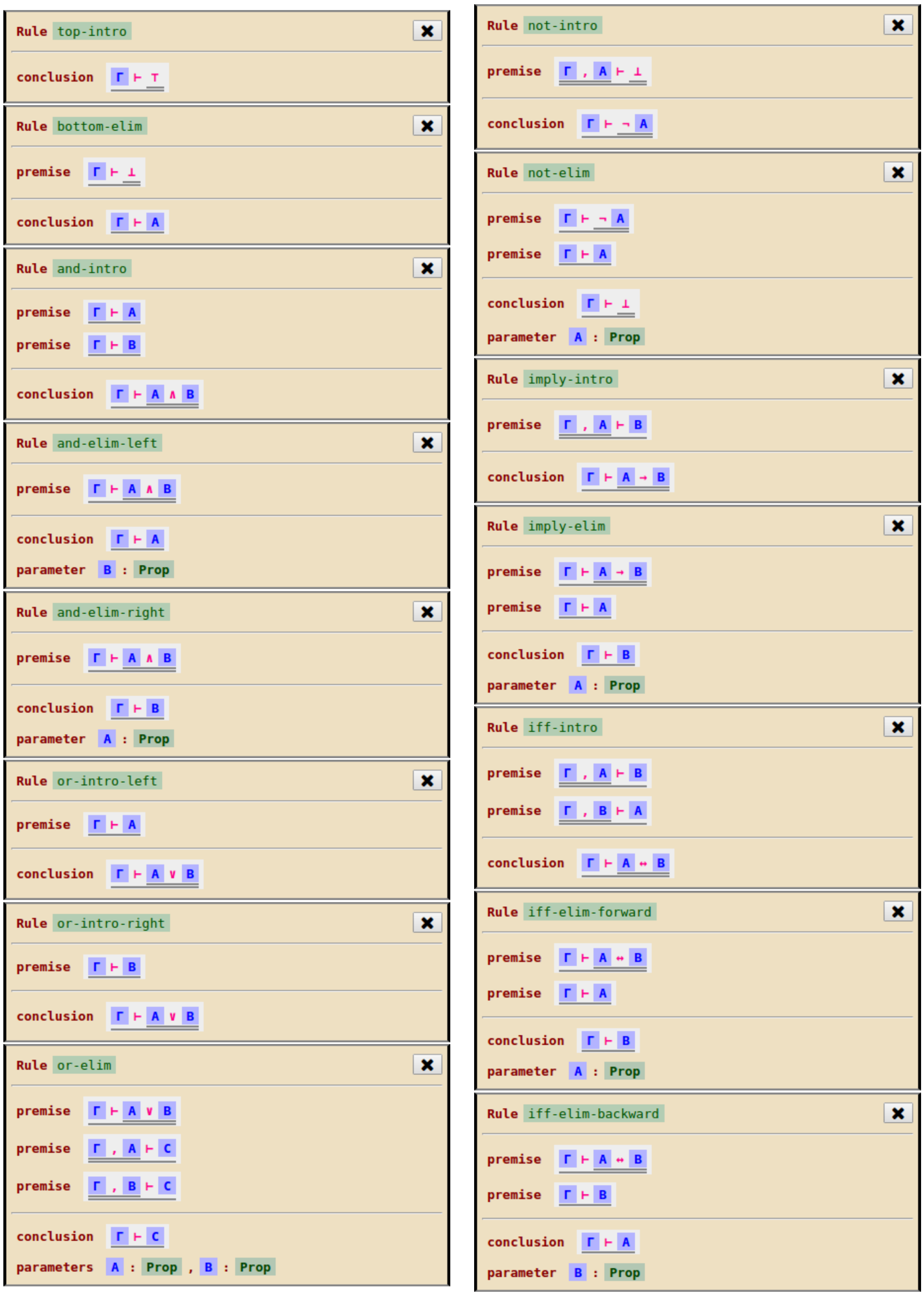
hypothesis , as sub-rule, let Γ , B ⊢ A pattern match against ε , P , Q , R , S ⊢ Q, the process is similar as before so I can tell directly that it will apply hypothesis as sub-sub-rule with goal = ε , P , Q , R ⊢ Q, which in tern, apply hypothesis as sub-sub-sub-rule with goal = ε , P , Q ⊢ Q.

Now, hypothesis-base with goal = ε , P , Q ⊢ Q will not fail again since the last assumption and the conclusion is exactly match, i.e. no further unification needed hence it will success and return no sub-goals as hypothesis-base doesn't have any. This success will propagate up to the top level and the entire will cascade success with no further sub-goals as shown in tutorial-hypothesis-2 .

Please note that a cascade premise is just another type of a premise, sub-goals that are generated from sub-rule will replace the cascade premise itself, similar to how a sub-goal replaces direct premise. Thus, cascade premise can be used alongside with direct premises, for more information on cascade blocks please see specification chapter.

5.3 Main Rules

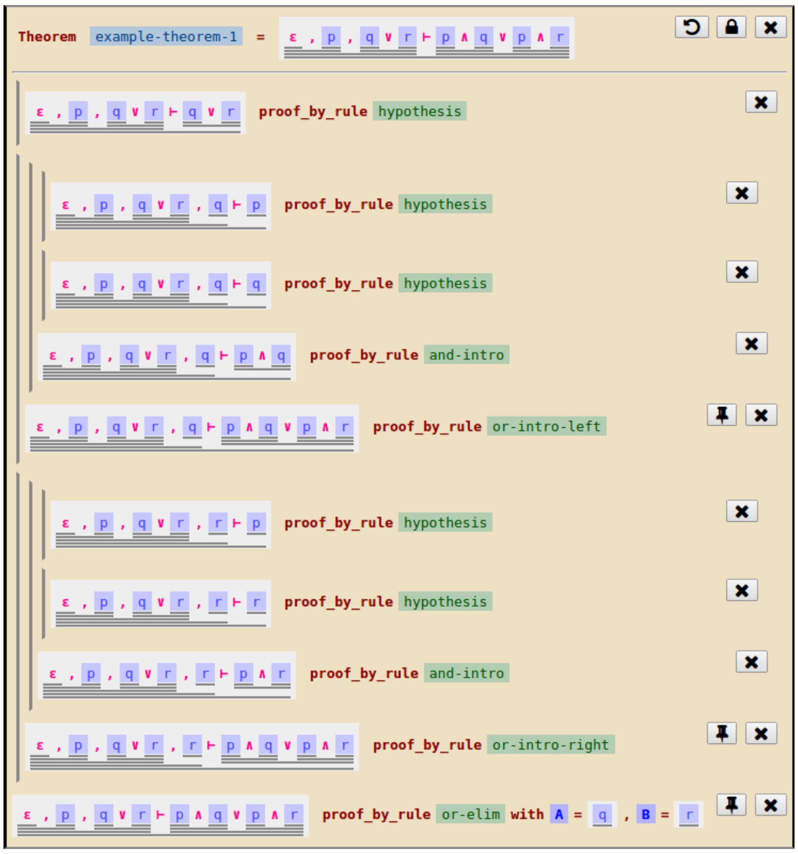
Now, Propositional Logic is ready for new rules as the following,



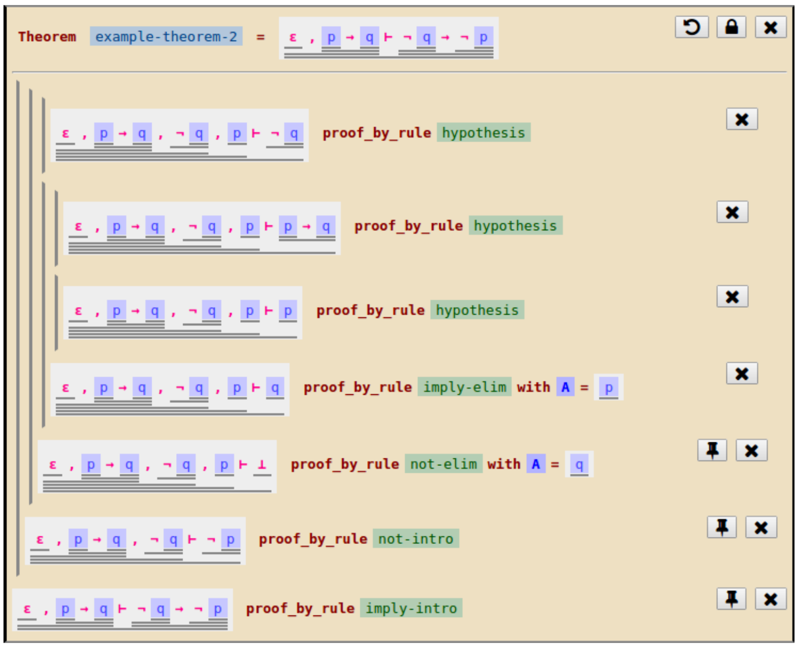
```
Rule top-intro
conclusion Γ ⊢ ⊤

Rule bottom-elim
premise Γ ⊢ ⊥
conclusion Γ ⊢ A

Rule and-intro
premise Γ ⊢ A
premise Γ ⊢ B
conclusion Γ ⊢ A ∧ B

Rule and-elim-left
premise Γ ⊢ A ∧ B
conclusion Γ ⊢ A
parameter B : Prop

Rule and-elim-right
premise Γ ⊢ A ∧ B
conclusion Γ ⊢ B
parameter A : Prop

Rule or-intro-left
premise Γ ⊢ A
conclusion Γ ⊢ A ∨ B

Rule or-intro-right
premise Γ ⊢ B
conclusion Γ ⊢ A ∨ B

Rule or-elim
premise Γ ⊢ A ∨ B
premise Γ , A ⊢ C
premise Γ , B ⊢ C
conclusion Γ ⊢ C
parameters A : Prop , B : Prop

Rule not-intro
premise Γ , A ⊢ ⊥
conclusion Γ ⊢ ¬ A

Rule not-elim
premise Γ ⊢ ¬ A
premise Γ ⊢ A
conclusion Γ ⊢ ⊥
parameter A : Prop

Rule imply-intro
premise Γ , A ⊢ B
conclusion Γ ⊢ A → B

Rule imply-elim
premise Γ ⊢ A → B
premise Γ ⊢ A
conclusion Γ ⊢ B
parameter A : Prop

Rule iff-intro
premise Γ , A ⊢ B
premise Γ , B ⊢ A
conclusion Γ ⊢ A ↔ B

Rule iff-elim-forward
premise Γ ⊢ A ↔ B
premise Γ ⊢ A
conclusion Γ ⊢ B
parameter A : Prop

Rule iff-elim-backward
premise Γ ⊢ A ↔ B
premise Γ ⊢ B
conclusion Γ ⊢ A
parameter B : Prop
```

Figure 5.7: Main Rules for Propositional Logic

For example, these rules can be used together with `hypothesis` as the following



```
Theorem example-theorem-1 = ε , p , q ∨ r ⊢ p ∧ q ∨ p ∧ r
  ε , p , q ∨ r ⊢ q ∨ r  proof_by_rule hypothesis
      ε , p , q ∨ r , q ⊢ p  proof_by_rule hypothesis
      ε , p , q ∨ r , q ⊢ q  proof_by_rule hypothesis
    ε , p , q ∨ r , q ⊢ p ∧ q  proof_by_rule and-intro
  ε , p , q ∨ r , q ⊢ p ∧ q ∨ p ∧ r  proof_by_rule or-intro-left
      ε , p , q ∨ r , r ⊢ p  proof_by_rule hypothesis
      ε , p , q ∨ r , r ⊢ r  proof_by_rule hypothesis
    ε , p , q ∨ r , r ⊢ p ∧ r  proof_by_rule and-intro
  ε , p , q ∨ r , r ⊢ p ∧ q ∨ p ∧ r  proof_by_rule or-intro-right
ε , p , q ∨ r ⊢ p ∧ q ∨ p ∧ r  proof_by_rule or-elim with A = q , B = r
```

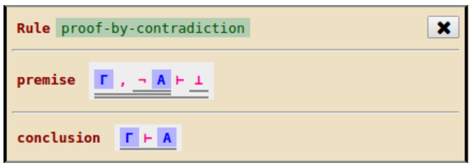


```
Theorem example-theorem-2 = ε , p → q ⊢ ¬ q → ¬ p
      ε , p → q , ¬ q , p ⊢ ¬ q  proof_by_rule hypothesis
        ε , p → q , ¬ q , p ⊢ p → q  proof_by_rule hypothesis
        ε , p → q , ¬ q , p ⊢ p  proof_by_rule hypothesis
      ε , p → q , ¬ q , p ⊢ q  proof_by_rule imply-elim with A = p
    ε , p → q , ¬ q , p ⊢ ⊥  proof_by_rule not-elim with A = q
  ε , p → q , ¬ q ⊢ ¬ p  proof_by_rule not-intro
ε , p → q ⊢ ¬ q → ¬ p  proof_by_rule imply-intro
```

5.4 Classical Logic

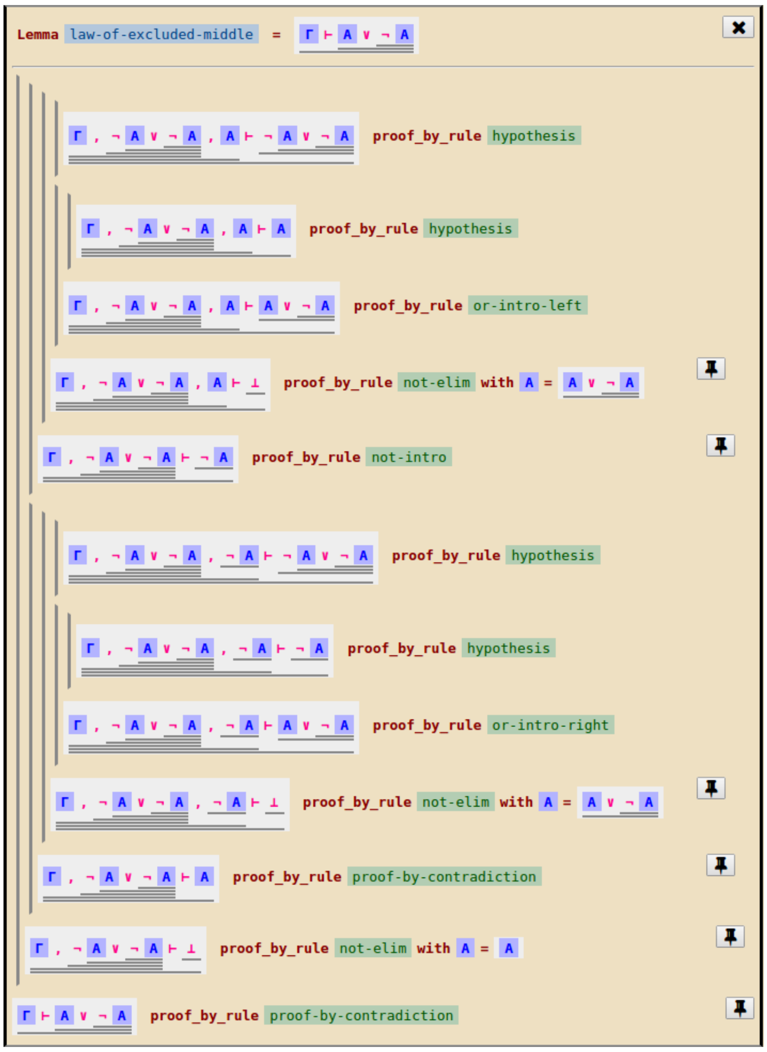
The rules so far create Intuitionistic Logic i.e. it doesn't assume that each proposition must be either true or false. Hence, cannot prove some thing like A ∨ ¬ A.

We can introduce rule `proof-by-contradiction`, which is equivalent to axiom `law-of-exclude-middle` or `double-negation-elim`, to make Intuitionistic Logic become Classical one.

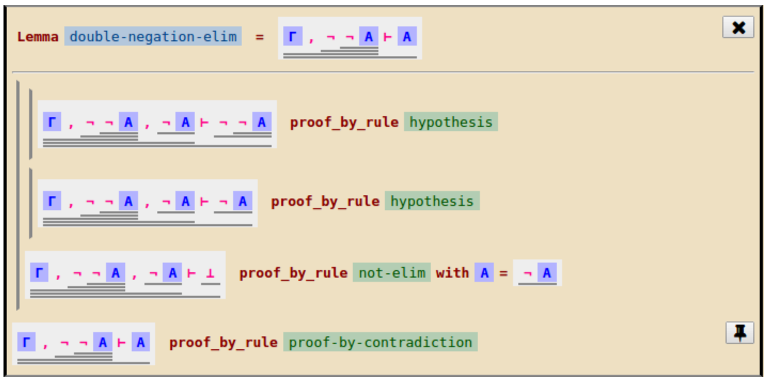


```
Rule proof-by-contradiction                              ✖

premise     Γ , ¬ A ⊢ ⊥

conclusion  Γ ⊢ A
```

And now we can prove `law-of-exclude-middle` and `double-negation-elim` as lemmas.

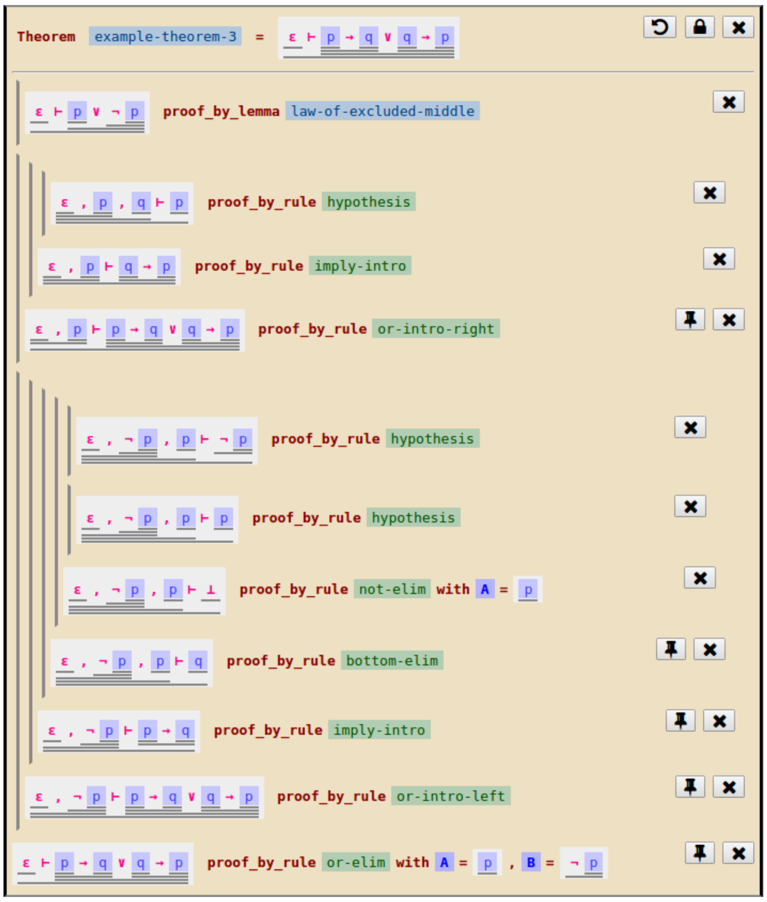


```
Lemma law-of-excluded-middle = Γ ⊢ A ∨ ¬ A                                        ✖

        Γ , ¬ A ∨ ¬ A , A ⊢ ¬ A ∨ ¬ A   proof_by_rule hypothesis
          Γ , ¬ A ∨ ¬ A , A ⊢ A   proof_by_rule hypothesis
        Γ , ¬ A ∨ ¬ A , A ⊢ A ∨ ¬ A   proof_by_rule or-intro-left
      Γ , ¬ A ∨ ¬ A , A ⊢ ⊥   proof_by_rule not-elim with A = A ∨ ¬ A
    Γ , ¬ A ∨ ¬ A ⊢ ¬ A   proof_by_rule not-intro
        Γ , ¬ A ∨ ¬ A , ¬ A ⊢ ¬ A ∨ ¬ A   proof_by_rule hypothesis
          Γ , ¬ A ∨ ¬ A , ¬ A ⊢ ¬ A   proof_by_rule hypothesis
        Γ , ¬ A ∨ ¬ A , ¬ A ⊢ A ∨ ¬ A   proof_by_rule or-intro-right
      Γ , ¬ A ∨ ¬ A , ¬ A ⊢ ⊥   proof_by_rule not-elim with A = A ∨ ¬ A
    Γ , ¬ A ∨ ¬ A ⊢ A   proof_by_rule proof-by-contradiction
  Γ , ¬ A ∨ ¬ A ⊢ ⊥   proof_by_rule not-elim with A = A
Γ ⊢ A ∨ ¬ A   proof_by_rule proof-by-contradiction
```



```
Lemma double-negation-elim = Γ , ¬ ¬ A ⊢ A

Γ , ¬ ¬ A , ¬ A ⊢ ¬ ¬ A   proof_by_rule hypothesis
Γ , ¬ ¬ A , ¬ A ⊢ ¬ A   proof_by_rule hypothesis
Γ , ¬ ¬ A , ¬ A ⊢ ⊥   proof_by_rule not-elim with A = ¬ A
Γ , ¬ ¬ A ⊢ A   proof_by_rule proof-by-contradiction
```

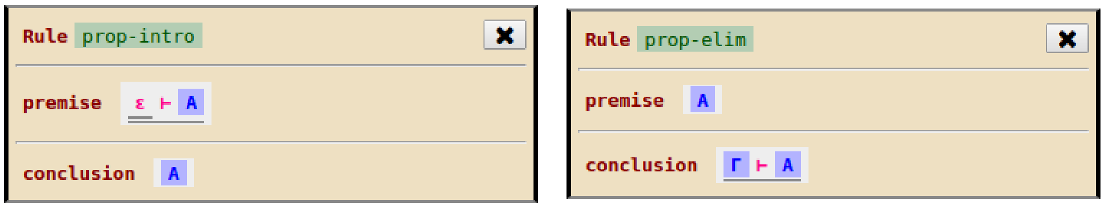
For example, this example only holds only under Classical Logic.



```
Theorem example-theorem-3 = ε ⊢ p → q ∨ q → p

ε ⊢ p ∨ ¬ p   proof_by_lemma law-of-excluded-middle
ε , p , q ⊢ p   proof_by_rule hypothesis
ε , p ⊢ q → p   proof_by_rule imply-intro
ε , p ⊢ p → q ∨ q → p   proof_by_rule or-intro-right
ε , ¬ p , p ⊢ ¬ p   proof_by_rule hypothesis
ε , ¬ p , p ⊢ p   proof_by_rule hypothesis
ε , ¬ p , p ⊢ ⊥   proof_by_rule not-elim with A = p
ε , ¬ p , p ⊢ q   proof_by_rule bottom-elim
ε , ¬ p ⊢ p → q   proof_by_rule imply-intro
ε , ¬ p ⊢ p → q ∨ q → p   proof_by_rule or-intro-left
ε ⊢ p → q ∨ q → p   proof_by_rule or-elim with A = p , B = ¬ p
```

5.5 Validity of Proposition

Although we prove validity of `Judgement` to show that a certain proposition holds under certain assumptions. But `Prop` it self has meaning of validity as well, that is, a proposition holds without any assumptions. The following rules allow us to prove that a `Prop` is valid and make use of its validity when needed.

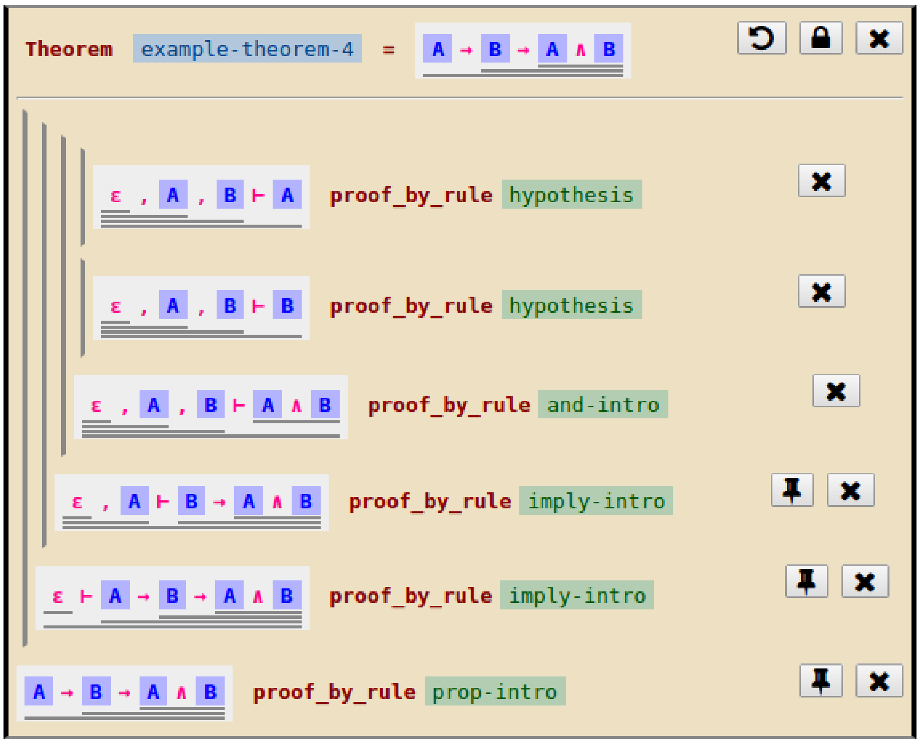


```
Rule prop-intro
premise     ε ⊢ A
conclusion  A
```

```
Rule prop-elim
premise     A
conclusion  Γ ⊢ A
```

Figure 5.8: `prop-intro` can be used to prove that a `Prop` is valid. And its duality, `prop-elim` can be used to prove `Judgement` when its conclusion is valid.

For example, the following theorem shows that `A → B → A ∧ B` always holds no matter of what `A` or `B` will be. Please note that the goal of this theorem is `Prop` rather than `Judgement` as usual.



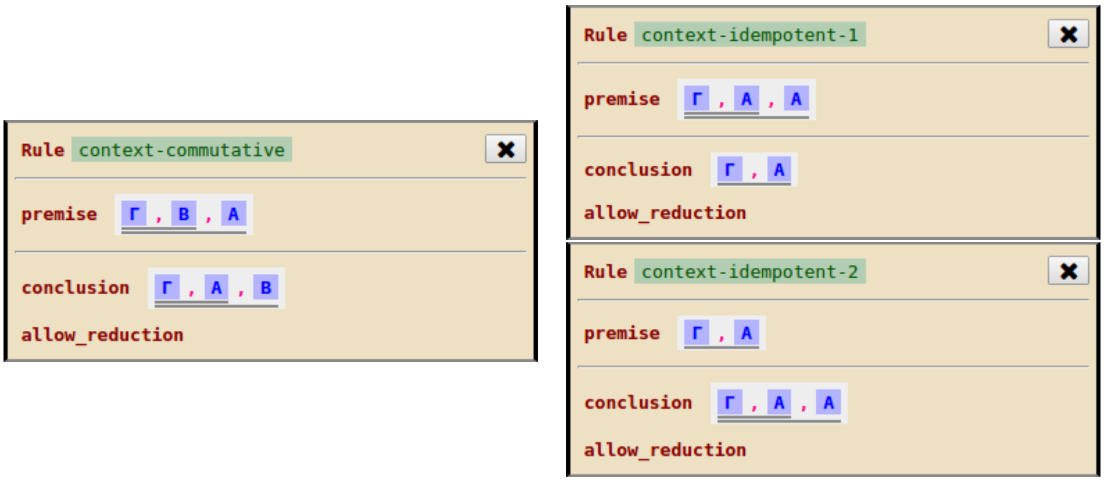
```
Theorem example-theorem-4 = A → B → A ∧ B

ε , A , B ⊢ A          proof_by_rule hypothesis
ε , A , B ⊢ B          proof_by_rule hypothesis
ε , A , B ⊢ A ∧ B      proof_by_rule and-intro
ε , A ⊢ B → A ∧ B      proof_by_rule imply-intro
ε ⊢ A → B → A ∧ B      proof_by_rule imply-intro
A → B → A ∧ B          proof_by_rule prop-intro
```

5.6 Context manipulation

Context in Propositional Logic is just a set of assumption so the order and duplication among assumptions shouldn't matter. Formally, Γ , A , B is *definitionally equal* to Γ , B , A and Γ , A , A is *definitionally equal* to Γ , A. Definitional equality is stronger than object level equality in the sense that two terms can be equal *implicitly* i.e. one term can be transformed to another without any rule or lemma required.

To handle definitional equality, phometa has a mechanism called meta-reduction which allow a rule that has flag `allow_reduction` and doesn't have any parameters to digest the target term, if it returns exactly one sub-goal that has the same grammar and doesn't have any further unification, then replace that sub-goal on the term.

Meta-reduction for context manipulation can be encoded by the following three rules



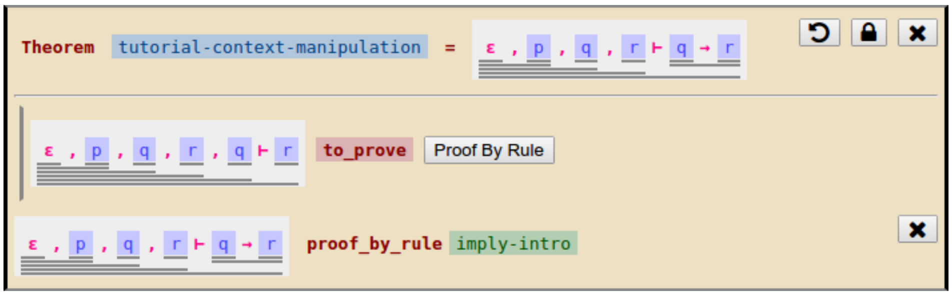
```
Rule context-commutative
premise Γ , B , A
conclusion Γ , A , B
allow_reduction
```

```
Rule context-idempotent-1
premise Γ , A , A
conclusion Γ , A
allow_reduction
```

```
Rule context-idempotent-2
premise Γ , A
conclusion Γ , A , A
allow_reduction
```

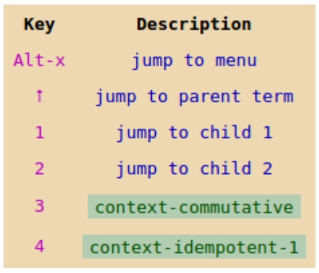
Figure 5.9: Rules for context manipulation

For example, let construct an unfinished theorem as the following



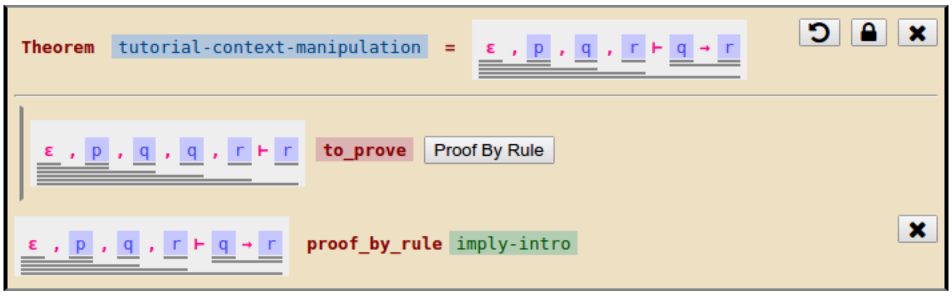
```
Theorem tutorial-context-manipulation = ε , p , q , r ⊢ q → r
ε , p , q , r , q ⊢ r  to_prove  Proof By Rule
ε , p , q , r ⊢ q → r  proof_by_rule imply-intro
```

Then, if you click ε , p , q , r , q on the sub-goal, the keymap will look like this



Key	Description
Alt-x	jump to menu
↑	jump to parent term
1	jump to child 1
2	jump to child 2
3	context-commutative
4	context-idempotent-1

Keymap says that this term is reducible by either `context-commutative` or `context-idempotent-1` . Now we will swap the last two assumption by pressing 4 or clicking the row the keymap directly, the theorem will look like this.

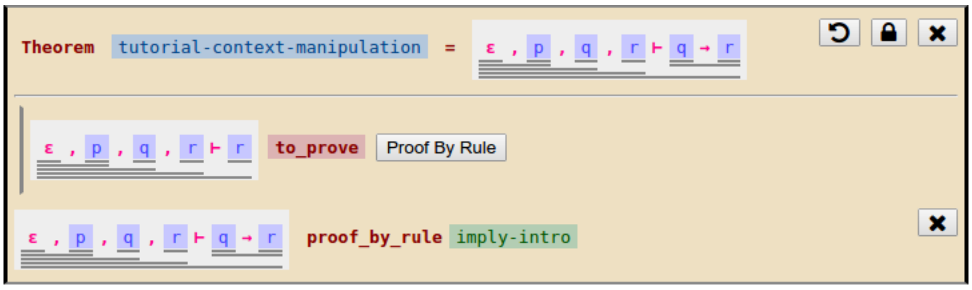


Theorem tutorial-context-manipulation = ε , p , q , r ⊢ q → r

ε , p , q , q , r ⊢ r to_prove Proof By Rule

ε , p , q , r ⊢ q → r proof_by_rule imply-intro

We want to eliminate duplication on of q, this canbe done by clicking ε , p , q , q now `context-idempotent-2` is also available in keymap so we can select it.



Theorem tutorial-context-manipulation = ε , p , q , r ⊢ q → r

ε , p , q , r ⊢ r to_prove Proof By Rule

ε , p , q , r ⊢ q → r proof_by_rule imply-intro

Although I said that white background means not modifiable, but meta-reduction is exceptional because its implicitness. Meta-reduction is very powerful but also dangerous so I limit usage of meta-reduction only for *empty* sub-goal[4] in unfinished theorem where proving process will benefit from it. In contrast, it doesn't make sense to do this in rules or lemmas and only make user confuse.

Context manipulation is not that important to for Propositional Logic since rule `hypothesis` can penetrate thought out of `Context` . However, is better to show it here so you will know how meta-reduction works. This is particularly important when we start to implement Lambda Calculus in the next chapter.

[4]Sub-goal that has associate lemma or rule (including the one that is waiting for parameter(s)), is not empty sub-goal.

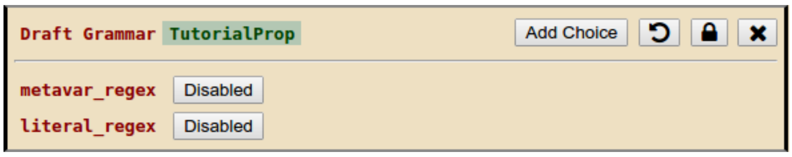
5.7 How to build Grammars and Rules

So far we introduce grammars and rules out of the box, this allows user to prove a theorem which is the most important part directly. However, Phometa also have ability to create new grammars and rules as well.

To show how to build these, I will recreate `Prop` for grammars and recreate `or-elim` and `hypothesis` for rules.

5.7.1 Recreation of `Prop`

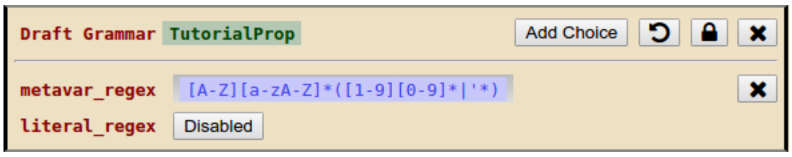
The first step, we need to press "Add Grammar" on one of adding panels in module "Propositional Logic", then enter the grammar name. I will use "TutorialProp" to avoid conflict with the real one.


Draft Grammar TutorialProp

metavar_regex Disabled

literal_regex Disabled

This is just a draft grammar i.e. it cannot be used by any rule or any theorem at the moment. To specify regular expression for meta variable, click a button that follow `metavar_regex`, the button will tern to input box so you can write regular expression[5] here,


Draft Grammar TutorialProp

metavar_regex [A-Z][a-zA-Z]*([1-9][0-9]*|'*)

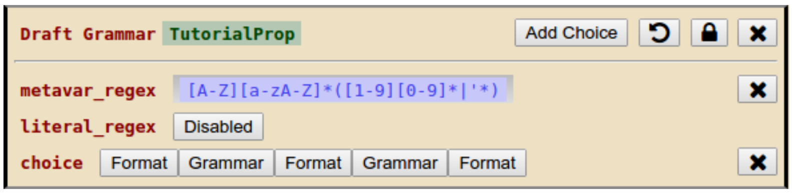
literal_regex Disabled

The background colour of this regular expression is grey, so you can edit it afterword by click the box again. Alternatively you can delete it using close button on the right of the row.

For `literal_regex`, we do nothing about it as `TutorialProp` will not instantiate literal.

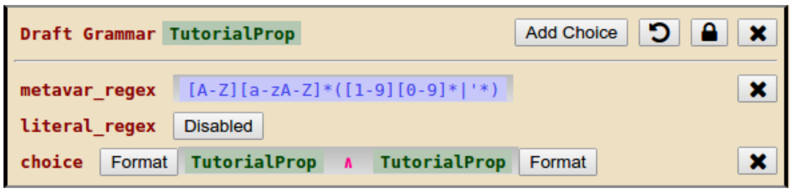
Next, we will add choices to the grammar, I will start with the one that has "$\wedge$" connector. Click "Add Choice" button on the header of this grammar, it will tern to be input box that you can write number of sub-grammars for this choice, in this case it is 2, so write it and press enter.

[5]Please note that this regular expression must comply to JavaScript regexp specification.


Draft Grammar TutorialProp
Add Choice
metavar_regex [A-Z][a-zA-Z]*([1-9][0-9]*|'*)
literal_regex Disabled
choice Format Grammar Format Grammar Format

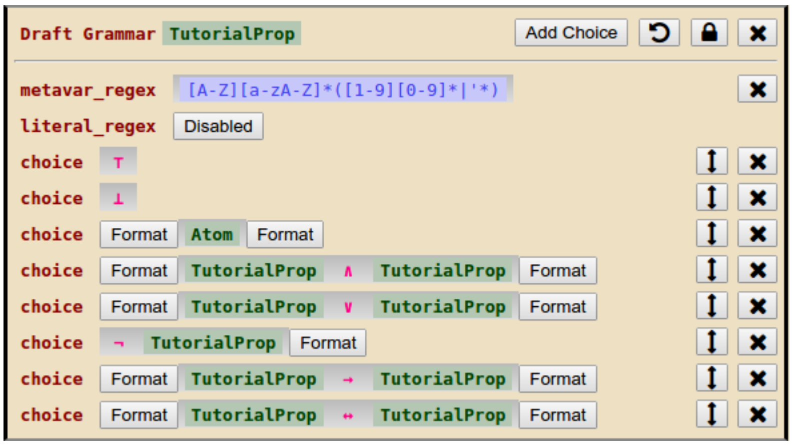
You will see a choice that have buttons labelled “Grammar”, you can click it to specify a sub-grammar. In this case, select TutorialProp in both of position.

There are other three buttons labelled “Format” intersperse sub-grammars which accept any string that will be used as syntax, you can also use unicode input method similar when we construct a term in last chapter. In this case, click button in the middle, press Alt-u , and type “wedge”, you will see “∧” on the keymap pane, then click it. “∧” will appear in input box of middle “Format”. Then hit Return to finished writing this.

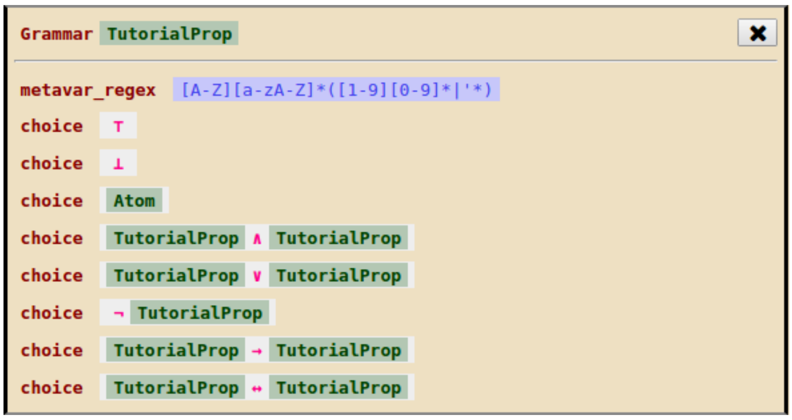

Draft Grammar TutorialProp
Add Choice
metavar_regex [A-Z][a-zA-Z]*([1-9][0-9]*|'*)
literal_regex Disabled
choice Format TutorialProp ∧ TutorialProp Format

The background colour of these are gray so you can edit it similar to metavar_regex . For the first and last “Format” buttons, we do nothing on it as this choice make no use on those positions.

Other choices can be done in similar way, once you finish adding choices, it should look like this.


Draft Grammar TutorialProp
Add Choice
metavar_regex [A-Z][a-zA-Z]*([1-9][0-9]*|'*)
literal_regex Disabled
choice ⊤
choice ⊥
choice Format Atom Format
choice Format TutorialProp ∧ TutorialProp Format
choice Format TutorialProp ∨ TutorialProp Format
choice ¬ TutorialProp Format
choice Format TutorialProp → TutorialProp Format
choice Format TutorialProp ↔ TutorialProp Format

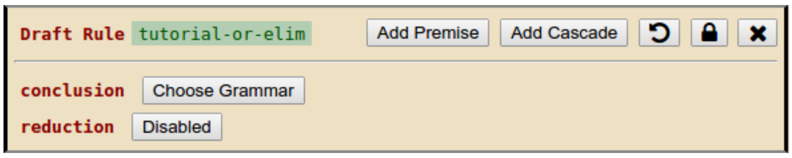
You can lock this draft grammar by clicking lock button on top-right corner, this will transform it as a proper grammar where you can refer it in rules and theorems.



```
Grammar TutorialProp
metavar_regex [A-Z][a-zA-Z]*([1-9][0-9]*|'*)
choice ⊤
choice ⊥
choice Atom
choice TutorialProp ∧ TutorialProp
choice TutorialProp ∨ TutorialProp
choice ¬ TutorialProp
choice TutorialProp → TutorialProp
choice TutorialProp ↔ TutorialProp
```

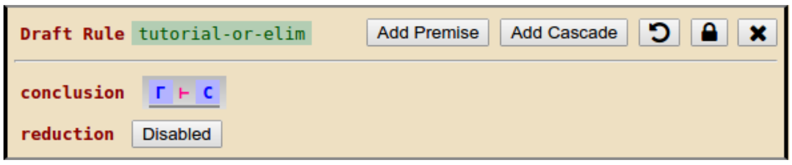
5.7.2 Recreation of `or-elim`

The next thing that we recreate is rule `or-elim` . First, click “Add Rule” in adding panel and write “tutorial-or-elim” on it.



```
Draft Rule tutorial-or-elim    Add Premise  Add Cascade
conclusion  Choose Grammar
reduction   Disabled
```

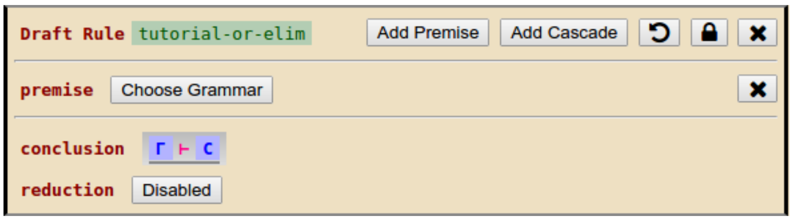
This is just a draft rule i.e. it cannot be used by any theorem at the moment. Let start by filling `conclusion` , we just need to input a term similar to when we input a goal of a theorem.



```
Draft Rule tutorial-or-elim    Add Premise  Add Cascade
conclusion  Γ ⊢ C
reduction   Disabled
```

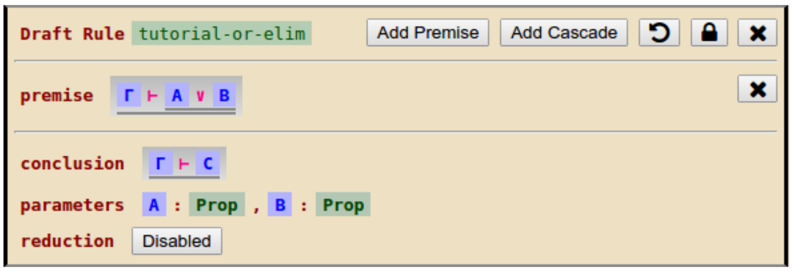
Next is `allow_reduction` it is disabled by default, we could click on button “Disable” to toggle it to “Enable”, however, we don’t want to rise this flag for `tutorial-or-elim` so leave it as it is.

Now, let build the first `premise` by clicking “Add Premise” on the top-right corner of this rule.



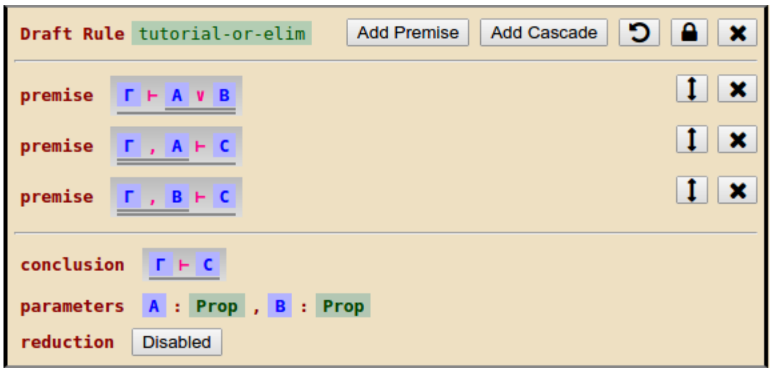
```
Draft Rule tutorial-or-elim    Add Premise  Add Cascade
premise     Choose Grammar
conclusion  Γ ⊢ C
reduction   Disabled
```

Then write a term similar to what we have done in `conclusion`

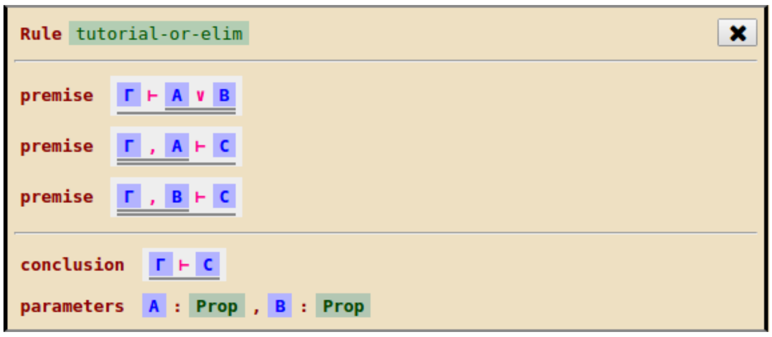


You can see that there is `parameter` row pop-up. This represents meta variables that appear in one of `premise` but not in `conclusion` and you can't manually change it.

The two remaining premises can be done by similar process above

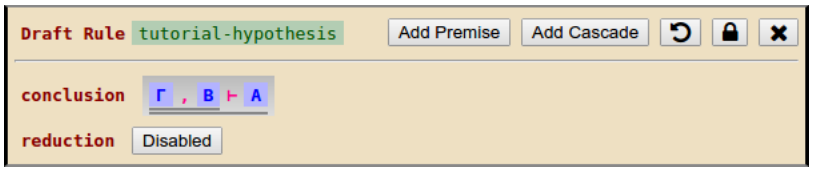


You can lock this draft rule by clicking lock button on top-right corner, this will transform it as a proper rule where you can refer it in theorems.



5.7.3 Recreation of `hypothesis`

The recreation of `or-elim` shows most of features of rule construction except how to deal with cascade so we will recreate `hypothesis` to illustrate this. First, adding a rule with name "tutorial-hypothesis" on it. Then, enter a term for `conclusion` .

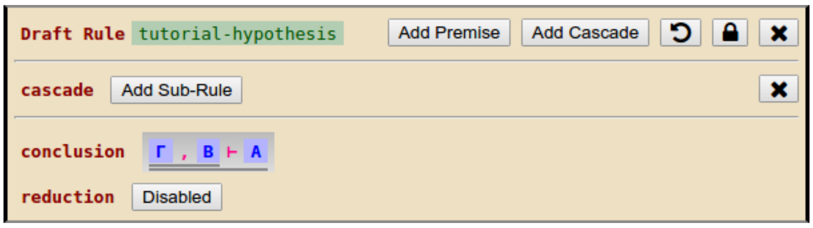


Draft Rule tutorial-hypothesis

conclusion Γ , B ⊢ A

reduction Disabled

Then, add a cascade premise by clicking "Add Cascade" on the top-right corner.



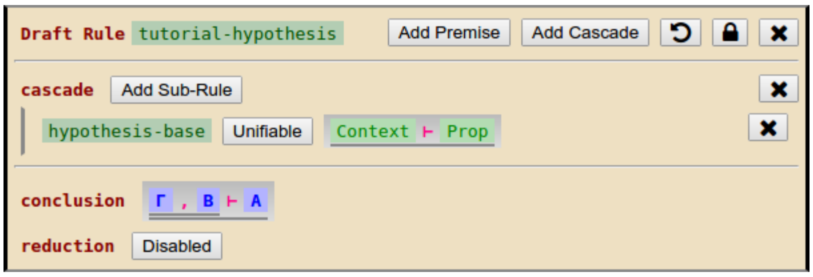
Draft Rule tutorial-hypothesis

cascade

conclusion Γ , B ⊢ A

reduction Disabled

Then click "Add Sub-Rule" to add the first sub-rule, the button will transformed into input box where you can select such a rule, in this case select `hypothesis-base`



Draft Rule tutorial-hypothesis

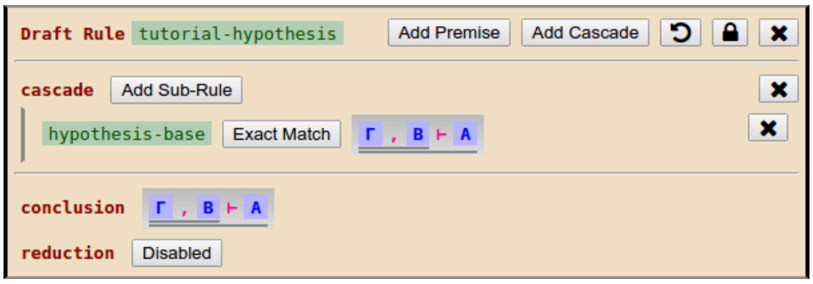
cascade

hypothesis-base Unifiable Context ⊢ Prop

conclusion Γ , B ⊢ A

reduction Disabled

Sub-rule calling-template will appear, you can construct a term that will passed into this sub-rule. A sub-rule is unifiable by default, you can set it to `exact_match` by toggling button "Unifiable".



Draft Rule tutorial-hypothesis

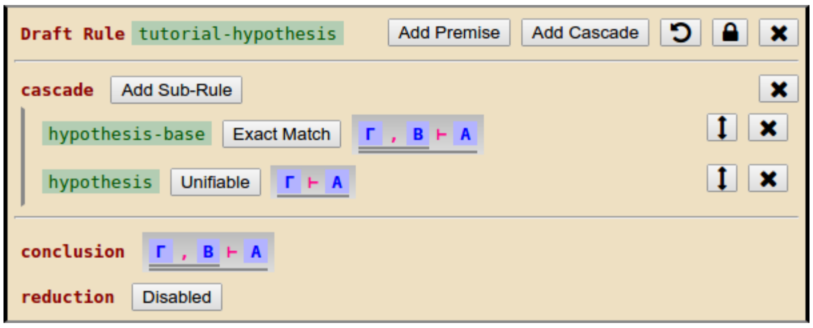
cascade

hypothesis-base Exact Match Γ , B ⊢ A

conclusion Γ , B ⊢ A

reduction Disabled

The second sub-rule could be construct on similar manner to the first one.


Draft Rule tutorial-hypothesis

cascade

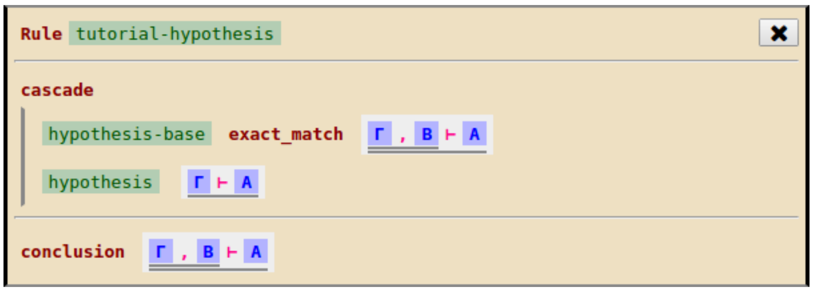
hypothesis-base Exact Match $\Gamma , B \vdash A$

hypothesis Unifiable $\Gamma \vdash A$

conclusion $\Gamma , B \vdash A$

reduction Disabled

Then you can lock this draft rule, to change it to a proper rule.


Rule tutorial-hypothesis

cascade

hypothesis-base exact_match $\Gamma , B \vdash A$

hypothesis $\Gamma \vdash A$

conclusion $\Gamma , B \vdash A$

5.8 Exercises

Credit: Some of material here modified from tutorial 3, 4, and 5 of first year Logic course, Department of Computing, Imperial College London. Thank you Prof Ian Hodkinson and Dr Krysia Broda for this.

- Create a theorem and proof each of the following
  - ε , p ∧ q ⊢ p
  - p ∧ q → p (reminder: this is `Prop` not `Judgement`)
  - ε , p ⊢ q → p ∧ q
  - ε , p → q → r ⊢ p ∧ q → r
  - ε , p → q → r ⊢ p → q → p → r
  - ε , p ∧ q → r ⊢ p → q → r
  - ε , p , q ∨ p → q ⊢ p ∧ q
  - ε , P → Q , ¬ P → R , Q → S , R → S ⊢ S
  - ε , R → ¬ I , I ∨ F , ¬ F ⊢ ¬ R
  - ε , A ∨ B , ¬ A ∧ ¬ C , B → C ⊢ C
  - ε , F → B ∨ W , ¬ B ∨ P , W → P ⊢ ¬ F
  - ε , A ∧ B → C , ¬ D → ¬ E → F , C → E → F ⊢ A → B → D
  - ε , ¬ P , A ∧ W → P , ¬ I → A , ¬ W → M , E → ¬ I ∧ ¬ M ⊢ ¬ E
  - ε , A → B → D ∨ E , ¬ D ∨ G , E ∧ B → G ⊢ B → ¬ A
  - ε , ¬ T , P → ¬ R ∧ Q , P → R ∨ T ⊢ P → ¬ Q
  - ε , C ∧ N → T , H ∧ ¬ S , H ∧ ¬ S ∨ C → P ⊢ N ∧ ¬ T → P

- `ε , A ↔ ¬ B ⊢ ¬ A ↔ B`

- Equivalence of Propositional Logic could be written in the form ($A \equiv B$) stated that, A holds if and only if B holds. Please introduce new a grammar `Equivalence` and write `equiv-intro` similar section 5.5 and prove the following
  - `A ∧ B ≡ B ∧ A`
  - `A ∨ B ≡ B ∨ A`
  - `A ∧ B ∧ C ≡ A ∧ B ∧ C`
  - `A ∨ B ∨ C ≡ A ∨ B ∨ C`
  - `A → ⊥ ≡ ¬ A`
  - `A ≡ ¬ ¬ A`
  - `A → B ≡ ¬ A ∨ B`
  - `A ↔ B ≡ A → B ∧ B → A`
  - `¬ A ∧ B ≡ ¬ A ∨ ¬ B`
  - `¬ A ∨ B ≡ ¬ A ∧ ¬ B`
  - `A ∧ B ∨ C ≡ A ∧ B ∨ A ∧ C`
  - `A ∨ B ∧ C ≡ A ∨ B ∧ A ∨ C`
  - `A ∧ A ∨ B ≡ A`
  - `A ∨ A ∧ B ≡ A`
  - `C → A ∧ ¬ C → B ≡ C ∧ A ∨ ¬ C ∧ B`
- (Challenge) Extend this Propositional Logic to become First Order Logic.

Chapter 6

Example Formal System - Lambda Calculus

Since the last two chapters show most of features and usability of *Phometa* already so this chapter aims to show that Phometa is powerful enough as it can even encode more complex formal system like *typed lambda calculus*. Hence, it is clear that Phometa is suitable to encode most of formal system that user can think of.

6.1 Untyped Lambda Calculus

6.2 Simply-typed Lambda Calculus

TODO: don't forget to write about unification (type resolution)

Chapter 7

Specification

In this chapter, we will the full detail of phometa.

7.1 Overview

mainly talk about phometa UI structure, grids, keymap pane, mode menu, and messages panel

7.2 Repository

structure of repository

- add new sub-package or module inside a package (using package pane)
- load/save repository using textarea in home pane + also talk about stdlib
- add/swap nodes inside module
- dedicate view for each node (using package pane)

7.3 Auto Complete

autocomplete in general

define its param i.e. return call back

how to do autocomplete

7.4 Node Comment

7.5 Node Grammar

7.6 Root Term

State that root term input method use lots of autocomplete and explain more on extended feature

explain navigation mode and meta reduction

make sure that we include EVERY key binding and generated messages

7.7 Node Rule

7.8 Node Theorem

Chapter 8

Implementation

This chapter aims to illustrate high-level implementation of Phometa. Reader will get a roughly idea on how does Phometa works.

8.1 Decision on programming language

Elm[1] is a functional reactive programming language. It allows programmer to create web application by declaratively coding in Haskell-like language then compile the program to JavaScript.

One of the most attractive feature of Elm is its reactivity. This idea introduces a new data type called "Signal"[1] which is a data type that can change over time. For example, `Mouse.isDown : Signal Bool` represents Boolean that holds "True" when the mouse is pressed, `Window.dimensions : Signal (Int, Int)` represents a pair of integers that holds current width and height of the window respectively. Custom Signal can be created using higher order functions as this example,

```
Signal.map  : (a -> b) -> Signal a -> Signal b

window_area : Signal Int
window_area = Signal.map (\(w, h) -> w * h) Window.dimensions
```

`window_area` is a signal that represents the area of current window, if you resize the window, this value will change on real time. This is bacause when the dimensions of the window is changed, it will notify `Window.dimensions` to updated its value, `Window.dimensions` in tern, notify `window_area` to update it value as well.

[1]Signal has been removed from Elm version 0.17 recently, however, Phometa uses Elm version 0.16 so it is fine to talk about Signal.

These Signals can link together to form a dependency graph, eventually we will create `main : Signal Html` which is a HTML representation that can change over time, Elm will detect this `main` and render it in web-browser. For example,

```
import Mouse
import Signal exposing (Signal)
import Html exposing (Html, div, text)
import Html.Attributes exposing (style)

view : Bool -> Html
view is_clicked =
  if is_clicked then div [style [("color", "green")]]
                         [text "Clicked !!"]
                 else text "Please click this page."

main : Signal Html
main = Signal.map view Mouse.isDown
```

Once this code is complied, we will get HTML file (with JavaScript embedded) such that if it is opened in web-browser, it will show text "Please click this page.", if you click somewhere in the page, the text will change to "Clicked !!" with green background colour, this will change back when you release the mouse.

The example above shows that Elm could replace all of HTML, JavaScript, and even CSS. In another word, the entire front-end development could be done in one purely functional language. Therefore, Elm is a good candidate to be used as main programming language for web-based application like Phometa.

For more information, please see Elm official website at elm-lang.org.

8.2 Model-Controller-View Architecture

Recall from the previous example, you could see that the HTML will depend directly on corresponding Signal(s) but in real application we need a way to store variables, this is achievable by the following function

```
Signal.foldp : (a -> b -> b) -> b -> Signal a -> Signal b
```

Initially, the Signal generated form `Signal.foldp` will have a value the same as the second argument, then whenever the Signal from the third argument is updated (possibly with the same value), it will be *reduced* with the current output Signal in order to get new output Signal. For example,

```
click_total : Signal Int
click_total =
  let fold_func clicked acc = if clicked then acc + 1 else acc
   in Signal.foldp fold_func 0 Mouse.isDown
```

`click_total` is a Signal that store an integers. Initially, it starts with 0. When user click mouse on the page, `Mouse.isDown` will notify `click_total` to invoke `fold_func`, and the value of this Signal will updated to 1. If you click it again it will be 2, 3, 4, and so on.

In Phometa, an accumulator of `Signal.foldp` is `Model` represented the global state of the entire program. The third argument (input Signal) of `Signal.foldp` is `Action` represented the Signal that change every time when user pressing the keyboard or clicking a button with Phometa. We could define the main entry of Phometa program as the following[2]

```
module Main where

import Html exposing (Html)
import Keyboard exposing (keysDown)
import Set exposing (Set)
import Models.Model exposing (Model)
import Models.ModelUtils exposing (init_model)
import Models.Action exposing (Action(..), mailbox, address)
import Updates.Update exposing (update)
import Views.View exposing (view)

keyboard_signal : Signal Action
keyboard_signal = Signal.map
  (Set.toList >> List.sort >> ActionKeystroke) keysDown

action_signal : Signal Action
action_signal = Signal.merge mailbox.signal keyboard_signal

model_signal : Signal Model
model_signal = Signal.foldp update init_model action_signal

main : Signal Html
main = Signal.map view model_signal
```

Figure 8.1: Simplified version of `Main.elm`

By using `Signal.foldp` together with `update : Action -> Model -> Model`, we achieve `model_signal` which is a Signal represented the current global state of Phometa, which, in tern, can be transformed using `Signal.foldp` together with `view : Model -> Html` to achieve `main` which represented the current HTML page.

Now, it becomes clear that Model-Controller-View architecture could be separated by `Model`, `update : Action -> Model -> Model`, and `view : Model -> Html` respectively.

[2]Phometa code represented here is just a simplify version, I can't show the real one because MCV architecture is modified to support back-end communication so it is harder to understand.

8.3 The Anatomy of Phometa Repository

Phometa is been developed using *Git* as the version control and hosted at `https://github.com/gunpinyo/phometa`.

The Phometa repository [3] has the structure as the following

- `build/` — contains complied version of Phometa and PDF version of this report.
- `doc/` — contains source code and supplementary images of this report.
- `scripts/` — contains scripts that can complie source code, run test suites, etc.
- `src/` — contains the entire source code of Phometa
    - ⋆ `Naive/` — contains naive JavaScript functions that Elm functions can call.
    - ⋆ `Tools/` — contains additional functions that are useful in general.
    - ⋆ `Models/` — contains dependencies of `Model` and its utilities.
    - ⋆ `Updates/` — contains dependencies of `updates` and its utilities.
    - ⋆ `Views/` — contains dependencies of `view` and its utilities.
    - ⋆ `Main.elm` — the main entry of Phometa that wire up MCV components.
    - ⋆ `repository.json` — initial proofs repository that contains standard library.
    - ⋆ `phometa-server.py` — a python scripts back-end server.
    - ⋆ `naive.js` — JavaScript that will be ship with complied version of Phometa.
    - ⋆ `style.scss` — layout and theme of Phometa.
- `tests/` — contains tests suites
- `.gitignore` — specification of files that will be ignored by Git.
- `.travis.yml` — meta data regarding to Travis continuous integration.
- `LICENSE` — 3-clause BSD license.
- `README.md` — summary information for of this repository.
- `elm-package.json` — meta data regarding to Elm dependencies.
- `logo.png` — logo that will be used for favicon.

Please note that some files are omitted here due to limited space.

[3] Please do not confuse with proofs repository which is the root package of proofs content

8.4 Model, Command, Keymap, and Action

Model is defined in src/Models/Model.elm as the following

```
type alias Model =
  { config        : Config
  , root_package  : Package
  , root_keymap   : Keymap
  , grids         : Grids
  , pane_cursor   : PaneCursor
  , mode          : Mode
  , message_list  : MessageList
  , environment   : Environment
  }
```

Figure 8.2: Declaration of Model

A function that can manipulate Model has type signature as

```
type alias Command = Model -> Model
```

Action is defined in src/Models/Action.elm as the following

```
type Action
  = ActionNothing
  | ActionCommand Command
  | ActionKeystroke Keystroke
```

Figure 8.3: Declaration of Action

One way to interact with Phometa is to click some button (or any clickable area), when this is clicked, it will update the value of `action_signal` to "`ActionCommand cmd`" where `cmd` is an instance of `Command` injected to that button earlier.

`Signal.foldp` will detect change in `action_signal` so it applies function `update` with "`ActionCommand cmd`" to `model_signal`. The function `update`, in tern, will apply command `cmd` to current model and this is how model can be updated.

Again, `Signal.map` will detect change in `action_model` so it applies function `view` with current model to get latest HTML instance, please note that `view` might inject further commands to buttons, if user click any button, then the entire process will happen again.

Another way to interact with Phometa is to press some keystroke, this will make action_signal changes to "ActionKeystroke keystroke", where keystroke is a List of KeyCode obtained from Keyboard.keysDown.

Signal.foldp will detect change in action_signal so it applies function update with "ActionKeystroke keystroke" to model_signal. The function update, in tern, will search a command that corresponded to this keystroke from model.root_keymap to apply such a command with current model. Now, the remaining process is the same as when we click button.

8.5 Structure of Proofs Repository

In this section, we will show the complete internal representation of a proofs repository. All codes inside this section come from src/Models/RepoModel.elm

Let define some common elements first[4].

```
type alias ContainerName = String
type alias ContainerPath = List ContainerName
type alias Format = String
type alias VarName = String
type alias Parameters = List { grammar  : GrammarName
                             , var_name : VarName }
type alias Arguments = List RootTerm
```

Figure 8.4: Representation of common elements that will be used later on

Now, we can define a proof repository which is just a big Package stored in global model named model.root_package and can be defined as the following

```
type alias PackageName = ContainerName
type alias PackagePath = List PackageName
type alias Package = { is_folded : Bool
                     , dict : Dict ContainerName PackageElem }
type PackageElem
  = PackageElemPkg Package
  | PackageElemMod Module
```

Figure 8.5: Representation of Package and related components.

[4] Arguments depends on RootTerm which will be defined later in this section

We could think a Package roughly as a collection containing other Packages and Modules. We also need PackagePath to reference a package relatively to root package.

Module is a collection containing nodes. ModulePath is a unique reference from root package to this module.

```
type alias ModuleName = ContainerName
type alias ModulePath = { package_path : PackagePath
                        , module_name  : ModuleName }
type alias Module = { is_folded : Bool
                    , nodes     : OrderedDict NodeName Node }
```

Figure 8.6: Representation of Module and related components.

There are four types of Node which are Comment, Grammar, Rule, and Theorem[5]. NodePath is a unique reference from root package to this node.

```
type alias NodeName = ContainerName

type alias NodePath =
  { module_path : ModulePath
  , node_name   : NodeName
  }

type Node
  = NodeComment String
  | NodeGrammar Grammar
  | NodeRule Rule
  | NodeTheorem Theorem Bool -- Bool = has_locked

type NodeType
  = NodeTypeComment
  | NodeTypeGrammar
  | NodeTypeRule
  | NodeTypeTheorem
```

Figure 8.7: Representation of Node and related components.

We could define Grammar as the following,

[5] Bool that follows NodeTheorem tells whether that theorem has been converted to lemma or not

```
type alias GrammarName = String

type alias Grammar =
  { is_folded      : Bool
  , has_locked     : Bool
  , metavar_regex : Maybe Regex
  , literal_regex : Maybe Regex
  , choices        : List GrammarChoice
  }

type alias GrammarChoice = StripedList Format GrammarName
```

Figure 8.8: Representation of `Grammar` and related components.

`StripedList Format GrammarName` is just a pair of “list of `Format`” and “list of `GrammarName` ” but it also ensure that length of “list of `Format`” is longer than “list of `GrammarName`” by exactly 1.

Next is a term, which can be defined like this,

```
type Term
  = TermTodo
  | TermVar VarName
  | TermInd GrammarChoice (List Term)

type alias RootTerm =
  { grammar : GrammarName
  , term : Term
  }

type VarType
  = VarTypeMetaVar
  | VarTypeLiteral
```

Figure 8.9: Representation of `Term` and related components.

We usually use `RootTerm` as a term at the top level, this is because it know its known grammar. In fact, `Term` is just an auxiliary model of `RootTerm`. Please note that it is possible to fine the grammar of every sub-term of a `RootTerm`, because we can refer to `grammar` property of `RootTerm` for top level and refer to `GrammarChoice` of `TermInd` for deeper sub-term.

Once `RootTerm` has been defined, we can define `Rule` and `Theorem` as the following,

```
type alias RuleName = String

type alias Rule =
  { is_folded         : Bool
  , has_locked        : Bool
  , allow_reduction   : Bool
  , parameters        : Parameters
  , conclusion        : RootTerm
  , premises          : List Premise
  }

type Premise
  = PremiseDirect RootTerm
  | PremiseCascade (List PremiseCascadeRecord)

type alias PremiseCascadeRecord =
  { rule_name : RuleName
  , pattern : RootTerm
  , arguments : Arguments
  , allow_unification : Bool
  }
```

Figure 8.10: Representation of `Rule` and related components.

```
type alias TheoremName = String

type alias Theorem =
  { is_folded : Bool
  , goal      : RootTerm
  , proof     : Proof
  }

type Proof
  = ProofTodo
  | ProofTodoWithRule RuleName Arguments
  | ProofByRule RuleName Arguments PatternMatchingInfo
      (List Theorem)
  | ProofByLemma TheoremName PatternMatchingInfo
```

Figure 8.11: Representation of `Theorem` and related components.

PatternMatchingInfo is a result when a pattern matching is successful and can be defined as the following,

```
type alias SubstitutionList =
  List { old_var : VarName
       , new_root_term : RootTerm
       }

type alias PatternMatchingInfo =
  { pattern_variables : Dict VarName RootTerm
  , substitution_list : SubstitutionList
  }
```

Figure 8.12: Representation of PatternMatchingInfo and related components.

As we know that terms that match to the same pattern variable will be unified during matching, SubstitutionList tell us that what meta-variables (`old_var`) should be replaced `new_root_term`. The order of substitution is important as later substitution might depend on earlier one. Now we can guarantee that, after unification, each pattern variable will match to only one term, this make it possible to create a dictionary `pattern_variables` that map these pattern variables to the corresponded terms.

8.6 Samples of code — Pattern Matching

Although it is impossible to show all of codes written for Phometa, but there is a faction of code related to *Pattern Matching* that is small to be shown entirely, it is one of the most interesting part as well. All codes inside this section come from `src/Models/RepoUtils.elm`.

Let start with the most important, `pattern_match` is a function that pattern match `pattern` against `target`, if success, it will the return pattern matching information as described in previous section. Another function is `pattern_match_multiple` which is the same as previous function but and receive list of pair of `(pattern, target)`

```
pattern_match : ModulePath -> Model -> RootTerm -> RootTerm
                  -> Maybe PatternMatchingInfo
pattern_match module_path model pattern target =
  pattern_match_get_vars_dict pattern target
    |> (flip Maybe.andThen)
         (vars_dict_to_pattern_matching_info module_path model)

pattern_match_multiple : ModulePath -> Model
                           -> List (RootTerm, RootTerm)
                           -> Maybe PatternMatchingInfo
pattern_match_multiple module_path model list =
  let vars_dict_maybe_list =
        List.map (uncurry pattern_match_get_vars_dict) list
   in if List.any ((==) Nothing) vars_dict_maybe_list then
        Nothing
      else
        vars_dict_maybe_list
          |> List.filterMap identity
          |> merge_pattern_variables_list
          |> vars_dict_to_pattern_matching_info
               module_path model
```

Figure 8.13: Functions `pattern_match` and `pattern_match_multiple`.

Both of functions above call function `pattern_match_get_vars_dict` that perform the actual pattern matching and return a dictionary that map each pattern variable to a list of terms. The the pattern match process can be described as the following

- If the grammar of pattern and target is not the same, fail immediately.
- If any of pattern or target is unfinished term, fail immediately.
- If pattern is a variable, success immediately with that variable map to the target.
- If target is a variable but pattern is not, fail immediately, this is one way matching.
- the only possibility left is both of them are `TermInd`[6], so it can process as the following
  - If the grammar choice of pattern and target is not the same, fail immediately.
  - Otherwise, recursively call current function on their sub terms,
    - If any of sub-term fail pattern matching, then the entire function fail.
    - Otherwise, merge these sub-result together and successfully return it.

[6]A term can be either unfinished, variable, or `TermInd`

```
pattern_match_get_vars_dict : RootTerm -> RootTerm ->
                       Maybe (Dict VarName (List RootTerm))
pattern_match_get_vars_dict pattern target =
  if pattern.grammar /= target.grammar then Nothing else
  case (pattern.term, target.term) of
    (TermTodo, _) -> Nothing
    (_, TermTodo) -> Nothing
    (TermVar var_name, _) -> Just (Dict.singleton
                                   var_name [target])
    (_, TermVar _) -> Nothing
    (TermInd pat_mixfix pat_sub_terms,
        TermInd tar_mixfix tar_sub_terms) ->
      if pat_mixfix /= tar_mixfix then Nothing else
        let maybe_result_list = List.map2
              pattern_match_get_vars_dict
              (get_sub_root_terms pat_mixfix pat_sub_terms)
              (get_sub_root_terms tar_mixfix tar_sub_terms)
         in if List.any ((==) Nothing) maybe_result_list then
              Nothing
            else
              maybe_result_list
                |> List.filterMap identity
                |> merge_pattern_variables_list
                |> Just
```

Figure 8.14: Function `pattern_match_get_vars_dict`.

After `pattern_match_get_vars_dict` produces matching dictionary. `vars_dict_to_pattern_matching_info` will try to fit this into `PatternMatchingInfo`. This can be done by finding a `SubstitutionList` that can eliminate all of ambiguity among terms which are the value of matching dictionary. This `SubstitutionList` can be found by using the following method.

- For each pattern variable, check whether it is meta variable or literal.
- If it is meta variable, unify corresponded terms to each other then append these unification result to the main `SubstitutionList`.
- If it is literal, check that each of term is that exact variable as well, otherwise, fail.

Then we can substitute this `SubstitutionList` to the matching directly to get `pattern_variables`, hence be able to return `PatternMatchingInfo` as expected.

```
vars_dict_to_pattern_matching_info : ModulePath -> Model
                                       -> Dict VarName (List RootTerm)
                                       -> Maybe PatternMatchingInfo
vars_dict_to_pattern_matching_info module_path model vars_dict =
  let subst_list_func var_name root_term_list maybe_acc_subst_list =
        let grammar_name = (.grammar) (list_get_elem 0 root_term_list)
         in case get_variable_type module_path model
                  var_name grammar_name of
              Nothing             -> Nothing
              Just VarTypeMetaVar -> case root_term_list of
                []              -> Nothing
                root_term :: [] -> maybe_acc_subst_list
                fst_root_term :: other_root_terms ->
                  let fold_func root_term maybe_acc =
                    let maybe_partial_subst_list =
                      Maybe.andThen maybe_acc (\acc ->
                       unify module_path model
                        (multiple_root_substitute acc fst_root_term)
                        (multiple_root_substitute acc root_term))
                     in Maybe.map2 (++) maybe_acc
                          maybe_partial_subst_list
                   in List.foldl fold_func
                        maybe_acc_subst_list other_root_terms
              Just VarTypeLiteral ->
                if List.all ((==) ({ grammar = grammar_name
                                   , term = TermVar var_name
                                   })) root_term_list
                  then maybe_acc_subst_list else Nothing
      maybe_subst_list = Dict.foldl subst_list_func
                           (Just []) vars_dict
   in Maybe.map (\subst_list ->
        { pattern_variables =
            Dict.map (\var_name root_term_list ->
              multiple_root_substitute subst_list <|
                list_get_elem 0 root_term_list) vars_dict
        , substitution_list = subst_list
        }) maybe_subst_list
```

Figure 8.15: Function `vars_dict_to_pattern_matching_info`.

Function `unify` receives term `a` and `b` then try to build most general unifier (mgu) which is the most general substitution list that is specific enough to make term `a` and `b` identical.

Unification process is similar to pattern matching in `pattern_match_get_vars_dict` but this is this is two wat matching and it is aware whether a variable is meta variable or literal.

```
unify : ModulePath -> Model -> RootTerm -> RootTerm
         -> Maybe (SubstitutionList)
unify module_path model a b =
  if a.grammar /= b.grammar then Nothing else
  case (a.term, b.term) of
    (TermTodo, _) -> Nothing
    (_, TermTodo) -> Nothing
    (TermVar a_var_name, _) ->
      case get_variable_type module_path model a_var_name a.grammar of
        Nothing -> Nothing
        Just VarTypeMetaVar -> Just ([{ old_var = a_var_name
                                      , new_root_term = b}])
        Just VarTypeLiteral -> case b.term of
          TermVar b_var_name ->
            case get_variable_type module_path model
                   b_var_name b.grammar of
              Nothing -> Nothing
              Just VarTypeMetaVar -> Just ([{ old_var = b_var_name
                                            , new_root_term = a}])
              Just VarTypeLiteral -> if a_var_name == b_var_name
                                     then Just [] else Nothing
          _                  -> Nothing
    (_, TermVar b_var_name) ->
      case get_variable_type module_path model b_var_name b.grammar of
        Nothing -> Nothing
        Just VarTypeMetaVar -> Just ([{ old_var = b_var_name
                                      , new_root_term = a}])
        Just VarTypeLiteral -> Nothing
    (TermInd a_mixfix a_sub_terms, TermInd b_mixfix b_sub_terms) ->
      if a_mixfix /= b_mixfix then Nothing else
        let fold_func (a_root_sub_term, b_root_sub_term)
                maybe_acc_subst_list =
              let maybe_partial_subst_list =
                    Maybe.andThen maybe_acc_subst_list
                      (\subst_list -> unify module_path model
                         (multiple_root_substitute
                           subst_list a_root_sub_term)
                         (multiple_root_substitute
                           subst_list b_root_sub_term))
               in Maybe.map2 List.append maybe_acc_subst_list
                                         maybe_partial_subst_list
         in List.foldl fold_func (Just []) <|
              List.map2 (,) (get_sub_root_terms a_mixfix a_sub_terms)
                            (get_sub_root_terms b_mixfix b_sub_terms)
```

Figure 8.16: Function unify.

Lastly, we can implement the remaining auxiliary functions of pattern matching as the following,

```
substitute : VarName -> Term -> Term -> Term
substitute old_var new_term top_term =
  case top_term of
    TermTodo -> TermTodo
    TermVar var_name -> if var_name == old_var
                        then new_term else TermVar var_name
    TermInd grammar_choice sub_terms ->
      TermInd grammar_choice <|
        List.map (substitute old_var new_term) sub_terms

multiple_root_substitute : SubstitutionList -> RootTerm -> RootTerm
multiple_root_substitute list top_root_term =
  let fold_func record acc =
        substitute record.old_var record.new_root_term.term acc
      update_func top_term = List.foldl fold_func top_term list
   in Focus.update term_ update_func top_root_term

pattern_substitute : Dict VarName RootTerm -> Term -> Term
pattern_substitute dict top_term =
  case top_term of
    TermTodo -> TermTodo
    TermVar var_name -> case Dict.get var_name dict of
                        Nothing           -> TermVar var_name
                        Just new_root_term -> new_root_term.term
    TermInd grammar_choice sub_terms ->
      TermInd grammar_choice <|
        List.map (pattern_substitute dict) sub_terms

pattern_root_substitute : Dict VarName RootTerm -> RootTerm -> RootTerm
pattern_root_substitute dict top_root_term =
  Focus.update term_ (pattern_substitute dict) top_root_term

pattern_matching_info_substitute : PatternMatchingInfo -> SubstitutionList
                                 -> PatternMatchingInfo
pattern_matching_info_substitute pm_info substitution_list =
  { pattern_variables = Dict.map (\ _ ->
      multiple_root_substitute substitution_list) pm_info.pattern_variables
  , substitution_list = List.append pm_info.substitution_list
                        substitution_list }

pattern_matchable : ModulePath -> Model -> RootTerm -> RootTerm -> Bool
pattern_matchable module_path model pattern target =
  pattern_match module_path model pattern target /= Nothing

merge_pattern_variables_list : List (Dict VarName (List RootTerm)) ->
                                 Dict VarName (List RootTerm)
merge_pattern_variables_list main_list =
  let dict_fold_func key target_val acc =
        let old_val = Maybe.withDefault [] (Dict.get key acc)
            new_val = List.append target_val old_val
         in Dict.insert key new_val acc
      list_fold_func dict acc = Dict.foldl dict_fold_func dict acc
   in List.foldl list_fold_func Dict.empty main_list
       |> Dict.map (\ _ list -> list_remove_duplication list)
```

Figure 8.17: Other auxiliary functions for pattern matching.

8.7 Building a Complied Version

We can build the complied version of Phometa by go to the top directory of Phometa repository then execute `./scripts/build.sh`. This will process as the following

- create `build/` (if doesn't exist) or clean it (if doesn't empty)
- compile(LaTeX) the document in `doc/` and copy the main PDF file to `build/`
- compile(Sass) `src/style.scss` to `build/style.css`
- copy the following files to `build/`
  - `src/naive.js`
  - `src/repository.json`
  - `src/phometa-server.py`
  - `logo.png`
- compile(Elm) `src/Main.elm` to `build/phometa.html`
- compress everything in `build/` to `build/phometa.tar.gz`

Please note that Elm compilation will consult `elm-package.json` for meta data. The dependency of target Elm file will be complied recursively. It can be outputted as HTML file (for standalone application) or JavaScript file (to embedded it as an element of bigger application).

8.8 Back-End Communication

There is a python server template from `https://gist.github.com/UniIsland/3346170` which allows client to download and upload local files directly in the server.

This server template, with some modification, allows us to create a local server to serve `phometa.html` which is the main application that user will interact with. `phometa.html`, in tern, will load further resource such as `naive.js` (for naive code that can't be done directly in Elm) and `style.css` (for theme of Phometa).

Once the setup is completed, Phometa will automatically load the repository by sending Ajax request to load `repository.json` from the server, and then, it will decode this json file to get instance of `Package` and will set this to `model.root_package`.

Most of operation after this doesn't require further communication with the server. Nevertheless, if user would like to save the repository, Phometa will encode `model.root_package` and send Ajax request to upload this as `repository.json`.

8.9 Testing / Continious Integration

Functional programming makes testing really easy. In order to test a function, we need to create a test suite and call that function with several inputs then check whether the outputs are the same as expectation or not.

For example, we want to test a function `list_insert` as the following

```
list_insert : Int -> a -> List a -> List a
list_insert n x xs =
  if n <= 0 then x :: xs else
    case xs of
      []      -> [x]
      y :: ys -> y :: (list_insert (n - 1) x ys)
```

Figure 8.18: Function `list_insert` from `src/Tools/Utils.elm`

We could write a test suite for `list_insert` in the test suite of its module like this

```
module Tests.Tools.Utils where

import ElmTest exposing (Test, test, suite, assert,
                         assertEqual, assertNotEqual)
import Tools.Utils exposing(..)

tests : Test
tests = suite "Tools.Utils" [
  suite "list_skeleton" [...],
  suite "list_insert" [
    test "n < 0" <|
      assertEqual [7, 6, 4, 9] (list_insert (-1) 7 [6, 4, 9]),
    test "n = 0" <|
      assertEqual [7, 6, 4, 9] (list_insert 0 7 [6, 4, 9]),
    test "n in range" <|
      assertEqual [6, 4, 7, 9] (list_insert 2 7 [6, 4, 9]),
    test "n >= length" <|
      assertEqual [6, 4, 9, 7] (list_insert 10 7 [6, 4, 9]),
    ... ],
  suite "parity_pair_extract" [...],
  suite "remove_list_duplicate" [...],
  ... ]
```

Figure 8.19: The test suite for module `src/Tools/Utils.elm`

A test suite of each module will be included in the main test suite as the following

```
module Main where

import Task
import Console exposing (run)
import ElmTest exposing (Test, suite, consoleRunner)

...
import Tests.Tools.Utils
...

tests : Test
tests = suite "Tests" [ ... , Tests.Tools.Utils.tests, ... ]

port runner : Signal (Task.Task x ())
port runner = run (consoleRunner tests)
```

Figure 8.20: The main test suite which is at `tests/Tests/Main.elm`

To run the main test suite, go to top level of Phometa repository
then execute `./scripts/tests.sh` which will be processed as the following.

- compile `tests/Tests/Main.elm` to `raw-tests.js`
- convert `raw-tests.js` to `tests.js` using external bash script[7].
- execute `tests.js` by *Node.js*, this will show the testing result.

Please note that the test suite doesn't have 100% coverage and functions related to HTML are not testable using this method.

Travis CI is used for continuous integration. Basically, every time when something is pushed to *GitHub*, Travis CI will check `.travis.yml` this will result in the main test suite being executed in virtual machine using similar testing process above.

[7] https://raw.githubusercontent.com/laszlopandy/elm-console/master/elm-io.sh

Chapter 9

Evaluation

9.1 Users Feedback — discuss with friends

On the 25th of May 2016, it was the first day of a project fair where students can demonstrate their work to other students and get a feedback so I went there and discuss about our projects. At this stage, the implementation is finished with Simple Arithmetic and Propositional Logic included in the standard library.

I started showing my project by explaining about Phometa background and Simple Arithmetic using chapter 3 and 4 on this report. Then I asked them do to exercises on chapter 4 by having me as helper. All of them understood Phometa and was able to proof a theorem. Finally, I asked them to try Propositional Logic, some of them really interest but of them didn't want to.

From my observation, all of them were comfortable to proof by clicking options from keymap pane rather than using keyboard shortcut. They also forgot to use searching pattern to select options faster.

There were a few parts of user interface that were not trivial enough, they needed to ask me what to do next, this should be fine if user have time to read whole tutorial.

On the bright side, most of them said that they really like the way that underlines was use to group sub-term rather than brackets (although they needed some time to familiar with it), they also said that the proof is quite easy to read and it will benefit newcomer.

There were several improvements that they suggest. Some of suggestions were easy to change (e.g. theorems should state its goal on header as well) so I changed it already. Some of other suggestions were quite big (e.g. make it mobile friendly and have a proper server) which can be considered as future works. We also managed to find some bugs[1] that I never found before, this gave me an opportunity to fix it in time.

[1]These bug are related Html and CSS rendering i.e. they are not related to Phometa internal.

9.2 Users feedback — discuss with junior students

The project fair happened again on the 1st of June. This time, I gave a demonstration to the first and second years students who study here.

The process was the same as last week, I started the demonstration by giving Phometa overview using background and Simple Arithmetic chapters and let them try natural deduction in Propositional Logic. At first, they usually make a mistake especially on how to construct a term, for example,

- One of students wrote a whole propositional formula in plain-text because he thought that is how Phometa accept a term.
- A few student students forgot to hit `Return` when finish entering a meta variable. Actually, this should be fine but I disabled such a feature because it has a problem with Phometa internal mode.
- One group of students accidentally clicked theorem reset button because they thought that was undo button. This reset button there was quite dangerous as it was nearly equivalent to delete button.

These problem occur only for the first time, once they understand more on the tool, they can use it easily. In fact, the tutorial that I have written already most of these mistake, but I didn't ask students to read thought the entire tutorial due to time constrain.

On the other hand, I was surprise on how fast first year students[2] can construct Gentzen style's natural deduction in Phometa given that they know how to do Fitch style's natural deduction in Pandora represented in first year Logic course. Moreover, one of second students told me that he didn't particularly like derivation tree[3] but Phometa managed to make him enjoy constructing this derivation tree.

In conclusion, everybody who try Phometa in the project fair was able to use Phometa properly when they got a proper guidance. Some students really enjoy proving thing Phometa and some students believe that Phometa could have helped them during their coursework that require constructing derivation tree in the past.

9.3 Strengths

- Phometa specification itself is more powerful than traditional derivation system because it has extra features such as cascade premise and meta-reduction. Thus, be able to support more formal systems than traditional one.

[2]Most of them were tutees in Personal Mathematics Tutorial(PMT) group that I took a role as Undergraduate Teaching Assistant(UTA) in this academic year.

[3]Second year students had studied derivation tree in the class already.

- It has less steep learning curve than mainstream proof assistants because the specification is small enough for user to learn in short time and all of component are diagram based which is easier to understand than sequence of characters.
- If a term can be construct, it is guaranteed to be well form. And if it is a goal of complete theorem (or lemma) it is definitely valid based on soundness on rules on that formal systems.
- The repository of phometa is always in consistent state. Phometa is quite caution when the repository is being modified, for example, theorem can apply only a rule or a lemma that has been locked i.e. it is impossible that its dependencies will be changed, another example is when a node is deleted, phometa will delete all of node that depend on it as well[4]. This is opposite to text-based proof assistants where user have full control over repository, if the repository is in inconsistent, the compiler will rise an error and user can fix it.
- Lemmas allow reuse of proofs so no need for duplication. User can select to do forward style proving (lots of small lemmas as steps of a proof) or backward style proving (a few big theorems).
- It supports unicode input method and doesn't have reserved words so formal system can be constructed in more mathematical friendly environment.
- It is web-application so it can run on any machine that support web browser. One might argue that it required Python for back-end but most of machine support Python out of the box anyway.

9.4 Limitation

- It is hard to extend a formal system at the moment because Phometa doesn't allow grammar to inherit choices from another grammar. If user want to extend a formal system, they need to create a new one from scratch. For example, first order logic cannot be built from an existing propositional logic. If user build grammars of first order logic from scratch, existing propositional logic rules cannot be extended to support first order logic anyway.
- Phometa doesn't support automation well i.e. when user construct a proof, they need to tell which rule or which lemma will be used explicitly. Guessing each step and automating the tree is possible, mainstream proof assistants such as Coq and Isabelle have done it, however, it requires lots of heuristics and cleaver tricks, this is unrealistic to implement due to project time frame but it is good consideration for future work.

[4]Of course, it will ask for confirmation first whether user want delete all of these or not.

- Each web-browser supports different set of keyboard shortcuts. It is very hard for Phometa to find such keystrokes that are not visible characters and not keyboard shortcuts of any web-browser. So I end-up using `Alt` combined with a visible character to create Phometa shortcut. This might have unwanted side effect but at least it work reasonably well with Google Chrome[5] under a condition that the window containing Phometa have only one tab, so it will not suffer from `Alt-1..9` are using for switch tab. However, this is not such a serious problem since user can always click a command in keymap pane directly.
- The entire repository must be loaded into Phometa when it starts. This impacts scalability where repository is large since JavaScript can run out of memory. This is not usually a problem of text-based proof assistants since it verify a theorems one by one and doesn't need to put everything in memory at once.
- Phometa required user to start a local server for individual use. It doesn't have a proper server where user can enter a link use it directly. To implement such a proper server, it requires user accounts and database to manage users repositories, although it is possible to implement but it seems to overkill method and doesn't match project objective, hence it has lower priority than other feature and hasn't been done.
- Directly modify `repository.json` before it is loaded into Phometa could result in undefined behaviour. This is because Phometa currently doesn't have mechanism to verify consistency of repository before it will be loaded. This shouldn't cause any problem if user only interact with repository via Phometa interface and not try to hack repository file directly.

[5]To be precise, Chromium web-browser.

Chapter 10

Conclusion

At the end of this project. Phometa has been designed and been implemented up to the level that is ready to use by anybody with a decent standard library and tutorial. This, in tern, satisfies all of objective stated in introduction chapter. In addition, I also believe that Phometa on this state is a potential replacement for derivation-tree's manually-drawing so people don't have to suffer from it tedious process and error prone anymore.

10.1 Lesson Learnt

Time management for research project is one of many thing that I have learnt during this project. I learnt that tasks are always take time twice or thrice longer than expectation so it is vital to spare plenty of time before the deadline. More importantly, I learnt that better idea of feature always come after we start to implement something. It is quite hard to decide whether Phometa should include some certain feature or not. It is about a tread off between usefulness of the feature and the risk of the project being not finish in time. This kind of features usually came near the end of implementation where I knew exactly what Phometa should be. This is bad because if I accepted the feature, this would take sometime to implement and edit related part of this report, which in tern, would impact the entire schedule of the plan. So I usually take it as future work as described in next section.

I also learnt to believe in myself being capable to building something I dream of. Formal proof always be my favourite topic since I studied Logic in the first year. One day, I was drawing a derivation tree for a coursework, I had the idea of this project. At the first time it seemed too scary because it is about building a proof assistant from scratch, however after evaluated proof of concept, it turned out to be feasible. So I decide to start it and approached my supervisor.

Most importantly, I learnt many thing regarding to formal proof from this project which is relevant to the topic that I want to do for PhD (Dependent Type Theory). This gave me more familiarity and confident in that field. Oppositely, curiosity on the field motivated me to work on this project better since I know that this kind of knowledge gained during the project will be useful later for sure.

10.2 Future Works

Although Phometa has been designed and implemented up to satisfactory level. There are still plenty of room for improvement. For example,

- **Making Grammars extensible** — Grammars should be able to extend another grammar in similar manner to how class in Object Oriented Programming extend other classes, for example, grammar of proposition in First Order Logic could extend grammar of proposition in Propositional Logic. This allows polymorphism where a term of extending grammar could be proven using rules or lemmas related based grammar.
- **Plain-text as alternative term input method** — When constructing a term, user should be able to write some term in plain-text and Phometa will try to parse it according to formats of that grammar.
- **Making Theorems construction become more automatics** — When constructing a theorem, Phometa should be able to guess what rules or lemmas should be applied next (similar to Isabelle).
- **Importation between Modules** — Modules should be able to import some nodes from other modules, this improve scalability since a formal system can be expand across multiple modules.
- **Exportation to LaTeX** — Grammars, Rules, and Theorem should be exportable to LaTeX source code. So it can be used further other document such as report or presentation.
- **Repository Verification on Loading** — Creating a function that check whether the repository is in valid state or not. Normally, it is not necessary as Phometa will always be in consistent state, but when the you want use other people formal system by obtaining `repository.json`, it is not a grantee that someone will not modify that repository directly. So it is better to have mechanism to check consistency of the repository anyway.

There are also other minor improvements such as adding new formal systems to standard library, making notification message become more informative, deploy phometa to a proper server, making user interface become more mobile friendly, and making a theme and keyboard shortcut become configurable.

Bibliography

[1] Evan Czaplicki et al. *Elm offical website*. http://www.elm-lang.org. [Online; accessed 3-February-2016].

[2] Encyclopedia Britannica. *Formal system*. http://www.britannica.com/topic/formal-system. [Online; accessed 29-January-2016].

[3] Krysia Broda et al. "Pandora: A Reasoning Toolbox using Natural Deduction Style". In: *Logic Journal of the IGPL* 15 (2007), pp. 293–304.

[4] University of Cambridge and Technische Universität München. *Isabelle offical website*. http://isabelle.in.tum.de/index.html. [Online; accessed 15-May-2016].

[5] Chalmers and Gothenburg University. *Agda offical website*. http://wiki.portal.chalmers.se/agda/pmwiki.php. [Online; accessed 29-January-2016].

[6] Thierry Coquand. *Thierry Coquand homepage*. http://www.cse.chalmers.se/~coquand. [Online; accessed 29-January-2016].

[7] Imperial College London Department of Computing. *Pandora offical website*. https://www.doc.ic.ac.uk/pandora/. [Online; accessed 3-June-2016].

[8] University of Edinburgh. *Proof General offical website*. http://proofgeneral.inf.ed.ac.uk. [Online; accessed 18-May-2016].

[9] Olivier Gasquet, François Schwarzentruber, and Martin Strecker. "Tools for Teaching Logic: Third International Congress, TICTTL 2011, Salamanca, Spain, June 1-4, 2011. Proceedings". In: ed. by Patrick Blackburn et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. Chap. Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students, pp. 85–92. ISBN: 978-3-642-21350-2. DOI: 10.1007/978-3-642-21350-2_11. URL: http://dx.doi.org/10.1007/978-3-642-21350-2_11.

[10] *Homotopy Type Theory Repository*. https://github.com/HoTT/HoTT. [Online; accessed 18-May-2016].

[11] *Homotopy Type Theory Repository — Agda alternative*. https://github.com/HoTT/HoTT-Agda. [Online; accessed 18-May-2016].

[12] Inria. *Coq offical website*. https://coq.inria.fr. [Online; accessed 29-January-2016].

[13] Inria. *CoqIde offical website*. https://coq.inria.fr/cocorico/CoqIde. [Online; accessed 18-May-2016].

[14] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* New York, NY, USA: Oxford University Press, Inc., 1994. ISBN: 0-19-853835-9.

[15] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[16] Institut de Recherche en Informatique de Toulouse. *Panda offical website.* https://www.irit.fr/panda/. [Online; accessed 3-June-2016].

[17] Massachusetts Institute of Technology. *Logitext offical website.* http://logitext.mit.edu/. [Online; accessed 3-June-2016].

[18] Wikipedia. *Curry Howard correspondence.* https://en.wikipedia.org/wiki/Curry-Howard_correspondence. [Online; accessed 29-January-2016].

[19] Wikipedia. *Formal system.* https://en.wikipedia.org/wiki/Formal_system. [Online; accessed 29-January-2016].