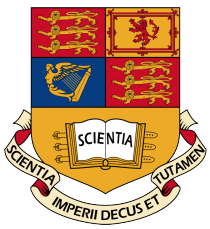


Imperial College London

Department of Computing

Phometa — a visualised proof assistant that builds a formal system and proves its theorems using derivation trees

Author:
Gun Pinyo

Supervisor:
Dr. Krysia Broda

Second Marker:
Prof. Alessio R. Lomuscio

June 16, 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 rewritten (by unification when derivation rule is applied). Even when the tree is completed, there is nothing to guarantee that the tree is error free.

This thesis describes *Phometa* which is a proof assistant that allows a 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 an educational platform for students to learn certain formal systems provided in a standard library. Alternatively, it also can be used as an experimental sandbox where a user implements their own formal system and tries to reason about it.

Phometa is a web application so components such as terms and derivation trees can be rendered nicely in a web browser and users can interact with these components directly by clicking a button or pressing keyboard shortcuts. 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 a derivation-trees's manual-drawing and easy enough to be used by anyone. Its standard library also includes 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 advice.

I also would like to thank my second marker, Prof. Alessio R. Lomuscio, for his invaluable advice on the 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 **4**
- 1.1 Motivation . . . 4
- 1.2 Objectives . . . 6
- 1.3 Achievement . . . 6

2 Related Work **7**
- 2.1 Text-Base Proof Assistants . . . 7
  - 2.1.1 Coq . . . 7
  - 2.1.2 Agda . . . 8
  - 2.1.3 Isabelle . . . 9
- 2.2 Visualised Proof Assistant . . . 10
  - 2.2.1 Logitext . . . 10
  - 2.2.2 Panda . . . 11
  - 2.2.3 Pandora . . . 12

3 Background **13**
- 3.1 Formal Systems . . . 13
- 3.2 Backus-Naur Form . . . 13
- 3.3 Meta Variables and Pattern Matching . . . 16
- 3.4 Derivation of Formal Systems . . . 17

4 Example Formal System — Simple Arithmetic **20**
- 4.1 First time with Phometa . . . 20
- 4.2 Grammars . . . 22
- 4.3 Rules . . . 23
- 4.4 Theorems and Lemmas . . . 25
- 4.5 More complex theorem . . . 30

5 Further Examples of Formal Systems **35**
- 5.1 Propositional Logic . . . 35
- 5.2 Typed Lambda Calculus . . . 40
  - 5.2.1 Terms and Variables . . . 40
  - 5.2.2 Side Conditions . . . 41

5.2.3 β Reduction . . . 41
5.2.4 Simply Types . . . 43

6 Specification **46**
6.1 User Interface Overview . . . 46
6.2 Auto-Complete Input Box . . . 48
6.3 The proofs repository . . . 50
6.4 Node Comment . . . 52
6.5 Node Grammar . . . 53
6.6 Root Term . . . 54
6.7 Node Rule . . . 56
6.8 Node Theorem . . . 58
6.8.1 Theorem Construction . . . 58
6.8.2 Theorem Fold/Unfold . . . 60
6.9 Meta Reduction . . . 61

7 Implementation **62**
7.1 Decision on programming language . . . 62
7.2 Model-Controller-View Architecture . . . 63
7.3 Model, Command, Keymap, and Action . . . 65
7.4 Structure of Proofs Repository . . . 66
7.5 Pattern Matching and Unification . . . 70
7.6 Testing / Continious Integration . . . 73

8 Evaluation and Conclusion **75**
8.1 Users Feedback — discuss with friends . . . 75
8.2 Users Feedback — discuss with junior students . . . 76
8.3 Strengths . . . 76
8.4 Limitation . . . 77
8.5 Lesson Learnt . . . 78
8.6 Future Works . . . 79
8.7 Conclusion . . . 80

Appendices **81**

A Example Formal System — Propositional Logic **82**
A.1 Grammars . . . 82
A.2 Hypothesis rules . . . 85
A.3 Main Rules . . . 87
A.4 Classical Logic . . . 90
A.5 Validity of Proposition . . . 92
A.6 Context manipulation . . . 93
A.7 How to build Grammars and Rules . . . 95
A.7.1 Recreation of `Prop` . . . 95

A.7.2 Recreation of `or-elim` . . . 97
A.7.3 Recreation of `hypothesis` . . . 99

B Example Formal System - Typed Lambda Calculus **101**
B.1 Terms and Variables . . . 101
B.2 Side Conditions . . . 102
B.3 β Reduction . . . 103
B.4 Simply Types . . . 106

C Exercises on Examples Formal Systems **109**
C.1 Simple Arithmetic . . . 109
C.2 Propositional Logic . . . 109

D The Phometa Repository **112**
D.1 Installation and Startup . . . 112
D.2 The Anatomy of Phometa Repository . . . 113
D.3 Building a Compiled Version . . . 114
D.4 Back-End Communication . . . 114

E Supplementary Source Code **115**
E.1 Pattern Matching and Unification . . . 115

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[13], Agda[6], and Isabelle[5] which are suitable for extreme use cases of complex proofs. Nevertheless, they have a steep learning curve and have specific meta-theories behind them, for example, Coq has Calculus of Inductive Construction (CIC), Agda has Unified Theory of Dependent Types[16][15] both of 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 points 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, leading to a tree-like structure of the final proof, which 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 — conflicting with the ambition to use proof assistant at the first place.

So I decided to create a proof assistant called *Phometa* to solve this derivation-tree manual-drawing problem. To be precise, Phometa is proof assistant that allows users 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-formed term.
  For example, a simple arithmetic expression can be constructed by a number *or* two expressions adding together *or* two expressions multiplied 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 MULT-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

- Learning one of many existing formal systems provided in Phometa standard library and try to prove some theorem in that formal system.
- Creating their own formal system or extending an existing formal system, then do some experiments about it.

In order to make Phometa easy to use, it is designed to be a web-based application. Users will interact with Phometa mainly by clicking buttons and pressing keyboard-shortcuts. This has advantages over a traditional proof assistant because it is easier to read, ill-from terms never occur, and guarantees 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 system and reason about it.
- To show that most of formal systems have a similar meta-structure which can be implemented using a common framework.
- To show advantages of a visualised proof assistant over a traditional one.

1.3 Achievement

- Finished designing Phometa specification (chapter 6) in such a way to keep it simple yet be able to produce a complex proof.
- Finished implementing Phometa (chapter 7). All of basic functionality is working.
- Encoded the following formal systems as standard library in Phometa.
  - Simple Arithmetic (chapter 4)
  - Propositional Logic (chapters 5 and A)
  - Typed Lambda Calculus (chapters 5 and B)
- Wrote a tutorial for newcomers to use Phometa (chapters 3, 4, A, B, and C).

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 two 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 where a user writes everything in text-files and compiles 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 proof assistants have a plug-in to a mainstream text editor, so users can use their favourite text editor with full performance.

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

2.1.1 Coq

Coq[13] is one the most famous proof assistants. It is based on the Calculus of Inductive Constructions (CIC)[1] developed by Thierry Coquand[7].

Coq has customisable tactics which are commands that transform a goal into smaller-sub goal (if any), this makes proving process become faster compared to other proof assistants. In contrast, tactics reduce readability, and a 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[9] which is a plugin on Emacs[2]. Nevertheless, Coq has its own editor called CoqIde[14] that newcomers can use without learning Emacs.

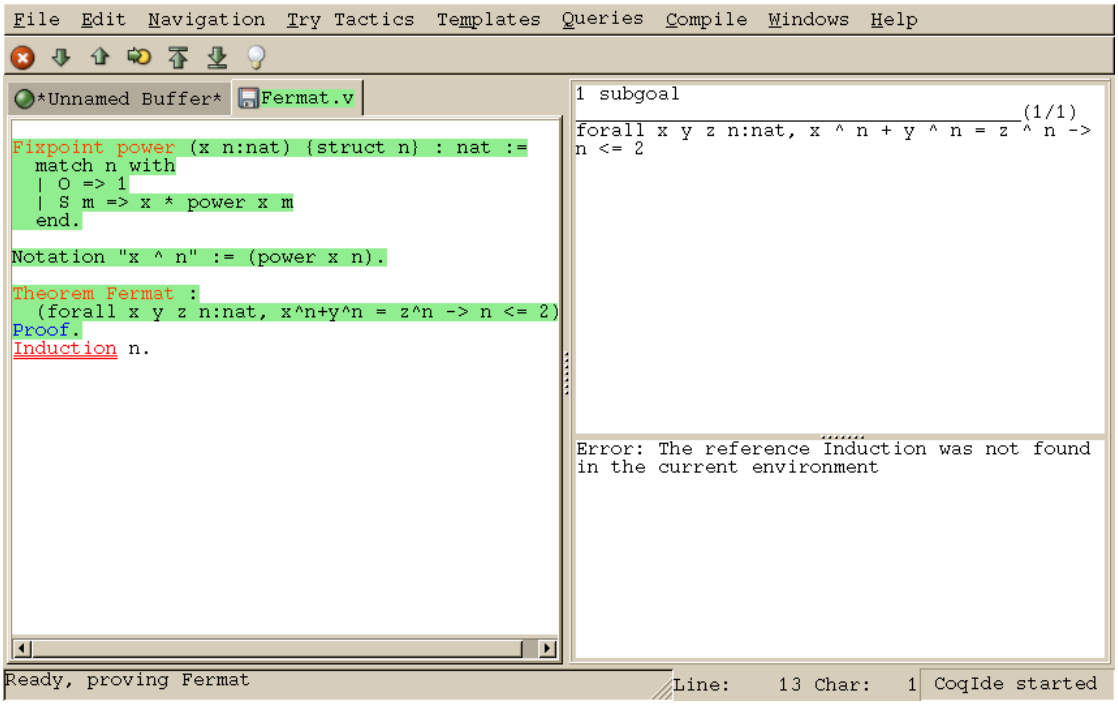


Figure 2.1: Screenshot of Coq (using CoqIde, Credit: [13]) — The left pane is file content and Upper right pane is the current goal which is changed depending on where the cursor point on file content.

2.1.2 Agda

Adga[6] is (dependently typed) functional programming which can be seen as a proof assistant as well. It is based on Unified Theory of Dependent Types[16][15] similar to Martin Lof Type Theory.

Its proving technique relies on the Curry-Howard correspondence which states that there is duality between computer programs and mathematical proofs[19]; for example functions corresponded to implication, product types 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 a less steep learning curve compared other proof assistants such as Coq. This is because users don't need to learn about proving in the system since it is the same as programming. In contrast, it doesn't have fancy tactic system so the proving process is slower.

In term of popularity, it is less popular than Coq, however, some projects such as Homotopy Type Theory[11][12] 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[5] 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 provides a higher order logic environment that is ready for a big application.

One of the main advantages of Isabelle is its readability; the proofs will be constructed by a language called *Isar* which is designed in such a way that it can 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 improves 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.

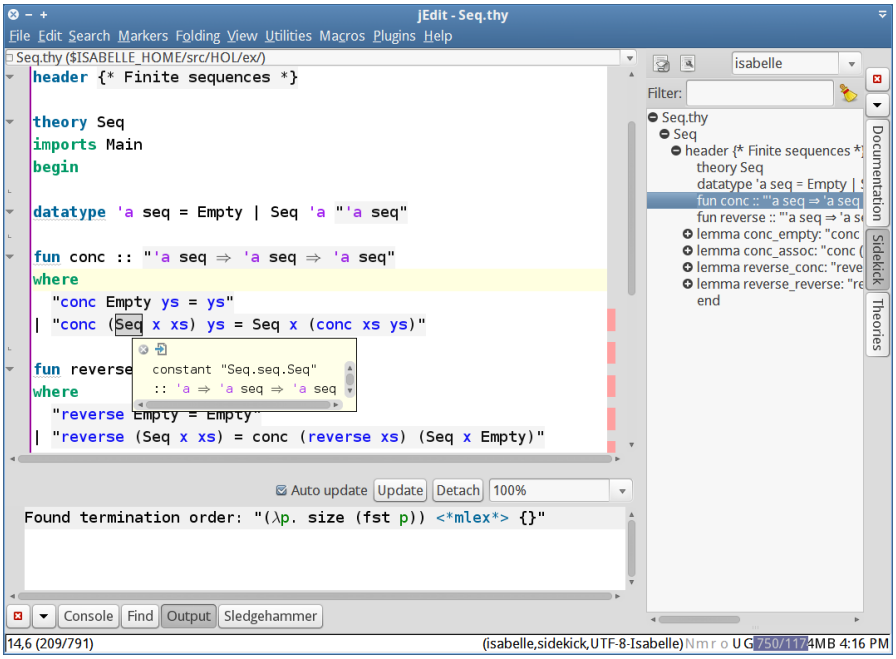


Figure 2.3: Screenshot of Isabelle (using Isabelle/jEdit, Credit: [5]) — The top-left pane shows file content and the right pane shows content structure.

2.2 Visualised Proof Assistant

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

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

2.2.1 Logitext

Logitext[18] 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 is one of a way to construct a derivation system. Logitext uses Coq internally to check validity of proof steps.

Logitext is very easy to use, the user usually only needs 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 users cannot extend the system.

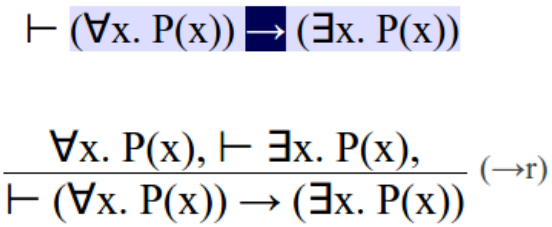


Figure 2.4: Screenshots of Logitext — You can see that a rule could be invoke by just clicking a logical connector.

2.2.2 Panda

Panda[17][10] (**P**roof **A**ssistant for **N**atural **D**eduction for **A**ll) is a graphical proof assistant that can be used to prove first order logic using Gentzen style's natural deduction.

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

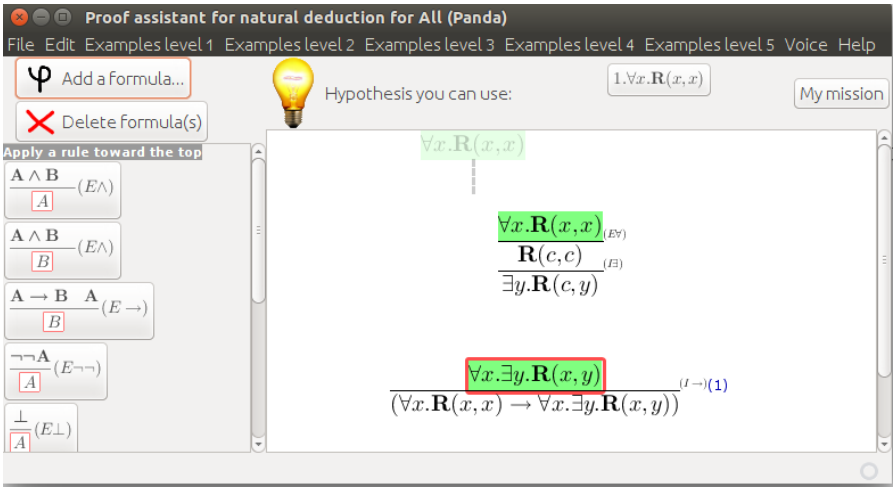


Figure 2.5: Screenshots of Panda — User can expand the current tree by selecting a rule on the left pane and can drag and drop formula around to connect it to another one.

2.2.3 Pandora

Pandora[8][3] (**P**roof **A**ssistant for **N**atural **D**eduction using **O**rganised **R**ectangular **A**reas) is a 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 uses boxes (Fetch style) rather than derivation trees (Gentzen style). It also have a comprehensive document for newcomers as well.

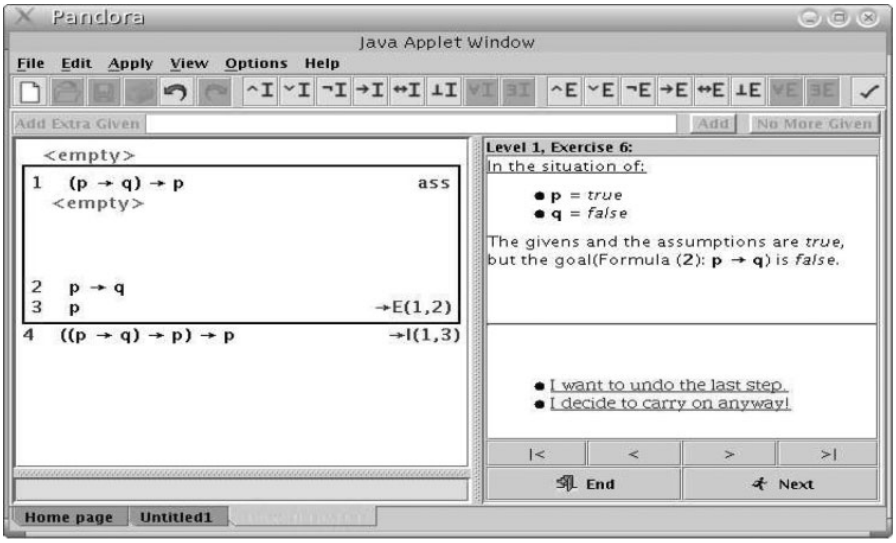


Figure 2.6: Screenshots of Pandora (Credit: [3]) — The left pane shows current proof, user can use a rule by clicking on the upper pane.

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 Systems

A formal system is any well-defined system of abstract thought based on a mathematical model[20]. Each formal system has a formal language composed of primitive symbols[1] acted by certain formation[2].

Informally, it 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 the 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.

2 is `<Digit>` (3^{rd} choice)

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

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

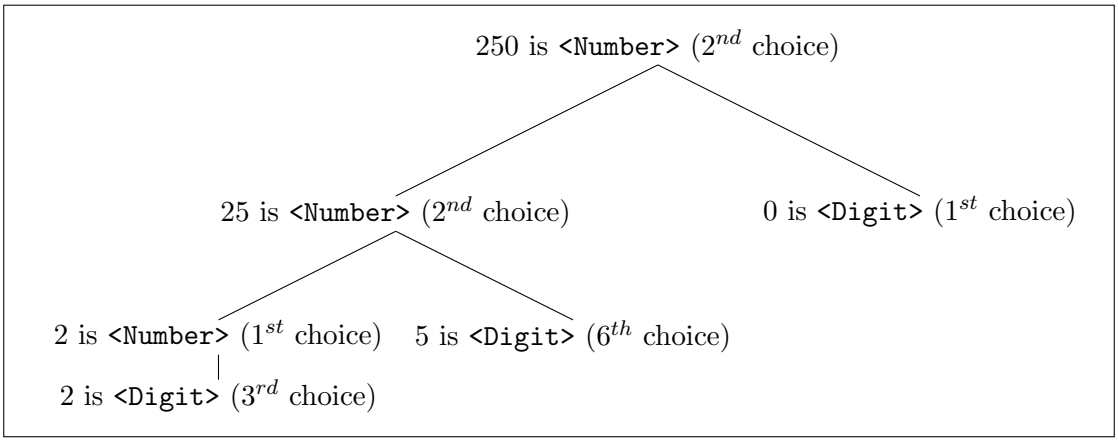


Figure 3.3: This diagram explains why 250 is a term of `<Number>`

- A term of `<Expr>` can be either
  - `<Number>`
  - two `<Expr>`s concatenated using '(' '+' ')'
  - two `<Expr>`s concatenated using '(' '×' ')'

  Please note that we need brackets around '+' and '×' 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 something like $(12 + 0) \times 6$.

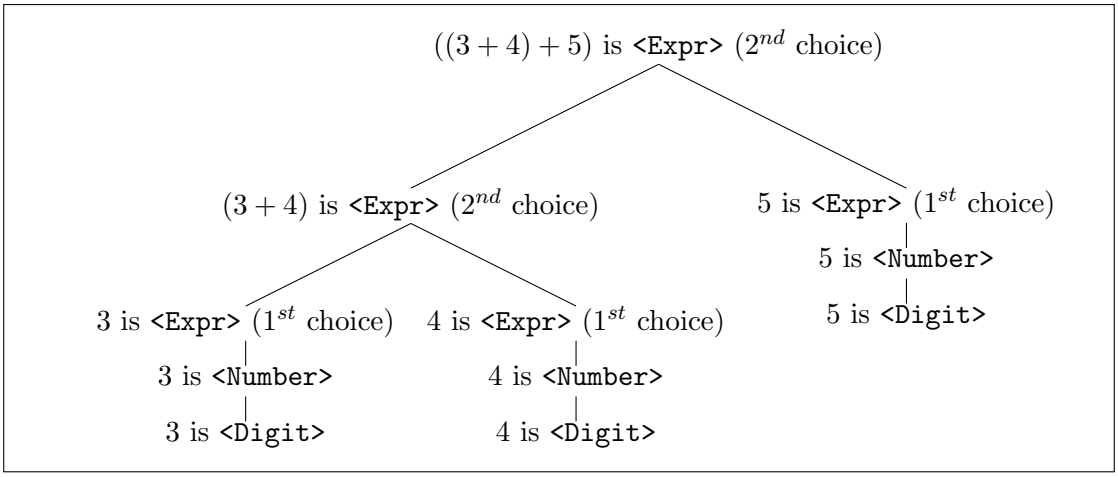


Figure 3.4: This diagram explains why $((3 + 4) + 5)$ is a term of `<Expr>`

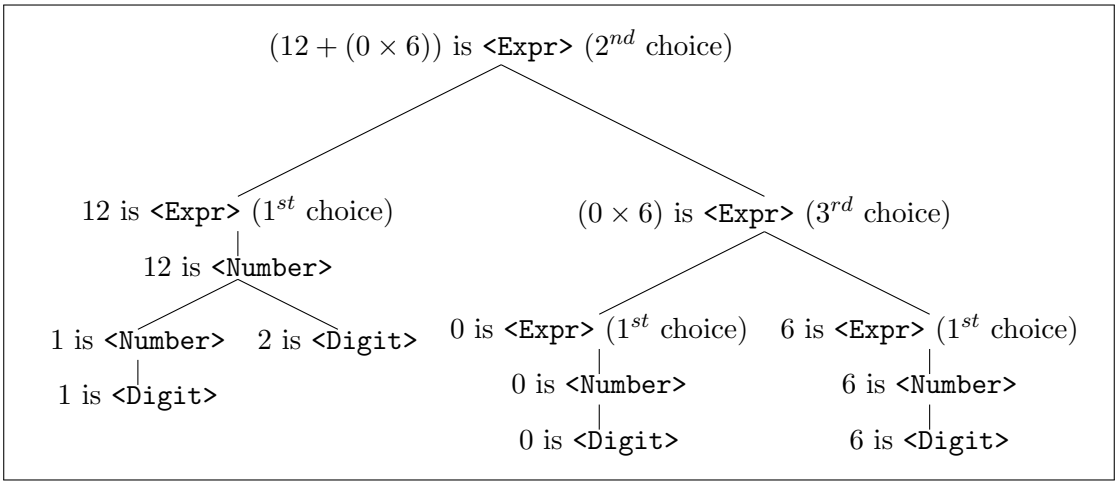


Figure 3.5: This diagram explains why $(12 + (0 \times 6))$ is a term of `<Expr>`

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

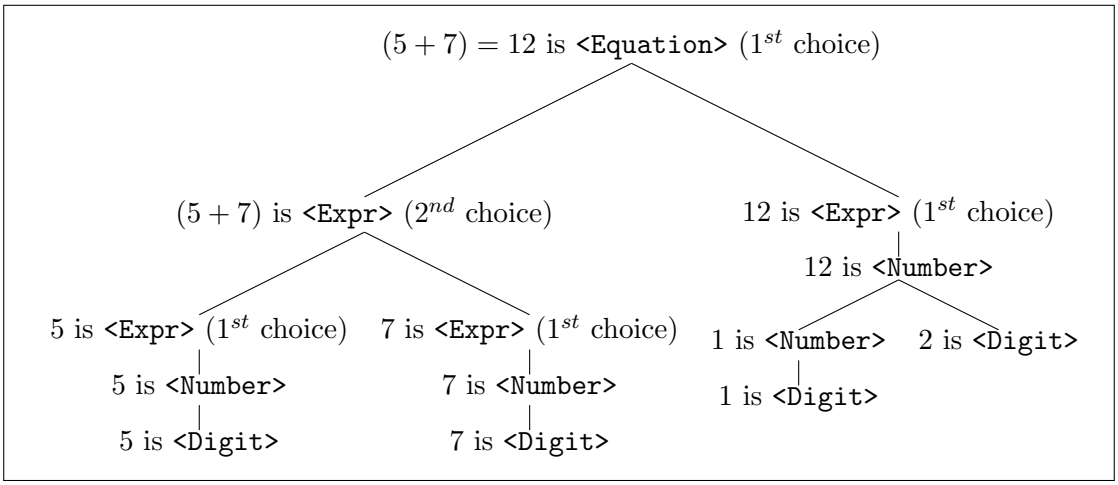


Figure 3.6: This diagram explains why $(5 + 7) = 12$ is a term of `<Equation>`

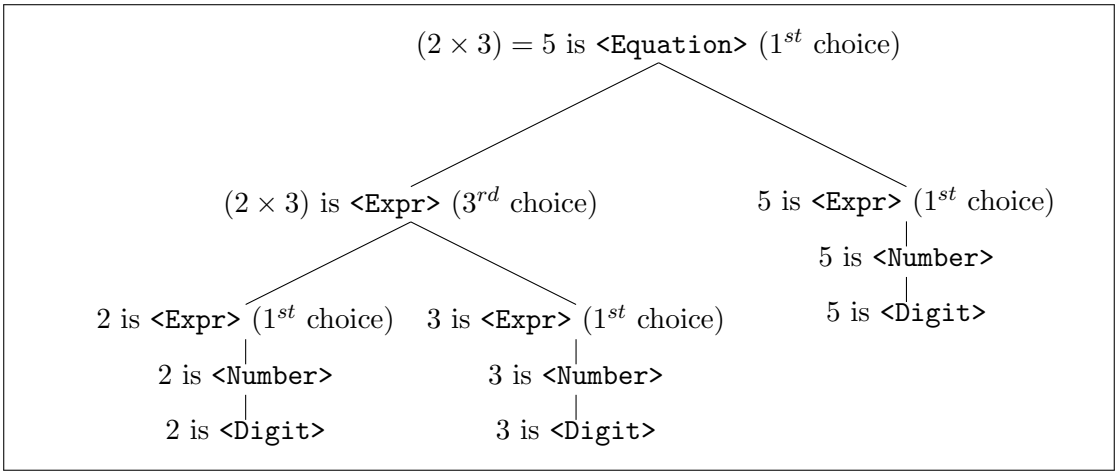


Figure 3.7: This diagram explains why $(2 \times 3) = 5$ is a term of `<Equation>`. Please note that this construction is purely syntactic so wrong equation is acceptable.

3.3 Meta Variables and Pattern Matching

Meta variables are arbitrary sub-terms embedded inside a 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 a special form of "*variable as arbitrary term*".

Meta variables help us to represent statements in a more general manner. For example, "*the same expressions added 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 an algorithm 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 an 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 a derivation system. First, we have a set of derivation rules that has format as the following

$$\text{RULE-NAME}\ \frac{Premise_1 \qquad Premise_2 \qquad Premise_3 \qquad \ldots \qquad Premise_n}{Conclusion}$$

Figure 3.8: Structure of derivation rule.

This says 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 and then proof its premises.

Derivation rules of the 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 matched against the current goal. If it is pattern matchable then meta variables in premises are substituted respect to the pattern matching[2]. 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

[2]If some meta variables of premises doesn't exist in substitution list then we are free to substitute by anything e.g. in EQ-TRAN, we can replace z with arbitrary `<Expr>`, this free substitution gives a better control over premises, which is important when we apply rules to sub-goals.

$$\frac{(3+4)=(4+3) \qquad 5=5}{((3+4)*5)=((4+3)*5)}\ \text{MULT-INTRO}$$

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

$$\frac{\dfrac{}{(3+4)=(4+3)}\ \text{ADD-COMM} \qquad \dfrac{}{5=5}\ \text{EQ-REFL}}{((3+4)*5)=((4+3)*5)}\ \text{MULT-INTRO}$$

Figure 3.11: Example of derivation tree.

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{((w \times (x+y)) \times z)=(w \times ((x+y) \times z))}\ \text{MULT-ASSOC}}{\vdots}}{((w \times (x+y)) \times z)=(w \times ((x \times z)+(y \times z)))}\ \text{EQ-TRAN}}{\vdots}}{(((w \times x)+(w \times y)) \times z)=(w \times ((x \times z)+(y \times z)))}\ \text{EQ-TRAN}$$

$$\dfrac{\dfrac{\dfrac{}{(w \times (x+y))=((w \times x)+(w \times y))}\ \text{DIST-LEFT}}{((w \times x)+(w \times y))=(w \times (x+y))}\ \text{EQ-SYMM} \qquad \dfrac{}{z=z}\ \text{EQ-REFL}}{(((w \times x)+(w \times y)) \times z)=((w \times (x+y)) \times z)}\ \text{MULT-INTRO}$$

$$\dfrac{\dfrac{}{w=w}\ \text{EQ-REFL} \qquad \dfrac{}{((x+y) \times z)=((x \times z)+(y \times z))}\ \text{DIST-RIGHT}}{(w \times ((x+y) \times z))=(w \times ((x \times z)+(y \times z)))}\ \text{MULT-INTRO}$$

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 concepts 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 fraction 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 the compiled 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

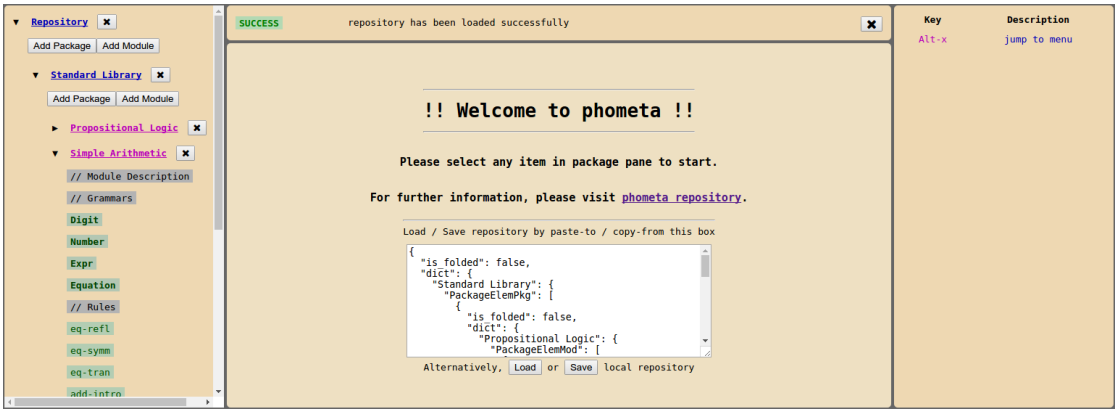


Figure 4.1: Screenshot of Phometa when you open it from web-browser.

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

Modules in Phometa are analogous to text files. 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 package 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 package pane directly to focus on a particular node.

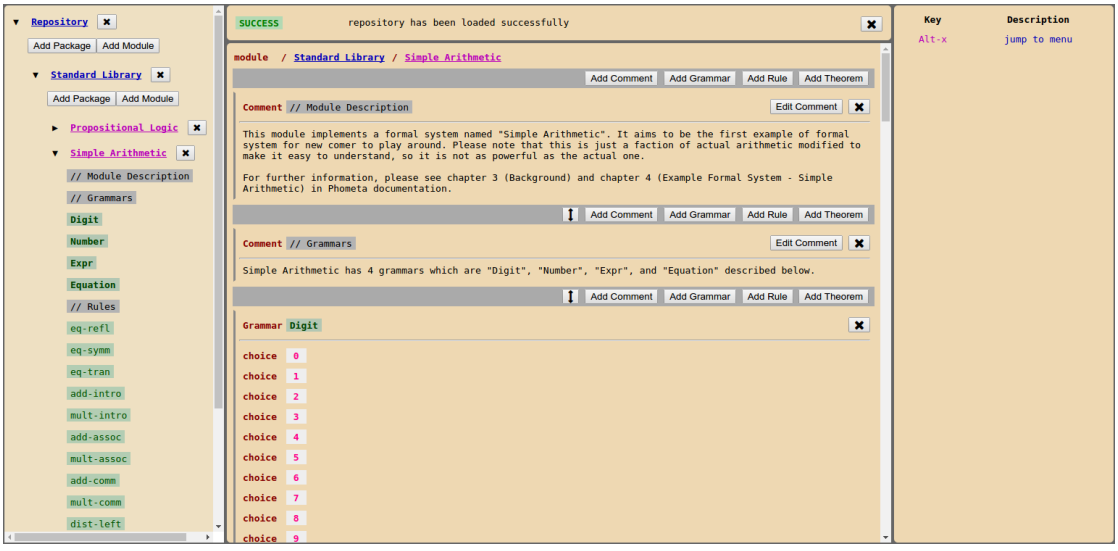


Figure 4.2: Screenshot of Phometa when you click "Simple Arithmetic" module.

In order to improve productivity, Phometa has several key-bindings specific to certain states of the program. Fortunately, users 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 newcomers 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 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 diagrams 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
```

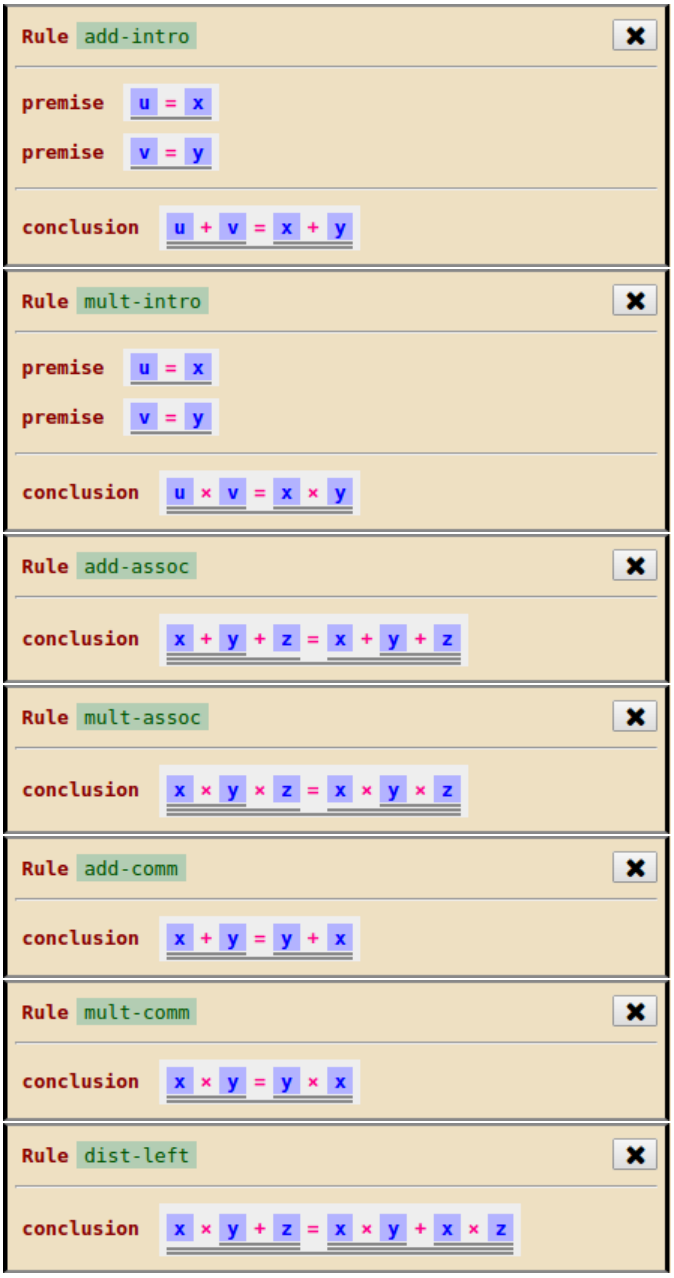


Figure 4.4: Rules of Simple Arithmetic

Most of the rules here are self explanatory but in rule `eq-tran` , there is an additional property named `parameter` (s) which is a meta veritable that appears 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

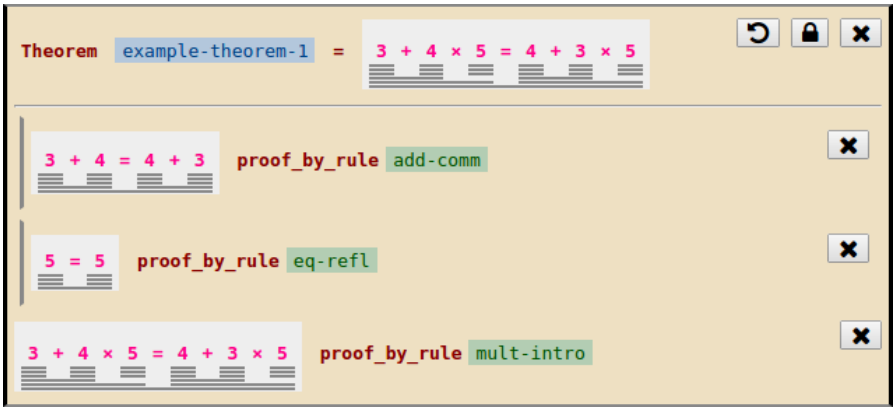
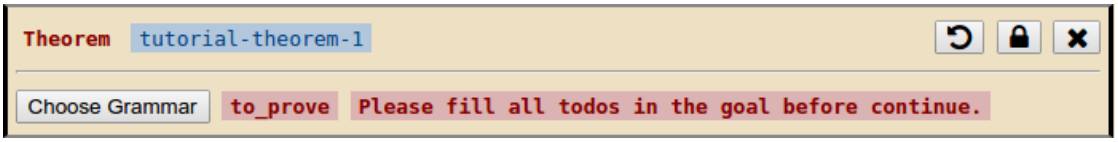


Figure 4.5: A theorem that show that $(3 + 4) \times 5 = (4 + 3) \times 5$.

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

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

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



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 turn, will change to input box. Now the keymap pane will look like this

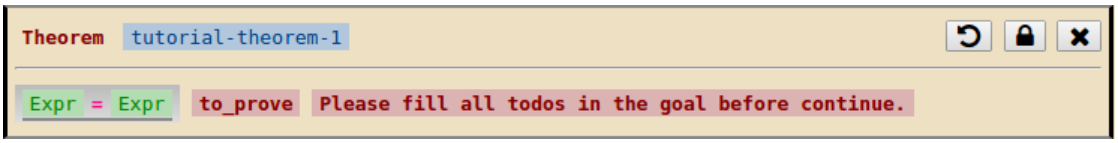
Key	Description
Alt-1	Digit
Alt-2	Number
Alt-3	Expr
Alt-4	Equation
Alt-u	search unicode
Alt-x	jump to menu
↑	quit root term

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 mode. Please note that this input box supports 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 clicks such a choice and the theorem will look like this



You may notice that the goal term has a grey background rather than white as before. This indicates 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

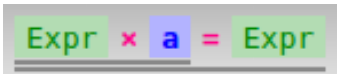
Again, there are three choices available which can be selected in a similar manner when we select a grammar. At the this stage you might wonder how to type "×" since it is unicode character. Well, we can go to unicode mode by press `Alt-u` as keymap pane suggest. Then keymap pane will look like this

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

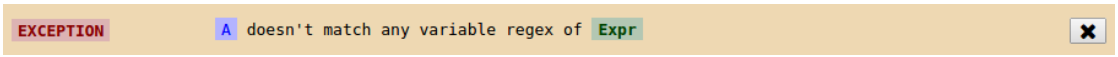
This allows us to search for a unicode character by using its LaTeX's math-mode name. Now type "times" in the input box, you should see "×" appear on keymap. Once you select it, the unicode mode disappear and put "×" in the input box, which in turn, filter other choices out so you can hit `Return` for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

Expr × Expr = Expr

Next we will focus a middle `Expr` . If we type string and hit `Return` , 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

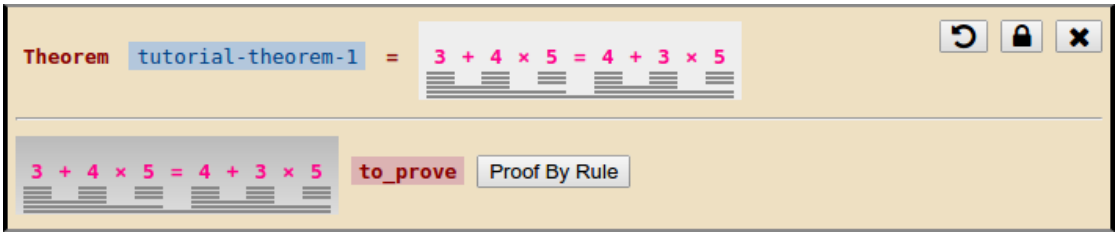


If we enter the name that that doesn't comply to a regular expression, it will do nothing and prompt an exception message above the main pane as the following

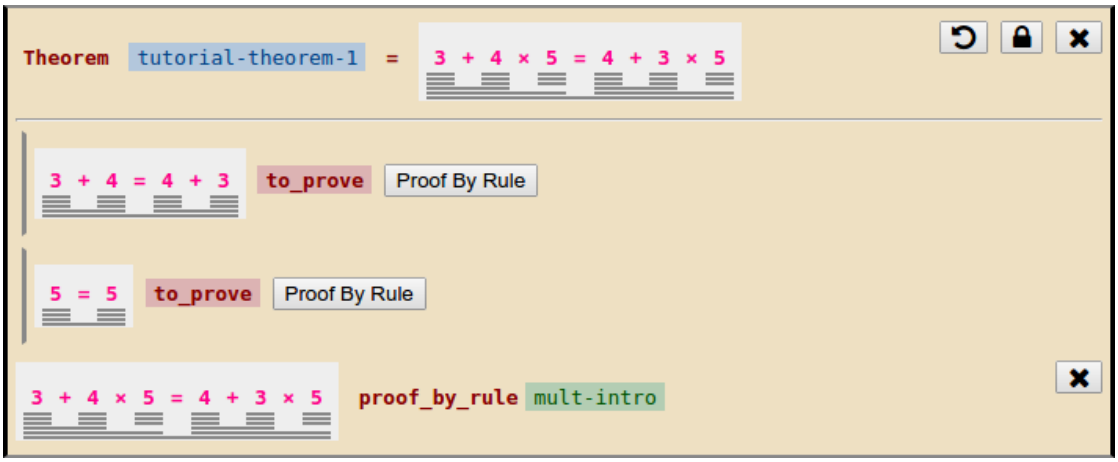


By the way, the goal here doesn't involve any meta variable. We can reset any sub-term (e.g. in this case `a`) by pressing `Alt-t` . Ultimately, you can reset the whole term by pressing `Alt-r` . In addition, you can jump to parent term by pressing `UP` , this is particularly useful when combine with `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 filling the the goal, eventually it will become like this



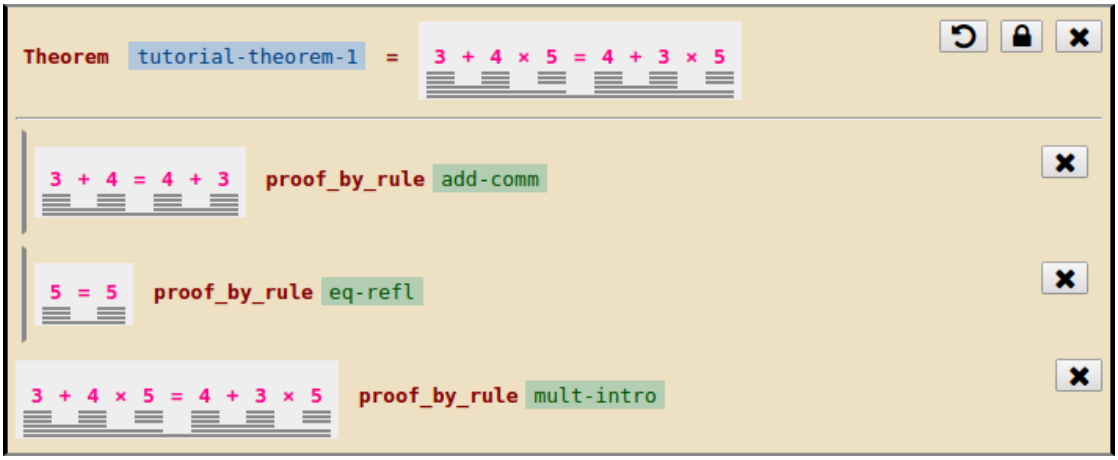
Since the goal is complete, it is ready to be proven. You can select a rule by clicking "Proof By Rule" and select `mult-intro` in a similar manner to choose grammar. Then the theorem will look like this



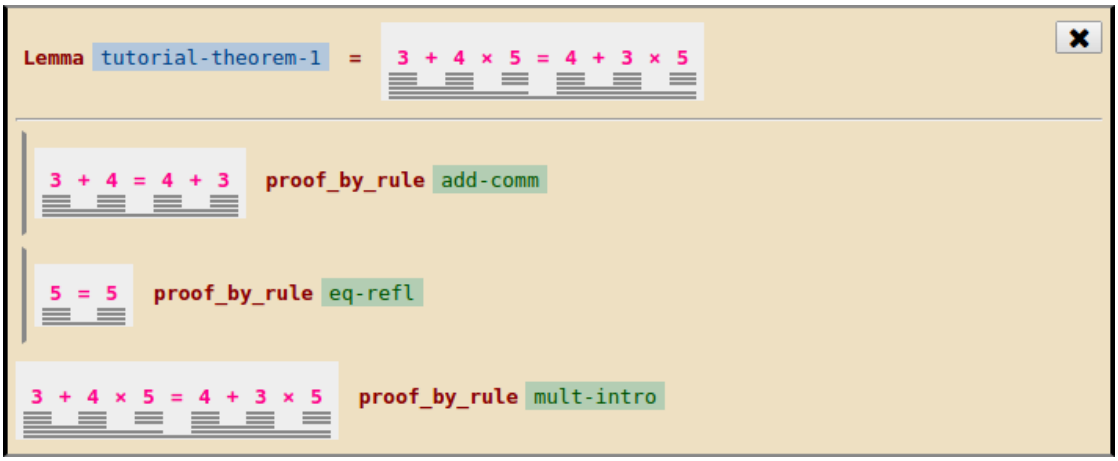
Once the rule is applied, it will generate further sub-goals.

Please notice that the goal background changes to white as we can no longer modify the goal. However if you made a mistake and want to go back, you can click ✖ 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 ↺ 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.



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



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

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

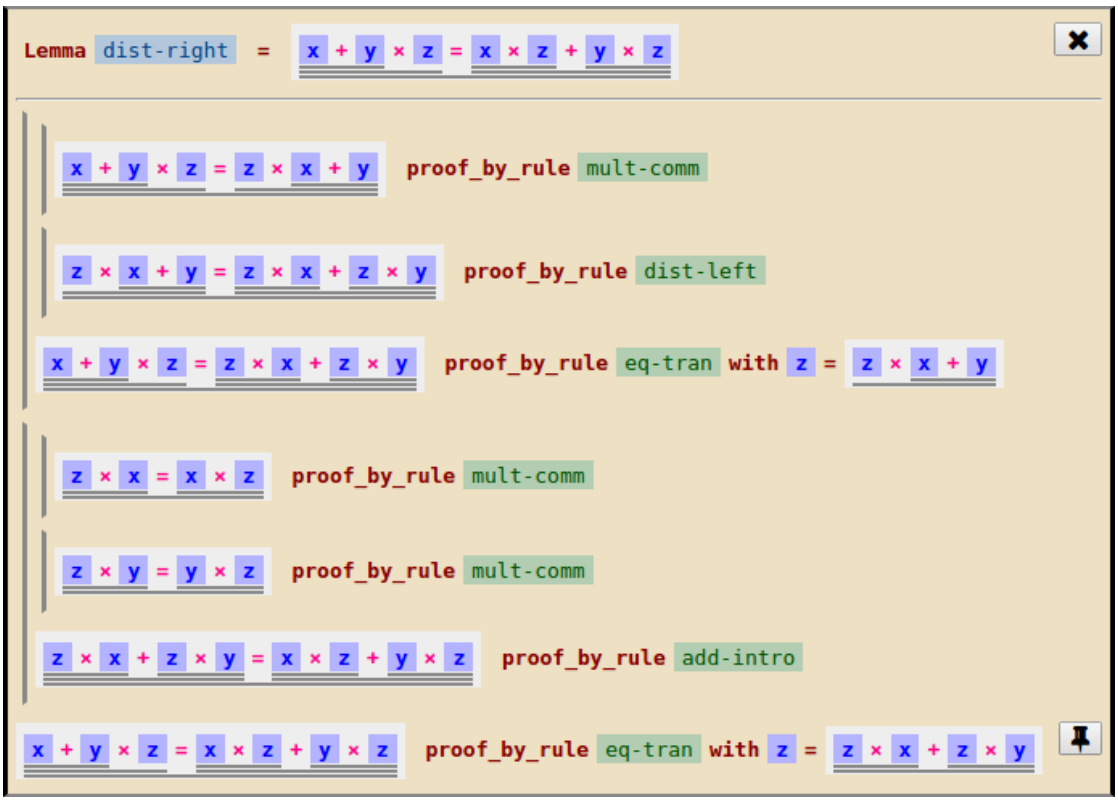


Figure 4.6: An example of lemma obtained by lock a theorem.

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 times i.e. there is no need to duplicate sub-proof.

4.5 More complex theorem

To gain more familiarity on theorem, here is more complex theorem corresponding to the second example of derivation tree on figure 3.12

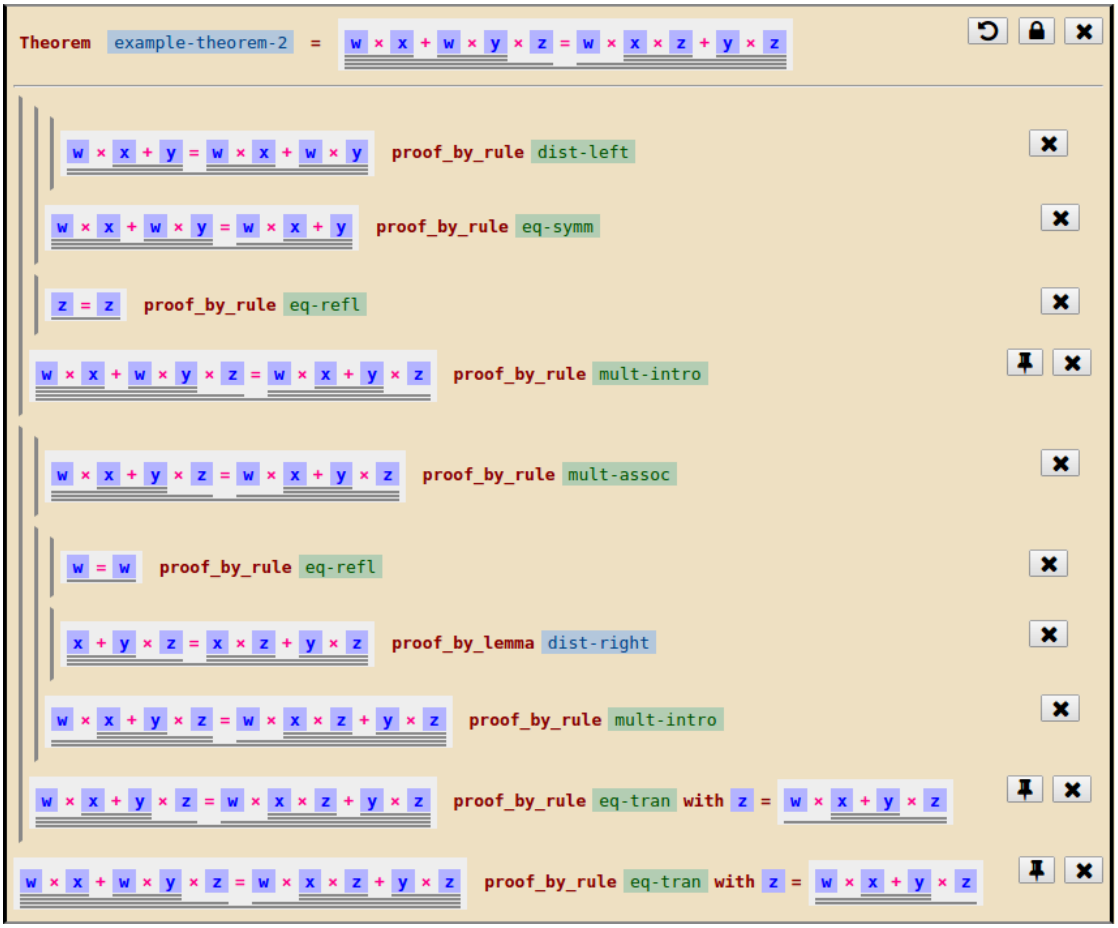
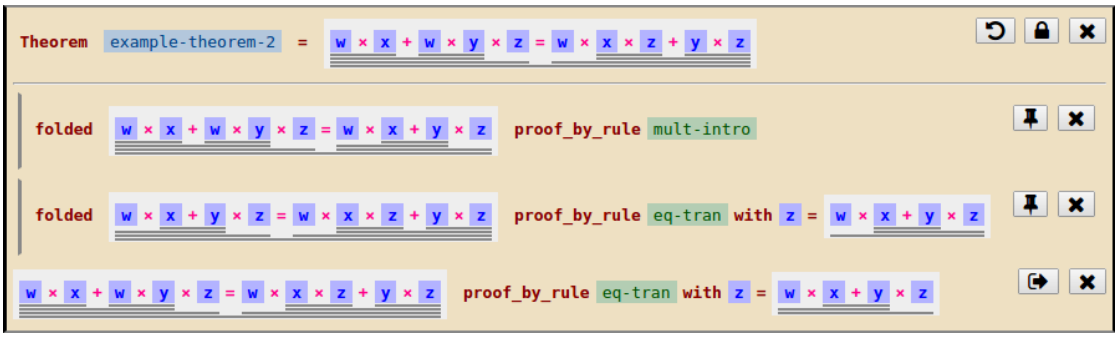


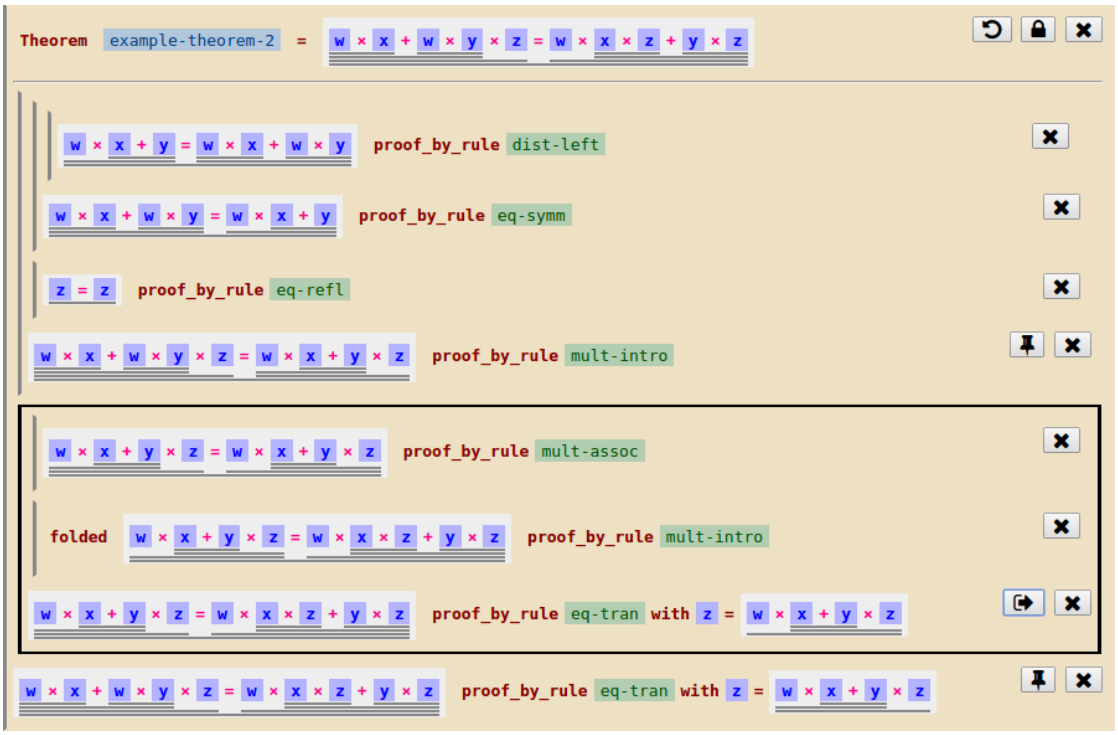
Figure 4.7: The second example theorem of Simple Arithmetic.

Sub-proofs have become more complex and premises of the 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 on the main proof it will look like this



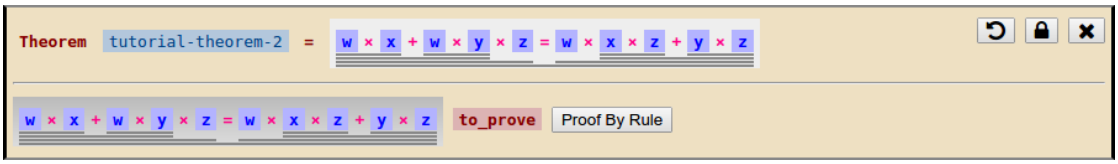
Two sub-proofs of the main theorem are folded, which allows us to read rule instance of `eq-tran` easily. You can unfold sub-proofs by clicking unfocus button ( ) at the same position that was there before. And the theorem will look the same as figure 4.7 again.

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 that correspond to that sub-proof. If you click on the first sub-proof it will look like this

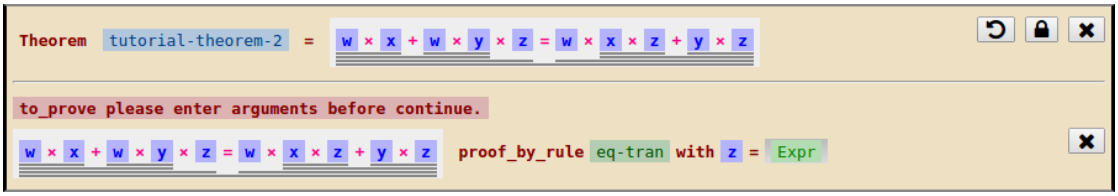


This process automatically unfold the previous one before folding sub-proof of this proof again. Please note that some of deeper proofs might not have at all, this is because its sub-proofs cannot reduce further than original one.

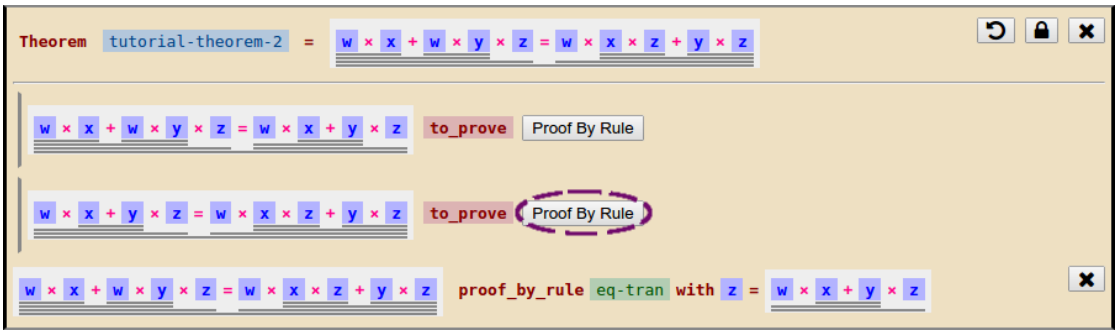
The next thing that I will show are how to use rule parameters and lemma. This can be illustrated by recreate this theorem again. First create create a theorem `tutorial-theorem-2` using the same goal as above theorem.



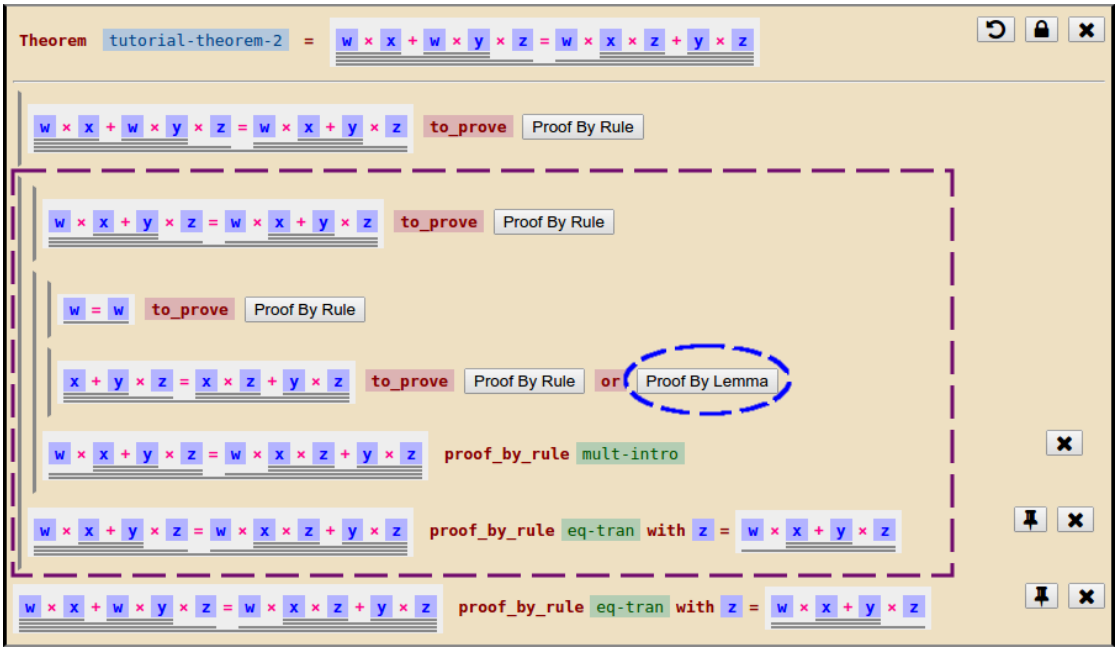
Then apply rule `eq-tran` to this goal



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



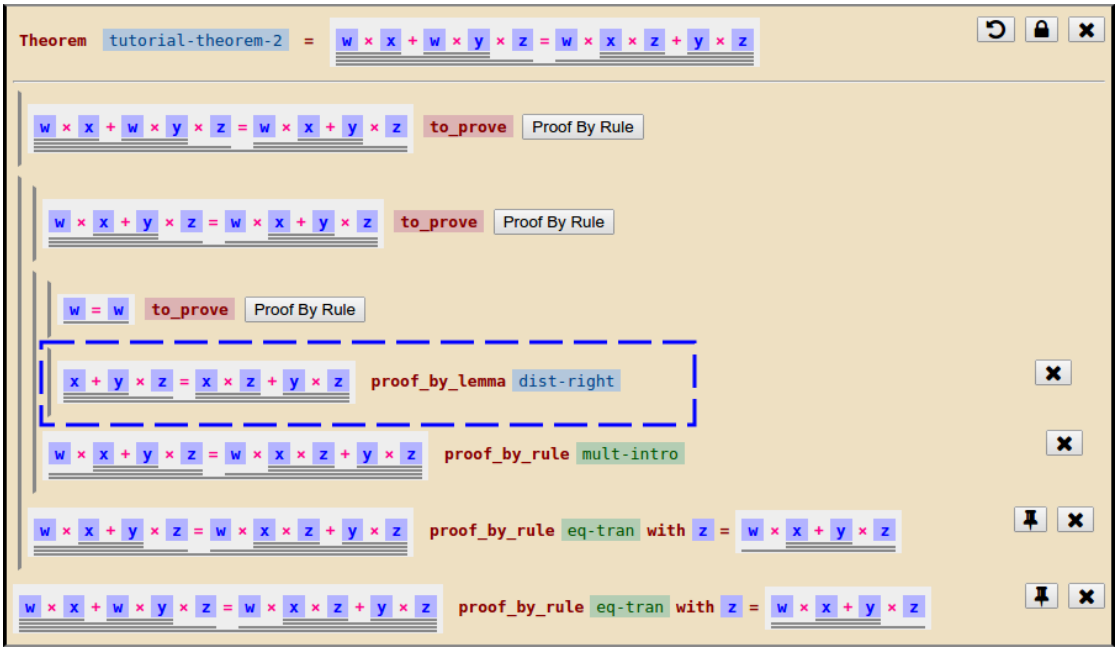
Now, let's focus on the second sub-goal, we can apply `eq-tran` again but with `z` = `w × x + y × z`. Then, apply `mult-intro` in the second its sub-goal.



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



The remaining step is easy enough.

Chapter 5

Further Examples of Formal Systems

The last chapter successfully encoded a formal system named "Simple Arithmetic", this chapter aims to show that other formal systems such as Propositional Logic and Typed Lambda Calculus can be define in the same way, therefore Phometa can encode them.

For the actual encoding and deeper explanation, please see appendix chapter A for Propositional Logic and appendix chapter B for Typed Lambda Calculus.

5.1 Propositional Logic

Propositional Logic lets us manipulate a statement that has a truth value. For example "it is raining", "today is Monday", "it is raining and today is Monday", "if today is Monday then it is not raining", and so on.

The most primitive unit is `<Atom>` where a truth statement cannot be broken down further e.g. "it is raining", "today is Monday". Formally `<Atom>` can be written in Backus-Naur Form like this

```
<Atom> ::= literal comply with regex
             /[a-z][a-zA-Z]*([1-9][0-9]*|'*)/
```

Figure 5.1: Backus-Naur Form of `<Atom>`

Basically, `<Atom>` is just a set of string that the first letter must be lower case and end with either number or "'" sign. For simplicity will use just p, q, r, ... for `<Atom>`.

Literal is used there is the same as meta variables but only be able to substitute for itself, e.g. p represent only p, not q or r or other terms of `<Atom>`.

Next we want to construct more general from of truth statement called `<Prop>` which lets `<Atom>`s combined to each other using connector such as "and ($\wedge$)", "or ($\vee$)", "not ($\neg$)", "if ... then ... ($\rightarrow$)", and "if and only if ($\leftrightarrow$)". `<Prop>` can be defined as the following

```
<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 of `<Prop>`

For example "it is raining and today is Monday" and "if today is Monday then it is not raining" can be defined as $(r \wedge m)$ and $(m \rightarrow (\neg r))$ respectively where r is "it is raining" and m "today is Monday".

`<Prop>` is similar to `<Expr>` in the way that it has meta variables e.g. $(A \vee (\neg A))$ represent any term of `<Prop>` that has "$\vee$" as the main connector and the right sub-term is the same as the left sub-term but has "$\neg$" cover again.

The difference between `<Atom>` and meta variables of `<Prop>` is that `<Atom>` represent a specific unbreakable proposition whereas meta variables of `<Prop>` represent any term of `<Prop>`. If we can proof that $(A \vee (\neg A))$ we can derive $(B \vee (\neg B))$, $((m \rightarrow (\neg r)) \vee (\neg (m \rightarrow (\neg r))))$ immediately. However if we can proof $(r \vee (\neg r))$ we might not want to share this to $(m \vee (\neg m))$ as it is irrelevant. This also explain why `<Atom>` use literal and but `<Prop>` use meta variables.

Now we could start to prove something with `<Prop>`, however, this is not so useful as the validity of `<Prop>` means "holds in every circumstance" but most of `<Prop>` only holds under curtain assumptions so we need a way to define set of assumptions (`<Context>`) and the bigger (meta) proposition that include assumptions (`<Judgement>`) which can be defined as the following

```
<Context>      ::= 'ϵ'
                 | <Context> ',' <Prop>
                 | meta-variables comply with regex
                     /[ΓΔ]([1-9][0-9]*|'*)/

<Judgement>    ::= <Context> '⊢' <Prop>
```

Figure 5.3: Backus-Naur Form of `<Context>` and `<Judgement>`

For example, "$\epsilon, p, (q \vee r) \vdash ((p \wedge q) \vee (p \wedge r))$" means "assume that p and $(q \vee r)$ hold then $((p \wedge q) \vee (p \wedge r))$ also holds".

We represent `<Context>` using a list but it is actually a set, hence ordering and duplication among propositions doesn't matter so we could define some term to be "equal by definition" as the following

- Γ, A, B is definitionally equal to Γ, B, A
- Γ, A is definitionally equal to Γ, A, A

For example, "ϵ, p, q" is definitionally equal to "ϵ, q, p, q, q, p", which in turn, makes "$\epsilon, p, q \vdash q$" become definitionally equal to "$\epsilon, q, p, q, q, p \vdash q$".

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. $\epsilon, A_1, A_2, \ldots, A_n \vdash A_i$ where $i \in 1..n$. This is achievable by the following rule

$$\textsc{hypothesis}\ \frac{}{\Gamma, A \vdash A}$$

Figure 5.4: Rule HYPOTHESIS

For example, "$\epsilon, q, p, q, q, p \vdash r$" provable using HYPOTHESIS, this because we can change the goal implicitly to "$\epsilon, p, q \vdash q$" (by definition) then apply HYPOTHESIS using Γ as ϵ, p and A as q. HYPOTHESIS doesn't have any further premise so it is finished.

In the actual encoding of Propositional Logic in Phometa, HYPOTHESIS will automatically search the usable assumption by itself using an additional feature of Phometa called *cascade premise* which can be seen in appendix section A.2.

The rest of the rules can be defined as the following

$$\text{TOP-INTRO}\ \frac{}{\Gamma \vdash \top} \qquad \text{BOTTOM-ELIM}\ \frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

$$\text{AND-INTRO}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash (A \wedge B)} \qquad \text{OR-INTRO-LEFT}\ \frac{\Gamma \vdash A}{\Gamma \vdash (A \vee B)}$$

$$\text{AND-ELIM-LEFT}\ \frac{\Gamma \vdash (A \wedge B)}{\Gamma \vdash A} \qquad \text{OR-INTRO-RIGHT}\ \frac{\Gamma \vdash B}{\Gamma \vdash (A \vee B)}$$

$$\text{AND-ELIM-RIGHT}\ \frac{\Gamma \vdash (A \wedge B)}{\Gamma \vdash B} \qquad \text{OR-ELIM}\ \frac{\Gamma \vdash (A \vee B) \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

$$\text{NOT-INTRO}\ \frac{\Gamma, A \vdash \bot}{\Gamma \vdash (\neg A)} \qquad \text{PROOF-BY-CONTRADICTION}\ \frac{\Gamma, (\neg A) \vdash \bot}{\Gamma \vdash A}$$

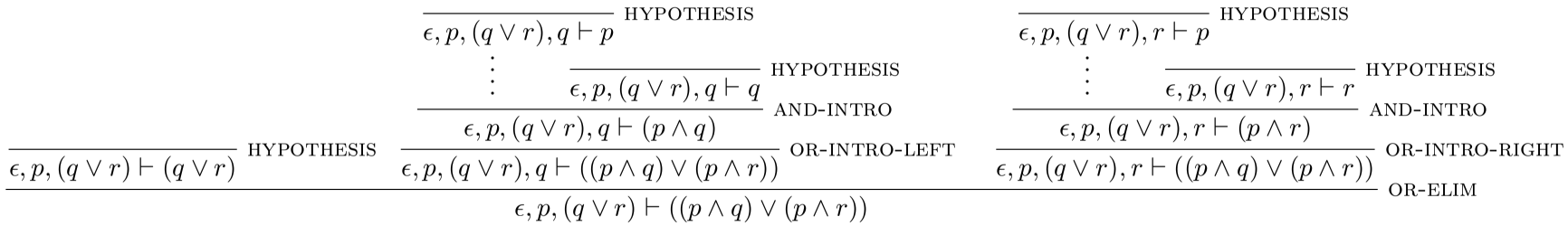
$$\text{NOT-ELIM}\ \frac{\Gamma \vdash (\neg A) \qquad \Gamma \vdash A}{\Gamma \vdash \bot} \qquad \text{IFF-INTRO}\ \frac{\Gamma, A \vdash B \qquad \Gamma, B \vdash A}{\Gamma \vdash (A \leftrightarrow B)}$$

$$\text{IMPLY-INTRO}\ \frac{\Gamma, A \vdash B}{\Gamma \vdash (A \rightarrow B)} \qquad \text{IFF-ELIM-FORWARD}\ \frac{\Gamma \vdash (A \leftrightarrow B) \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\text{IMPLY-ELIM}\ \frac{\Gamma \vdash (A \rightarrow B) \qquad \Gamma \vdash A}{\Gamma \vdash B} \qquad \text{IFF-ELIM-BACKWARD}\ \frac{\Gamma \vdash (A \leftrightarrow B) \qquad \Gamma \vdash B}{\Gamma \vdash A}$$

Figure 5.5: Derivation rules of Propositional Logic.

Now we are ready to build derivation trees,



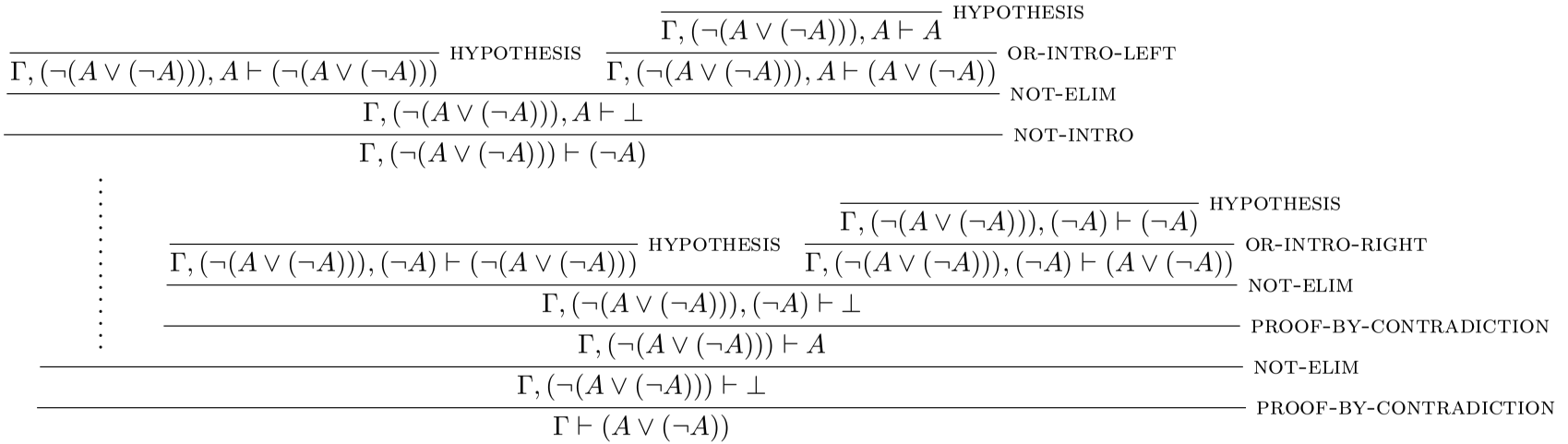


Figure 5.6: Derivation trees show that $\epsilon, p, (q \vee r) \vdash ((p \wedge q) \vee (p \wedge r))$ and $\Gamma \vdash (A \vee (\neg A))$ are valid `<Judgement>`s.

5.2 Typed Lambda Calculus

Credit: Some of material here modified from lecture note of "382 — Type Systems for Programming Languages" (third year course), Department of Computing, Imperial College London. Thank you Dr Steffen van Bakel for this.

Please note that the following formal system is just a faction of what Lambda Calculus could be. To be precise, this formal system includes β-reduction and simply types of Lambda Calculus.

5.2.1 Terms and Variables

Lambda Calculus stimulates computational model in functional manner. Basically, it will have a grammar `<Term>` which can be either a `<Variable>`, an anonymous function, an application, or a substitution. It can be described in Backus-Naur Form like this

```
<Variable> ::= meta variable comply with regex /[a-z]([1-9][0-9]*|'*)/

<Term>     ::= <Variable>
             | <Variable> '↦' <Term>
             | <Term> '·' <Term>
             | <Term> '[' <Term> '/' <Variable> ']'
             | meta variable comply with regex /[L-Z]([1-9][0-9]*|'*)/
```

Figure 5.7: Backus-Naur Form of `<Variable>` and `<Term>`

The first choice of `<Term>` represents variable which you can think as a mark point that other terms can refer to.

The second choice of `<Term>` represents an anonymous function i.e. $(x \mapsto M)$ means a function that get a variable x and return a term M where M might contain x. From now on, I will call as "blinder"

An anonymous function is usually represented by $(\lambda x.M)$ but I use $(x \mapsto M)$ since it works better with underlines when we implement it in Phometa.

The third choice of `<Term>` represents an application i.e. $(M \cdot N)$ means apply term M to term N where M usually be an anonymous function[1].

The fourth choice of `<Term>` represents a substitution i.e. $(M[N/x])$ means a term M that every occurrence of free variable x will be replaced by a term N.

A variable x is free variable if it doesn't live inside an anonymous function that has x as blinder[2], otherwise, it is bounded variable.

[1] Or will become anonymous function after evaluate the term.

[2] Blinder is a variable on the left hand side of $\mapsto$ of an anonymous function

5.2.2 Side Conditions

Sometime we want to build a simple formal system but its dependency relies on more complex formal system, for example, β reduction (will be defined later) needs to check whether a variable is a free variable of a curtain term or not, and checking free variable require knowledge of sets. Normally we should build a formal system regarding to set, then we can build β reduction, however this is overkill as formal system of set is much larger than β reduction. To solve this problem, we can build a grammar that construct a statement that need to be judge by the user as the following

```
<SideCondition> ::= <Variable> '≠' <Variable>
                  | <Variable> '=' <Variable>
                  | <Variable> '∈ fv(' <Term> ')'
                  | <Variable> '∉ fv(' <Term> ')'
                  | <Variable> '∈ bv(' <Term> ')'
                  | <Variable> '∉ bv(' <Term> ')'
                  | <Variable> '∈ fv(' <Context> ')'
                  | <Variable> '∉ fv(' <Context> ')'
```

Figure 5.8: Backus-Naur Form of `<SideCondition>`

Then write a single rule that allow to prove any side condition.

$$\textsc{side-condition}\ \frac{}{\mathit{SideCondition}}$$

Figure 5.9: Rule SIDE-CONDITION

Every time when `SideCondition` appear in a derivation tree, user needs extra care and determine whether that side condition holds or not. If it holds then apply SIDE-CONDITION and it is done. Please note that there is no mechanism to prevent user to apply SIDE-CONDITION on a false `SideCondition` and it will result in inconsistency, in another word, by using side condition technique, we *trust* user to do the right thing.

5.2.3 β Reduction

Since `<Term>` represents computational model, so it can be *evaluated*. A step of evaluation in Lambda Calculus is called β-`Reduction` which can be defined as the following

```
<β-Reduction> ::= <Term> '→β' <Term>
```

Figure 5.10: Backus-Naur Form of <β-Reduction>

$M \to_\beta N$ means M can reduce to N in one step, the rules to control this can be defined as the following

$$\text{REDUCTION-BASE}\ \frac{}{((x \mapsto M) \cdot N) \to_\beta (M[N/x])}$$

$$\text{REDUCTION-APP-LEFT}\ \frac{M \to_\beta N}{(M \cdot L) \to_\beta (N \cdot L)} \qquad \text{REDUCTION-APP-RIGHT}\ \frac{M \to_\beta N}{(L \cdot M) \to_\beta (L \cdot N)}$$

$$\text{REDUCTION-ABS}\ \frac{M \to_\beta N}{(x \mapsto M) \to_\beta (x \mapsto N)}$$

$$\text{REDUCTION-SUBST-LEFT}\ \frac{M \to_\beta N}{(M[L/x]) \to_\beta (N[L/x])}$$

$$\text{REDUCTION-SUBST-RIGHT}\ \frac{M \to_\beta N}{(L[M/x]) \to_\beta (L[N/x])}$$

$$\text{SUBST-SAME-VAR}\ \frac{}{(x[N/x]) \to_\beta N} \qquad \text{SUBST-APP}\ \frac{}{((P \cdot Q)[N/x]) \to_\beta ((P[N/x]) \cdot (Q[N/x]))}$$

$$\text{SUBST-DIFF-VAR}\ \frac{y \neq x}{(y[N/x]) \to_\beta y} \qquad \text{SUBST-ABS}\ \frac{y \notin fv(N) \qquad y \neq x}{((y \mapsto M)[N/x]) \to_\beta (y \mapsto (M[N/x]))}$$

$$\text{SUBST-ABS-RENAMING}\ \frac{y \in fv(N) \qquad y \neq x}{((y \mapsto M)[N/x]) \to_\beta ((z \mapsto (M[z/y]))[N/x])}$$

$$\text{SUBST-ABS-DISCARD}\ \frac{}{((x \mapsto M)[N/x]) \to_\beta (x \mapsto M)}$$

Figure 5.11: Rules related to <β-Reduction>

There is also β-ManyReduction which reflexive transitive closure of β-Reduction.

<β-ManyReduction> ::= <Term> '$\rightarrow_\beta^*$' <Term>

MANY-REDUCTION-BASE $\dfrac{}{M \rightarrow_\beta^* M}$ MANY-REDUCTION-NEXT $\dfrac{M \rightarrow_\beta L \qquad L \rightarrow_\beta^* N}{M \rightarrow_\beta^* N}$

Figure 5.12: Backus-Naur Form of <β-ManyReduction> and its rules.

Now we are ready to build a derivation tree,

$$\dfrac{\dfrac{\dfrac{}{(x[(y \mapsto y)/x]) \rightarrow_\beta (y \mapsto y)}\,\text{SUBST-SAME-VAR} \qquad \dfrac{}{(y \mapsto y) \rightarrow_\beta^* (y \mapsto y)}\,\text{MANY-REDUCTION-BASE}}{(x[(y \mapsto y)/x]) \rightarrow_\beta^* (y \mapsto y)}\,\text{MANY-REDUCTION-NEXT} \qquad \vdots}{((x \mapsto x) \cdot (y \mapsto y)) \rightarrow_\beta^* (y \mapsto y)}\,\text{MANY-REDUCTION-NEXT}$$

$$\dfrac{}{((x \mapsto x) \cdot (y \mapsto y)) \rightarrow_\beta (x[(y \mapsto y)/x])}\,\text{REDUCTION-BASE}$$

Figure 5.13: Derivation tree shows that $((x \mapsto x) \cdot (y \mapsto y)) \rightarrow_\beta^* (y \mapsto y)$ is valid a <β-ManyReduction>.

5.2.4 Simply Types

Some terms can be reduced infinitely many times e.g.
$((x \mapsto (x \cdot x)) \cdot (x \mapsto (x \cdot x))) \rightarrow_\beta ((x \cdot x)[(x \mapsto (x \cdot x))/x]) \rightarrow_\beta$
$((x[(x \mapsto (x \cdot x))/x]) \cdot (x[(x \mapsto (x \cdot x))/x])) \rightarrow_\beta ((x \mapsto (x \cdot x)) \cdot (x[(x \mapsto (x \cdot x))/x])) \rightarrow_\beta$
$((x \mapsto (x \cdot x)) \cdot (x \mapsto (x \cdot x))) \rightarrow_\beta \cdots$

This is not desirable as it implies a program that will never terminate. To solve this problem we can try to find a type of a term, if found, we know that the term as finite number of reductions. <Type> can be defined as the following

```
<Type> ::= <Type> '→' <Type>
         | meta variable comply with regex /[A-K]([1-9][0-9]*|'*)/
         | literal comply with regex /[α-ω]([1-9][0-9]*|'*)/
```

Figure 5.14: Backus-Naur Form of <Type>

<Type> can be instantiated as

- meta variable — for generic type which can be used in another place.
- literal — for constant type similarly to Int, Bool, Char, etc.
- $(A \to B)$ — a type of anonymous function that receive input of type A and return output of type B.

To state that a term has the particular type we can use <Statement>, and in order to prove a that <Statement> holds under certain assumptions, we need <Context> and <Judgement>. These three grammars can be defined as the following

```
<Statement>   ::= <Term> ':' <Type>

<Context>     ::= 'ϵ'
                | <Context> ',' <Prop>
                | meta-variables comply with regex
                    /[ΓΔ]([1-9][0-9]*|'*)/

<Judgement>   ::= <Context> '⊢' <Statement>
```

Figure 5.15: Backus-Naur Form of <Statemeny>, <Context>, and <Judgement>

The rules regarding to type can be defined as the following,

$$\text{ASSUMPTION}\ \frac{}{\Gamma, (M : A) \vdash (M : A)}$$

$$\text{ARROW-INTRO}\ \frac{\Gamma, (x : A) \vdash (M : B) \qquad x \notin fv(\Gamma)}{\Gamma \vdash ((x \mapsto M) : (A \to B))}$$

$$\text{ARROW-ELIM}\ \frac{\Gamma \vdash (M : (A \to B)) \qquad \Gamma \vdash (N : A)}{\Gamma \vdash ((M \cdot N) : B)}$$

Figure 5.16: Rules related to <Type>.

The ASSUMPTION is very similar to HYPOTHESIS in Propositional Logic. It also has cascade premise and context manipulation.

Now we are ready to build a derivation tree,

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon, (f : (A \to A)), (x : A) \vdash (f : (A \to A))}\ \text{ASSUMPTION} \qquad \dfrac{}{\epsilon, (f : (A \to A)), (x : A) \vdash (x : A)}\ \text{ASSUMPTION}}{\epsilon, (f : (A \to A)), (x : A) \vdash ((f \cdot x) : A)}\ \text{ARROW-ELIM}}{\vdots \qquad\qquad \dfrac{}{f \notin fv(\epsilon, (f : (A \to A)))}\ \text{SIDE-CONDITION}}{\epsilon, (f : (A \to A)) \vdash ((x \mapsto (f \cdot x)) : (A \to A))}\ \text{ARROW-INTRO} \qquad \dfrac{}{f \notin fv(\epsilon)}\ \text{SIDE-CONDITION}}{\epsilon \vdash ((f \mapsto (x \mapsto (f \cdot x))) : ((A \to A) \to (A \to A)))}\ \text{ARROW-INTRO}$$

Figure 5.17: Derivation tree shows that $\epsilon \vdash ((f \mapsto (x \mapsto (f \cdot x))) : ((A \to A) \to (A \to A)))$ is valid a `<Judgement>`.

Chapter 6

Specification

After gaining some familiarity with Phometa in the previous chapters, this chapter provides a complete reference of Phometa. After reading this chapter, you should be able to use Phometa in full efficiency.

6.1 User Interface Overview

After you install and start Phometa according to appendix section D.1, the program will look like this in a web-browser.

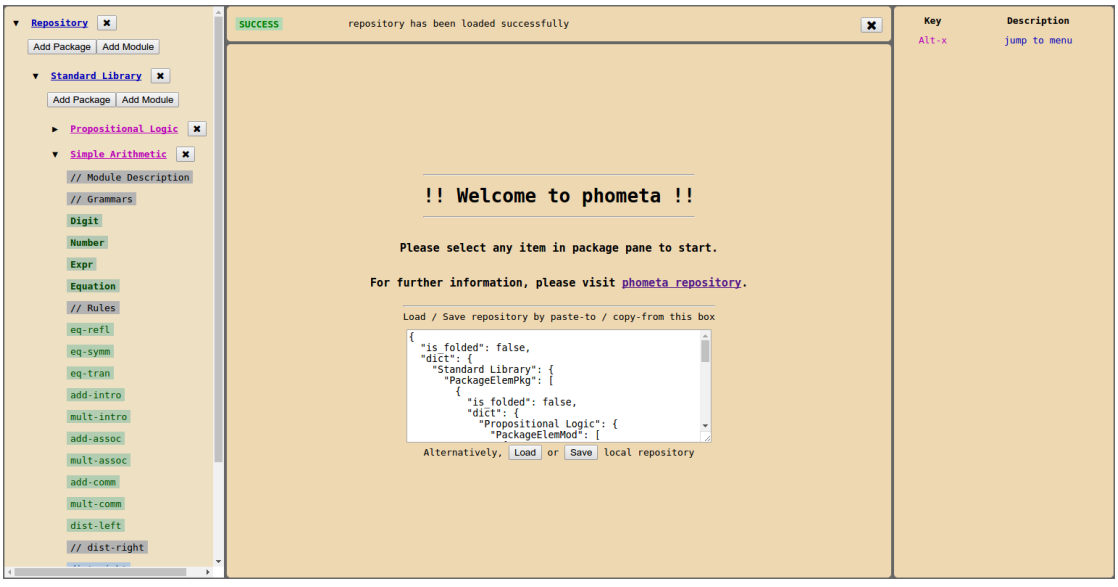


Figure 6.1: Screenshot of Phometa when you open it from web-browser.

According to figure 6.1, user interface of Phometa is divided into 4 panes as the following

- **Package Pane (left)** — shows and let users interact with the global structure of proofs repository.
- **Keymap Pane (right)** — shows every possible key-binding on the current state of the program, user can also click any row directly to get the same effect as pressing the corresponded key.
- **Grids Pane (centre)** — shows and lets user interact with the current content, grids the pane can be split into several sub-panes using mode menu explained below.
- **Messages Pane (upper-centre)** — show incoming notifications such as success or user exception notification; if there isn't any notification, this pane will disappear.

On any state of the program, user can click `Alt-x` to jump to menu, this will change the keymap pane to display the following key-bindings

- `l` — reload proofs repository from `repository.json`
- `s` — save proofs repository to `repository.json`
- `1` — reform the grids pane to display just 1 sub-pane
- `2` — reform the grids pane to display 2 sub-panes aligned horizontally.
- `3` — reform the grids pane to display 3 sub-panes aligned horizontally.
- `4` — reform the grids pane to display 4 sub-panes as 2×2 grids.
- `8` — reform the grids pane to display 2 sub-panes aligned vertically.
- `9` — reform the grids pane to display 3 sub-panes aligned vertically.
- `h` — make focused sub-pane of the grids pane display home page
- `p` — toggle (show / hide) the package pane
- `k` — toggle (show / hide) the keymap pane

For example, in any state of the program, we can split the grids pane into two horizontally sub-panes by pressing `Alt-x` followed by `2` to get a result similar to figure 6.2.

In fact, splitting grids is recommended when you construct a proof, because it will let you use one of the sub-panes to look up relevant grammars and rules without losing concentration on a theorem that is on another sub-pane.

Figure 6.2: Screenshot of Phometa when you split the grids pane into 1×2 grids.

6.2 Auto-Complete Input Box

Although users will interact with Phometa mainly by clicking a button and pressing a keyboard shortcut, there are several places that might require the user to type something, and in order make this as convenient as possible, every input box except textarea in comment node will have *auto-complete* functionality.

An (auto-complete) input box will internally have a list of choices, which will be mapped to keys `Alt-1..9` respectively. Because these choices are keyboard shortcut hence they will be visible in the keymap pane so user can see all of this. If there are more than 9 choices, it will map just the first 9 choices to `Alt-{1..9}` and provide `Alt-[` and `Alt-]` that can jump to previous and next page of choices respectively.

Key	Description
Alt-1	eq-refl
Alt-2	eq-symm
Alt-3	eq-tran
Alt-4	add-intro
Alt-5	mult-intro
Alt-6	add-assoc
Alt-7	mult-assoc
Alt-8	add-comm
Alt-9	mult-comm
Alt-u	search unicode
Alt-x	jump to menu
Alt-[	prev choices
Alt-]	next choices

Key	Description
Alt-1	dist-left
Alt-2	add-identity
Alt-u	search unicode
Alt-x	jump to menu
Alt-[	prev choices
Alt-]	next choices

Figure 6.3: Key-blinding of an example of auto-complete before and after pressing `Alt-]`

Another way to access further choices is to type some sub-strings (separated by space) in the input box. This will filter only choices that still match with those sub-string. For example, according to the choices in figure 6.3, if we type ‘‘dd’’, it will match just 4 rules which are add-intro , add-assoc , add-comm , add-identity since this has ‘‘dd’’ as sub-string. Now, if we type ‘‘dd in’’ it will match just add-intro since it is the only one that has ‘‘dd’’ *and* ‘‘in’’ as sub-string. In addition, if there is just one choice left, we can hit “Return” for that choice alternative to Alt-1 for convenience.

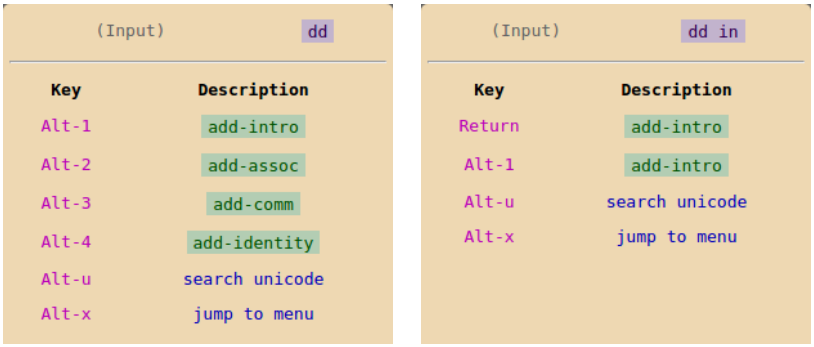


Figure 6.4: Keymap pane of figure 6.3 after be filtered by ‘‘dd’’ and ‘‘dd in’’.

When typing something into the input box, it also appears on the top of the keymap pane as well (example as in figure 6.4). Although it is not related to keyboard shortcut but that position is closer to the keymap table, hence user can type the filter then look at the remaining choices more easily.

Some input box can accept newly created string as well e.g. input box waiting for the name of a new node. If this is the case, you can write the string directly into the input box then hit Return to complete it[1]. Please note that the written string still be used to filter choices out (if exists), but Return as alternative to Alt-1 will no longer work as it contradicts with this new key.

Auto-complete is also capable for unicode input method as well, if you want to add a unicode character, just press Alt-u then the keymap pane will look like figure 6.5.

In unicode input method, we have choices that map to each unicode character using its name in LaTeX math mode, these choices can be manipulated by Alt-[and Alt-] and filtering method similar to normal choices above. Once a choice is selected, it will be appended to the end in input box and return to normal input method. However, if you enter unicode input method by accident you can press Escape to return to normal input method with appending any unicode character.

[1]If you click somewhere else before hitting Return , the string in the input box will be gone without producing any effect. This common pitfall for newcomers and will be explain further in chapter 8.

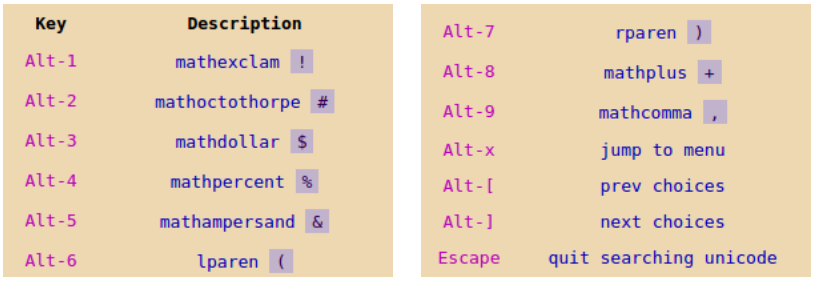


Figure 6.5: Key-blinding after entering unicode input method.

To conclude this section, originally, auto-complete was designed for an input box that is used to select a choice e.g. selecting a grammar for a term or selecting a rule for a step of a theorem, but now it is extended to other types of input box that doesn't have choice e.g. input box the name of a new node. So all of input boxes will have some feature in common, e.g. ability to use unicode input mode, this reduces Phometa learning curve.

6.3 The proofs repository

Phometa has a repository which consists of packages and modules. Analogously, you can imagine packages as directories and modules as files, therefore the proofs repository is similar to a real repository that contains directories and files.

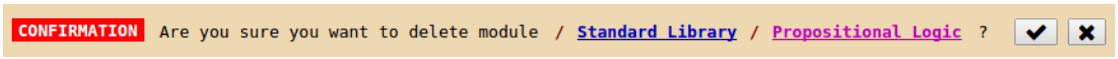
The package pane represents the global structure as shown in the following example,

Figure 6.6: Package pane represents the global structure of proofs repository

The current repository has one package named “Standard Library” which consists of three modules named “Propositional Logic”, “Simple Arithmetic”, and “Typed Lambda Calculus”. Each module of Phometa consists of a sequence of nodes, each node can be either comment (grey), grammar (dark green), rule (light green), or theorem (blue).

Please note that a package contains its elements alphabetically whereas a modules contains its nodes using an explicit ordering that user can swap among them later.

Each package and module can be folded or unfolded by clicking a triangle in front of it. You can also delete them by clicking ✖ on the right hand side, then the conformation notification will appear in the messages pane like this



You can add sub-packages or sub-modules in any package by clicking “Add Package” or “Add Module” respectively, the clicked button will transform to an input box that you can enter the name.

To read or edit a module, you can click at the label of the module directly, this will make the focused sub-pane[2] of the grids pane to display the module similar to figure 6.7.

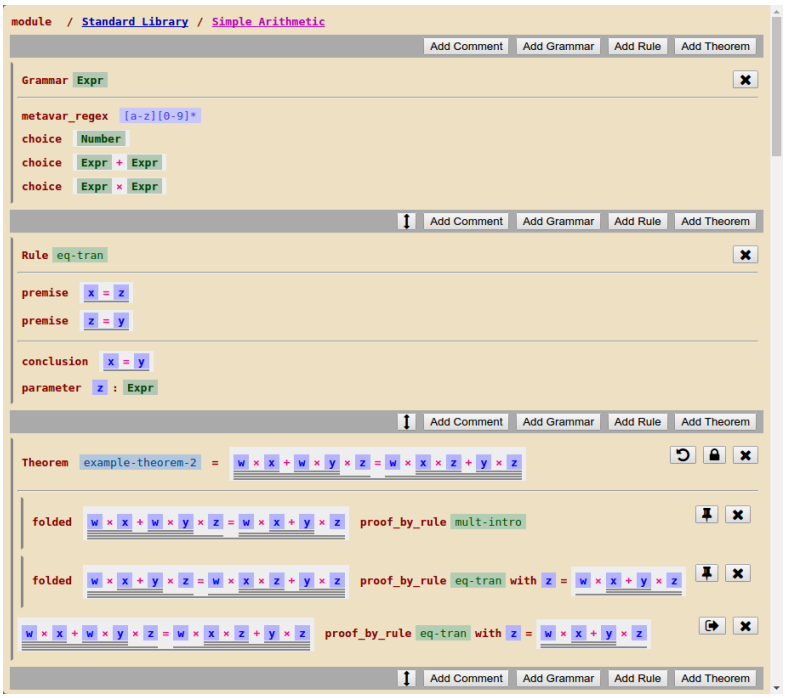


Figure 6.7: Screenshot of Simple Arithmetic module displayed in the grids pane after the user click the label “Simple Arithmetic” in the package pane. Please note that nodes in this module are reordered so you can see different type of nodes in one screenshot.

[2]If there isn’t a focus one, use the first sub-pane.

The displayed module contains nodes that you can interact[3] with directly. If you feel like this is too crowded, you can concentrate on particular node by clicking its label in the package pane, this will show only that node on the grids pane not the entire module.

Nodes in a module are interspersed by panels which allow you add a new node by clicking the button corresponded to the type of a new node, the button then change to an (auto-complete) input box that ask for the name of that node, after you type the name and hit `Return` , the new node will appear at the position of that panel. A panel also has ↕ that allow you to swap the location between the above and below nodes.

The current proofs repository can be loaded or saved with `repository.json` by using `Alt-x l` and `Alt-x s` as we mention before. This allows user to manipulate the proofs repository directly. For example,

- User can backup the current state of proofs repository by copy `repository.json` to somewhere else. If user accidentally delete some node or something else gone wrong, user can just replace it with the backup one.
- Users can work in a group on the same repository by passing `repository.json` around.
- Users can distribute their formal systems to others by publishing their `repository.json`

However, manually modifying `repository.json` without using Phometa interface is strongly discouraged as it will render the repository into an inconsistent state.

Alternative to `repository.json`, user can copy / paste the repository directly from the text-area in the home page.

6.4 Node Comment

Node comment allows users to add commentary anywhere in a module. It is a proper node i.e. we need to specify the name of the comment, by convention, we will add "\\" as the prefix to the name so we don't have to worry about name clash with another type of nodes.

To construct this, we need to obtain a blank node using one of adding panels explained in the last section. Then you can click "Edit Comment" which is on the top-right corner of the node then a HTML textarea[4] will appear so you can write your comment, once it is finished, you can click "Quit Editing" button (or simply click somewhere else), this will transform the HTML textarea to normal box. Unlike other nodes, there is no need to lock a comment node so you can come back and edit this anytime.

[3] Details on how to interact with each type of nodes will be describe in later sections.
[4] This is the only input box that doesn't have auto-complete functionality.

6.5 Node Grammar

Node grammar allows users to encode a Backus-Naur Form. To construct this, we need get a draft grammar using one of adding panels, this will let us define the following properties of the grammar

- `metavar_regex` — Specify JavaScript regex pattern that control the name of a meta variable of this grammar. This property is optional, and if it is absent, meta variables of this grammar cannot be instantiated.
- `literal_regex` — Specify JavaScript regex pattern that control the name of a literal of this grammar. This property is optional, and if it is absent, literal of this grammar cannot be instantiated.
- `choice` — Describe an alternative branch that a term can be instantiated to. A choice consists of sub-grammars and format-strings stripped to each other.

  This property is multiple, you can create a new choice by clicking "Add Choice" button on the top-right corner of the grammar, it will ask you for the number of sub-grammars that the new choice will have, after that, a new `choice` will appear with $2n + 1$ buttons, the odd buttons will asking for string of syntax, and even buttons will be asking for sub-grammar (draft grammar and itself included) which you can use auto-complete method described in auto-complete section to complete.

  You can also swap order among choices by clicking ↕ on the right hand side of that choice, this will swap the current choice with the one lower (wrap around if necessary).

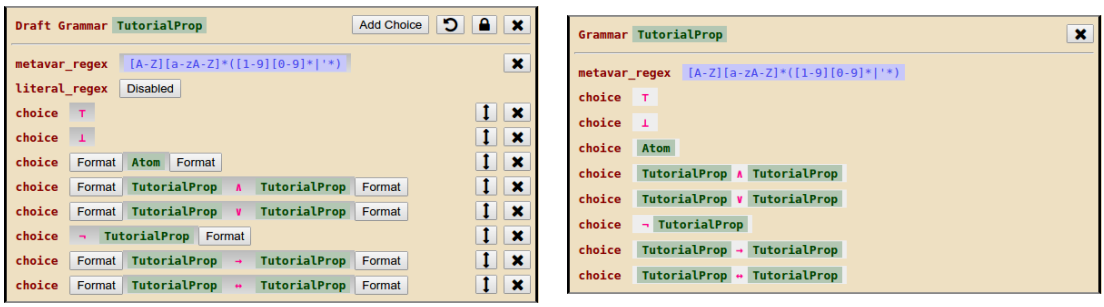


Figure 6.8: Example of a grammar before and after it is locked.

Once the grammar is completed i.e. all of buttons asking for sub-grammars are fulfilled, we can lock it using 🔒 on the top-right corner of the grammar so it will become a proper grammar and ready for usage of other type of nodes. In addition, if something is wrong during constructing this, you can reset the grammar to the initial state by clicking ↺.

For more information about grammar construction, see appendix section A.7.

6.6 Root Term

Root terms are not sub-terms of any other terms (as you can see from user interface). There 3 types of terms depending in its editability as the following

- Editable-up-to-grammar — Both of grammar and term content of a root term can be changed. They are used to construct premises and the conclusion of rules and the goal of theorems.
- Editable-up-to-term — Term content of a root term it editable but the grammar is given and fixed. It is used in the place that depend on other node such as an argument of a rule.
- Read-only — Everything is fixed, the root term is shown there only for your information.

To make this type more distinguishable, read-only root-term will have white background colour represented its immutability, in constant, other two will have grey background colour.

When an editable-up-to-grammar root-term is newly created, it doesn't know its own grammar and it will be shown as a button with a label "Choose Grammar". To specify a grammar, click that button, it will transform to an auto-complete input box waiting you to select a grammar using the method described in auto-complete section. Once you select a grammar for this, it will transform to another auto-complete input box but now asking for term content which behaves the same as an editable-up-to-term root-term.

At the beginning of term-content input-method, i.e. when editable-up-to-grammar root-term get its grammar or editable-up-to-term root-term is newly created, the root term will show a single auto-complete input box with a placeholder as the name of corresponding grammar and have green background colour. This auto-complete input box, from now on, will be called *todo-hole* and it can be fulfilled one of following method

- Select one of `choice` s stated in the corresponding grammar declaration using auto-complete method, this will replace current todo-hole with that choice with corresponding sub-grammars as further todo-holes (if any).
- Write a string to the input box and hit `Return`
  - if the string match with `metavar_regex` , then the current todo-holes will be replaced with meta variable
  - otherwise, if the string match with `literal_regex` , then the current todo-holes will be replaced with literal
  - otherwise, do nothing and prompt an exception on the messages pane.
- Select on of existing meta variables or literals (if any) using auto-complete method.

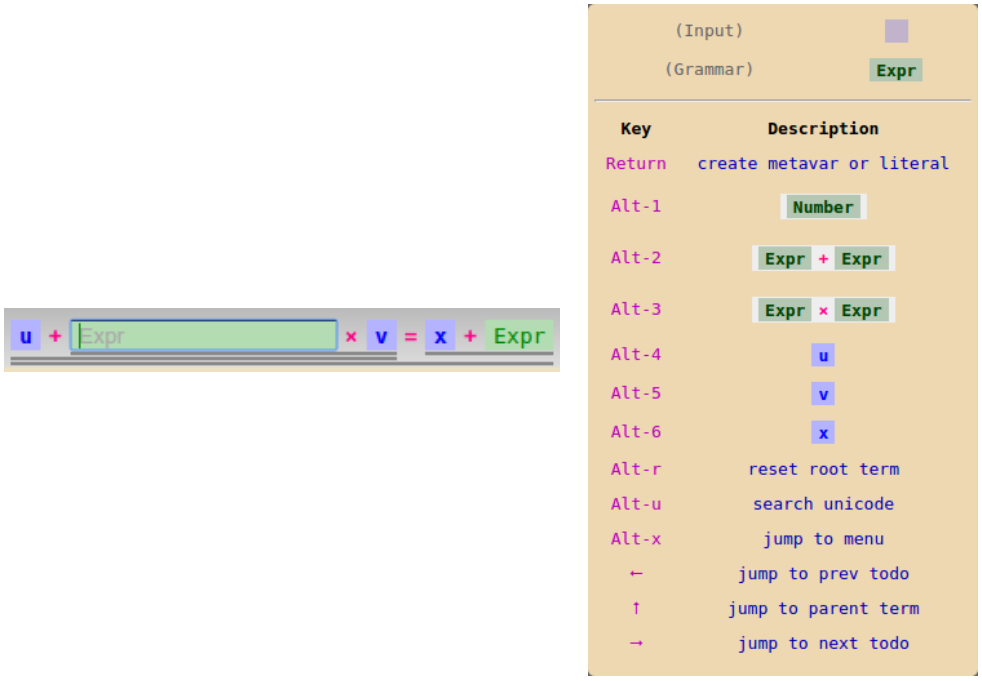


Figure 6.9: Example of root term during term-content input-method (left) and the corresponding key-blinding (right). You can see that the current todo-hole has `Expr` as its grammar so their are 3 choices from `Expr` and other 3 choices which are existing meta variables that has been entered bofore, nevertheless it can write a new string then hit `Return` for meta variable as well.

After the current todo-hole as been fulfilled, the cursor will automatically focus on the todo-hole, so user can finish entering the whole term content without touching a mouse.

Phometa also has another feature to automatically expand the term that has exactly one choice and can't instantiate neither of meta variable nor literal e.g. a term of `Equation`

Every (sub) term can be focused directly by clicking at it, then the focused term will be overlined. If the focused term is todo-hole, keymap show its input text and grammar (similar to the right screenshot in figure 6.9), otherwise, the keymap will show its grammar and the term (similar to the right screenshot in figure). In addition, if the focus term is `choice` term, we can jump to its sub-terms using `{0..9}` as showned in figure 6.10.

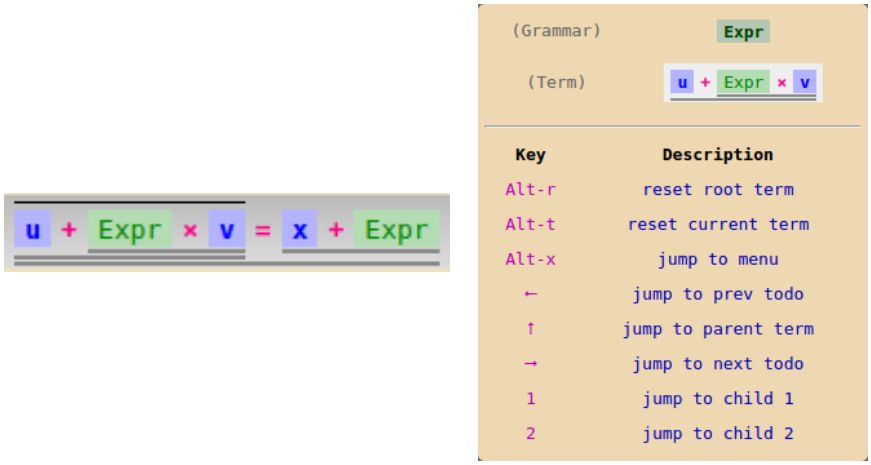


Figure 6.10: Example of root term when `choice` term is clicked, we can jump to its first term by clicking `1` or second term by clicking `2`.

Root term has limited space to interact, therefore, it make use lots of shortcut which can be described[5] as the following

- `r` — reset the whole root term to its initial state (mutable term only).
- `t` — reset the current term to todo-hole (mutable and non todo-hole term only).
- `Up` — if it is root term, quit root term and focus on outer nod
- `Up` — if it is sub term, jump to parent term of current term.
- `Left` — jump to next todo-hole.
- `Right` — jump to next todo-hole.

For further information about building a root term, please see the first half of section 4.4.

6.7 Node Rule

Node rule allows users to encode a derivation rule. To construct this, we need get a draft rule using one of adding panels, this will let us define the following properties of the rule

- `allow_reduction` — Specify whether this rule could be used for meta reduction or not. This property is a flag i.e. you can click "Disabled"/"Enabled" button to toggle it.

[5]This list excludes keyboard shortcut of auto-complete as it is not specific to root term

- `conclusion` — Specify a root term that will be used as the conclusion of the rule. The root term displayed here is editable-up-to-grammar where you can fulfil it using method described in the last section.
- `premise` — Describe a root term that will be used as one of premises of this rule. Root term input method is the same as `conclusion` .

  This property is multiple, you can create a new precise by clicking "Add Premise" button on the top-right corner of the rule.
- `cascade` — Describe a cascade premise which similar to a normal premise but when the rule is called, rather than doing normal pattern substitution, it tries to call its first sub-rule, if success this cascade premise will be replaced by zero-or-more premises of sub-rule, otherwise, cascadely calling other sub-rules. For more information about a cascade premise, please see appendix section A.2.

  This property is multiple, you can create a new cascade premise by clicking "Add Cascade" button on the top-right corner of the rule.

  Each cascade premise has a list of sub-rules that it can try cascade. To add a new sub-rule click "Add Sub-Rule" button which will transform to an auto-complete input box asking for a rule (draft rules and itself included), then the sub-rule calling template consists of the following

  - Sub-rule name — The rule that will be called as sub-rule, this is specified when we add this sub-rule calling template and now it is fixed.
  - Unifiable / Exact Match flag — The flag that you can toggle between "Unifiable" (which is the default) and "Exact Match" (treat sub-rule that require further substitution as failure).
  - Pattern of sub-rule goal — An editable-up-to-term root-term that is formed to match against sub-rule conclusion.
  - Patterns of sub-rule arguments — Editable-up-to-term root-terms that are formed to match against sub-rule parameters.

  You can use ↕ to reordering sub-rules around in the same way that you swap ordering of `choice` s of a grammar.
- `parameter` s — Show meta variables (literals excluded) that appear in one of premises but not in the conclusion. This property is automatic i.e. it will be updated every time when one of premises or conclusion is updated, user cannot manually change it.

List of `premise` s and list of `cascade` s are, in fact, the same list, you can use ↕ to swap ordering of these `premise` s and `cascade` s around in the same way that you swap ordering of `choice` s of a grammar.

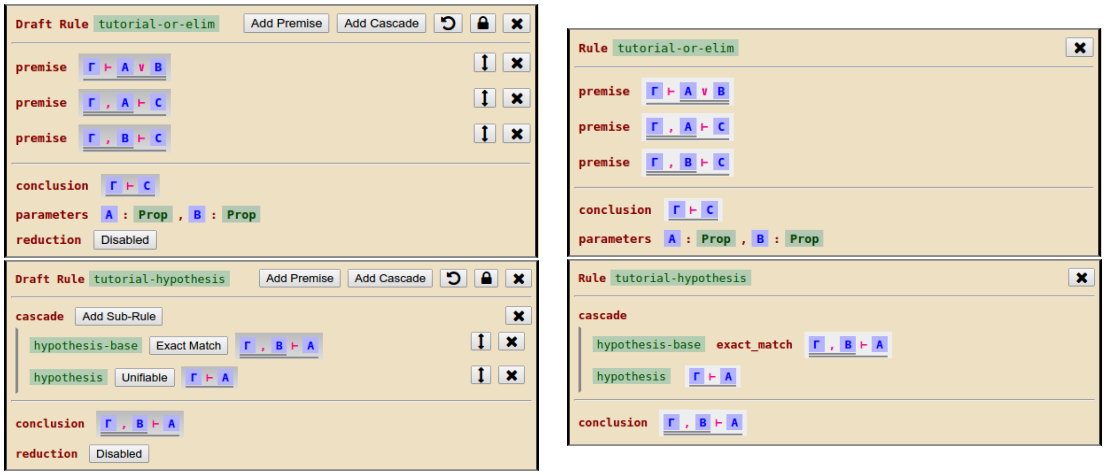


Figure 6.11: Example of rules before and after they are locked.

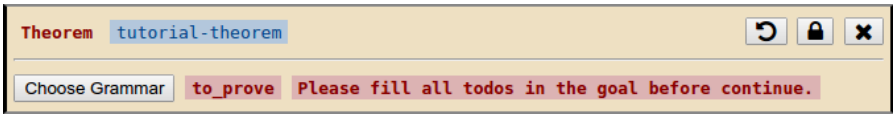
Once the rule is completed i.e. all of root terms has no todo-hole left, we can lock it using 🔒 on the top-right corner of the rule so it will become a proper rule and ready for usage of other type of nodes. In addition, if something is wrong during constructing this, you can reset the rule to the initial state by clicking ↺.

For more information about rule construction, see appendix section A.7.

6.8 Node Theorem

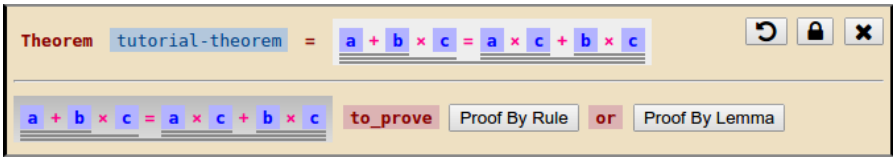
6.8.1 Theorem Construction

Node theorem allows users to encode a derivation tree. To construct this, we need get a draft theorem using one of adding panels, this will give us a blank theorem with the goal as editable-up-to-grammar root-term.



Once the we complete the goal, it will also appear in the header[6] and there might be two buttons pop-up as the following

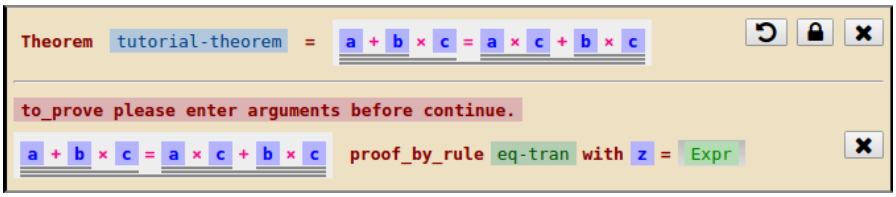
[6]This is useful for a long theorem as reader doesn't need to scroll down the very end just to see what has been proven.

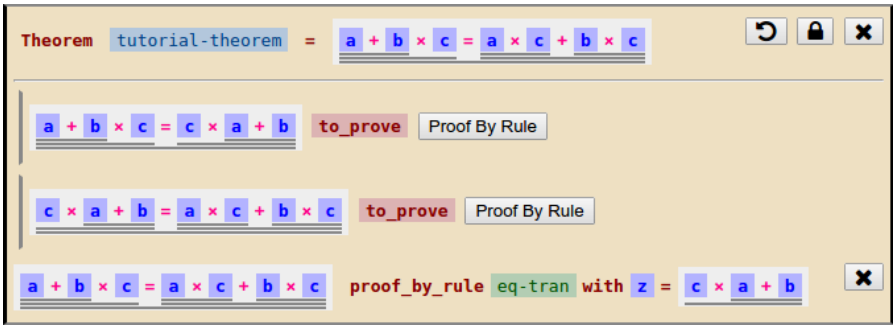


- "Proof By Rule" button — once clicked, it will turn to an auto-complete input box with choices consisting of rules which has the grammar of the conclusion the same as the grammar of the current goal. If there is no such a rule, "Proof By Rule" button will not appear at the first place. Please note that some rule on auto-complete choices might not be applicable to the current goal, if we select it, it will nothing and pop-up exception in the message pane.
- "Proof By Lemma" button — once clicked, it will turn to an auto-complete input box with choices consisting of lemmas which each of them has the goal that can turn to out current goal i.e. each meta variable of the lemma goal can be substituted in such a way that is it identical to out current goal. If there is no such a lemma, "Proof By Rule" button will not appear at the first place. In fact, if there is at least one lemma in the auto-complete choices, we can guarantee that our current goal can be proven as it can fail in the similar way as rules.

In the current example, we could "Proof By Lemma" and select `dist-left` then this theorem will finish immediately, but in order to investigate more on other thing, we will select "Proof By Lemma" and select `eq-tran` .

Normally, when we apply a rule, the conclusion of the rule will be pattern matched against the current goal, then substitute pattern match result to each premise to get sub-goals that are needed to be proven later. However if the rule has parameters, it will ask user to fill arguments that corresponding to each parameter using an editable-up-to-term root-term, similar to this example.





After the main goal knows its rule or lemma, the colour of it become white i.e. now it is read-only root-term. Sub-goals (if any) will have white background as well since they depend on a rule and it can't be changed.

The generated sub-goals can be proven using the same method as the main goal, and it will happened recursively until all leaf sub-goals are proven by rules with no premises or by lemma.

For each (sub) goal, that has been proven, we can reset its proof by clicking [✖] on the bottom-right corner of particular proof. If we reset the main proof, the theorem goal will turn grey again and allow we to modify it.

Once, the theorem is complete i.e. there is no active sub-goal left, we can lock a theorem to become a lemma by clicking [🔒] on the top-right corner of the theorem. In addition, if something is wrong during constructing this, you can reset the entire theorem to the initial state by clicking [↺].

Since user will interact with Phometa most the time by constructing a theorem, so we provide some keyboard short-cut that user can use.

- `r` — reset the whole theorem (equivalent to [↺]).
- `t` — reset the current current proof (equivalent to [✖] of the current proof).
- `l` — lock the theorem (equivalent to [🔒]).

For further information about building a theorem, please see the second half of section 4.4.

6.8.2 Theorem Fold/Unfold

Some (sub) proof in a theorem is quite long so its goal and sub-goals will be far away from each other, therefore, it is very hard to read them together as an instance of a derivation rule. To tackle this problem, we can click [📌] on the proof that we want to focus, this will fold its sub-proofs so goal and sub-goals will stick together and easier to read as instance of a derivation rule.

To unfocus the proof, we can click [⮕] at the same position that we click [📌]. This will unfold all of sub-proofs that are currently folded.

To avoid potential confusions among (sub) proofs, we allow to have a focus proof one at the time (per theorem) i.e. if you click [📌] on some proof during another proof is being focus, then it will automatically unfocus the old one (without clicking [⮕]) and then focus the new one.

Some proof might not have any sub-goal that is foldable, focusing on it will not produce any difference, hence, such a proof will simply doesn't have [📌] at the first place.

For further information about theorem fold/unfold, please see section 4.5.

6.9 Meta Reduction

Some rule has exactly one premise which has the same grammar as the conclusion. Conceptually, this establishes some kind of relation that suggests that the conclusion of this rule should be able to *reduce* to the premise. We can make this concept come true by enabling flag `allow_reduction` on that rule.

Meta reduction can be performed in a (sub)term that is inside a (sub)goal of a theorem, and that (sub)goal must be in "todo" state i.e. it must not be in parameters-waiting state nor has been proven by a rule or a lemma. Although sub-goal is read-only root-term but meta reduction can be preform since it is not related to the term construction.

Once you click such a (sub)term, you might see some rule appear as an auto-complete choice, this means that you can use that rule to reduce the term.

Formally, a rule can appear as an auto-complete choice to reduce a term if it satisfies the following condition

- The `allow_reduction` flag of the rule is set.
- The rule requires no parameters.
- The generates exactly one premise (might happen dynamically via cascade premises).
- That premise has the same grammar as the conclusion of the rule.
- The applying process doesn't involve any further substitution.

For further information about meta-reduction, please see appendix section A.6.

Chapter 7

Implementation

This chapter aims to illustrate the high-level implementation of Phometa. So that the reader will get a rough idea on how Phometa works.

7.1 Decision on programming language

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

One of the most attractive features 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 a Boolean that holds "True" when the mouse is pressed, `Window.dimensions : Signal (Int, Int)` represents a pair of integers that hold current width and height of the window respectively. Custom Signal can be created using higher order functions as in 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 because when the dimensions of the window is changed, it will notify `Window.dimensions` to update its value, `Window.dimensions` in turn, notifies `window_area` to update its 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 a 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 compiled, we will get a HTML file (with JavaScript embedded) such that if it is opened in a 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, which will change back when you release the mouse.

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

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

7.2 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 applications we need a way to store variables, which 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 clicks mouse on the page, `Mouse.isDown` will notify `click_total` to invoke `fold_func`, and the value of this Signal will be updated to 1. If you click it again it will be 2, 3, 4, etc.

In Phometa, an accumulator of `Signal.foldp` is `Model` represents the global state of the entire program. The third argument (input Signal) of `Signal.foldp` is `Action` representing the Signal that changes every time when user presses the keyboard or clicks 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 7.1: 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 turn, can be transformed using `Signal.foldp` together with `view : Model -> Html` to achieve `main` which represents 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.

7.3 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 7.2: 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 7.3: 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 the detect change in `action_signal` so it applies function `update` with "`ActionCommand cmd`" to `model_signal`. The function `update`, in turn, will apply command `cmd` to current model and this is how the model can be updated.

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

7.4 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's define some common data-types[3] which will be used by later data-types.

```
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 7.4: Representation of common data-types that will be used later on

Now, we can define a proof repository which is just a big `Package` stored in the 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 7.5: Representation of `Package` and related components.

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

We could think of a `Package` roughly as a collection containing other `Package`s and `Module`s. We also need `PackagePath` to reference a package relative to root package.

`Module` is a collection containing nodes. `ModulePath` is a unique reference from root package to this module. Please note that `Module` uses `OrderedDict` in contrast to `Package` which uses `OrderedDict`; this is because `Module` needs to remember order of nodes but `Package` simply shows elements alphabetically.

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

Figure 7.6: Representation of `Module` and related components.

There are four kinds of `Node` which are `Comment`, `Grammar`, `Rule`, and `Theorem`[4]; each of them will have corresponding data-type which will be defined later. `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

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

Figure 7.7: Representation of `Node` and related components.

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

We could define `Grammar` as the following figure which is coincided to *Node Grammar* described in section 6.5.

```
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 7.8: 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 the former is longer than the latter 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 7.9: Representation of `Term` and related components.

We usually use `RootTerm` as a term at the top level, this is because it know its own grammar. In fact, `Term` is just an auxiliary model of `RootTerm`. Please note that it is possible to find 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 figures which are coincided to *Node Rule* and *Node Theorem* described in sections 6.7 and 6.8, respectively.

```
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 7.10: 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 7.11: 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 7.12: 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.

7.5 Pattern Matching and Unification

Although it is impossible to show all of codes written in Phometa, there is a faction of code related to *Pattern Matching* that is small enough to be described entirely; it is also one of the most interesting part as well.

Credit: some material in this section inspired from [4].

Please note that this section only described how pattern matching works. For the supplementary source code, please see appendix chapter E.

Let start with the most important function, `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 receive a list of pair of `(pattern, target)` rather than just one `pattern` and one `target`. These two are shown in figure E.1.

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 is shown in figure E.2 and 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, succeed immediately with that variable mapped to the target.
- If target is a variable but pattern is not, fail immediately (one-way matching only).
- the only possibility left is both of them are `TermInd`[5], so it can process as the following
  - If the grammar-choice of pattern and target is not the same, fail immediately.
  - Otherwise, construct n pairs of (sub-pattern, sub-target) where
    - n is the number of sub-term of pattern which must the same as number of sub-term of target.
    - i^{th} sub-pattern is constructed from i^{th} sub-term of pattern and i^{th} sub-grammar in the grammar-choice.
    - i^{th} sub-target is constructed from i^{th} sub-term of target and i^{th} sub-grammar in the grammar-choice.

    For each pair of (sub-pattern, sub-target), apply `pattern_match_get_vars_dict` then it will return either pattern-to-list-of-terms dictionary or failure.
    - If any of sub-result return as failure, fail immediately
    - Otherwise, merge all dictionaries of sub-results together and succesfully return it.

After `pattern_match_get_vars_dict` produces matching dictionary.
`vars_dict_to_pattern_matching_info` as shown in figure E.3 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[6] 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.

[5] A term can be either unfinished (`TermTodo`), variable (`TermVar`), or `TermInd`
[6] Unification process will be explained after this function.

Function `unify` as described in figure E.4 receives term `a` and `b` then try to build the most general unifier (mgu) which is the most general substitution list that is specific enough to make term `a` and `b` identical. This function is similar to pattern matching in `pattern_match_get_vars_dict` but this is two way matching and it is aware whether a variable is meta variable or literal.

Recall back to function `vars_dict_to_pattern_matching_info`, "unify the corresponded terms" means we will find the most general substitution list that can be used to substitute to each corresponded term and make every term identical. This achievable by

- apply function `unify` on the 1^{st} and 2^{nd} term, then we will get the first substitution list.
- substitute the first substitution list to the 1^{st} and 3^{rd} terms, then apply function `unify` to it, then append this substitution list to the first one to get the second substitution list.
- substitute the second substitution list to the original 1^{st} and 4^{th} terms, then apply function `unify` to it, then append this substitution list to the second one to get the third substitution list.
- keep repeat this pattern until the last term get substituted and unify, the latest substitution list will the most general substitution list that we want at the first place.

There are also other several auxiliary functions for pattern matching as described in figure E.5 which can be described as the following

- `substitute` — replace every occurrence of the specified variable ($1^{st}arg$) by the new term ($2^{st}arg$) inside the current term ($3^{rd}arg$).
- `multiple_root_substitute` — substitute a root term by a substitution list.
- `pattern_substitute` — substitute a term by pattern-to-root-term dictionary.
- `pattern_root_substitute` — the same as `pattern_substitute` but substitute to a root-term rather than a term.
- `pattern_matching_info_substitute` — update `PatternMatchingInfo` by a substitution list.
- `pattern_matchable` — check whether the pattern root-term can be pattern matched against target root-term or not.
- `merge_pattern_variables_list` — merge all of pattern-to-list-of-terms dictionaries in to a single dictionary.

7.6 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 expected 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 7.13: 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 7.14: 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 7.15: The main test suite which is at `tests/Tests/Main.elm`

To run[7] 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[8].
- 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.

Last but not least, *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] The instruction of running test suite is shown here to give a favour to reader that how did I test my code, rather than asking user to run the test.

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

Chapter 8

Evaluation and Conclusion

8.1 Users Feedback — discuss with friends

On the 25^{th} of May 2016, it was the first day of a project fair where students can demonstrate their work to other students to get a feedback so I went there and discuss about my project. At this stage, the implementation was 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 prove a theorem. Finally, I asked them to try Propositional Logic, some of them showed interest but not all of them wanted to.

From my observation, all of them were comfortable to prove 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 had time to read the whole tutorial.

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

There were several improvements that they suggested. Some 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.

8.2 Users Feedback — discuss with junior students

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

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

- One student 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's internal mode.
- One group of students accidentally clicked theorem reset button () because they thought that was an 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, and once they understood more of the tool, they could use it easily. In fact, the tutorial that I have written already covered most of these mistake, but I didn't ask students to read though the entire tutorial due to time constraints.

On the other hand, I was surprised at 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 the first year Logic course. Moreover, one of second students told me that he didn't particularly like derivation trees[3] but Phometa managed to make him enjoy constructing this derivation tree.

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

8.3 Strengths

- Phometa specification itself is more powerful than traditional derivation systems because it has extra features such as cascade premise and meta-reduction. Thus, it is 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 a less steep learning curve than mainstream proof assistants because the specification is small enough for a user to learn in a short time and all of components are diagram based which is easier to understand than sequences of characters.
- If a term can be constructed, it is guaranteed to be well formed. And if it is a goal of a proved theorem (or lemma) it is definitely valid based on soundness of rules on that formal systems.
- The repository of phometa is always in a consistent state. Phometa is quite cautious 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 that when a node is deleted, Phometa will delete all of the nodes that depend on it as well[4]. This is opposite to text-based proof assistants where user has full control over the repository, and if the repository is an inconsistent, the compiler will raise an error and user would fix it.
- Lemmas allow reuse of proofs so there is no need for duplication. Users 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 systems can be constructed in a more mathematical friendly environment.
- It is a web-application so it can run on any machine that supports a web-browser. One might argue that it required Python for the back-end but most machines support Python out of the box anyway.

8.4 Limitation

- It is hard to extend formal systems at the moment because Phometa doesn't allow grammar to inherit choices from another grammar. If a user wanted to extend a formal system, they would need to create a new one from scratch. For example, first order logic cannot be built from the existing propositional logic. This can be solved by making grammars extensible as described in future work.
- Phometa doesn't support automation well i.e. when a user constructs a proof, they need to say which rule or which lemma should be used. 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 a 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 sets 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 ended-up using `Alt` combined with a visible character to create a Phometa shortcut. This might have unwanted side effects but at least it work reasonably well with Google Chrome[5] under a condition that the window containing Phometa has only one tab, so it will not suffer from `Alt-{1..9}` which are used for switching tabs. However, this is not such a serious problem since user can always click a command in the keymap pane directly. Alternatively, this can be solved completely by making the keyboard shortcut customisable e.g. if some keyboard shortcut crash and user really want to use it, they can bind this command to another keystroke. Again, it is a good consideration for future work.
- 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 requires a user to start a local server for individual use. It doesn't have a proper server where user can enter a link to and use it directly. To implement such a proper server, it requires user accounts and a database to manage users repositories, although it is possible to implement but it seems to overkill method and doesn't match project objective. It had lower priority than other features and can be considered as future work.
- Directly modifying `repository.json` before it is loaded into Phometa could result in undefined behaviour. This is because Phometa currently doesn't have any mechanism to verify consistency of repository before it will be loaded. This shouldn't cause any problem if user only interacts with repository via Phometa interface and does not try to hack the repository file directly. To solved this problem, we can write a function to verify consistency of loading repository as discussed further in future work.

8.5 Lesson Learnt

Time management for a research project is one of the many things that I have learnt during this project. I learnt that tasks always take time twice or thrice longer than expected so it is vital to leave plenty of time before the deadline. More importantly, I learnt that a better idea of feature always comes 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 trade-off between usefulness of the feature and the risk of the project not being finished in time. These 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

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

feature, this would take sometime to implement and to edit the related part of this report, which in turn, would impact the entire schedule of the plan. So I usually took it as future work as described in next section.

I also learnt to believe in myself being capable to build something I dream of. Formal proof always was 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 decided 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 confidence in that field. Oppositely, the 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.

8.6 Future Works

Although Phometa has been designed and implemented up to a satisfactory level, there is still plenty of room for improvements. 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 to the base grammar.
- **Plain-text as alternative term input method** — When constructing a term, user should be able to write some terms 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 could be applied next (similar to Isabelle).
- **Importation between Modules** — Modules should be able to import some nodes from other modules, this improving scalability since a formal system can be expanded across multiple modules.
- **Exportation to LaTeX** — Grammars, Rules, Theorems, and Proofs should be exportable to LaTeX source code. So it can be used further in other documents such as reports or presentations.

- **Repository Verification on Loading** — Creating a function that checks whether the repository is in a valid state or not. Normally, it is not necessary as Phometa will always be in a consistent state, but when you want other people to use a formal system by obtaining `repository.json`, it is not guaranteed that someone will not modify that repository directly. So it is better to have a mechanism to check consistency of the repository anyway.

There are also other minor improvements such as

- Adding new formal systems to the standard library.
- Making undo buttons.
- Making notification message become more informative.
- Deploy phometa to a proper server.
- Making user interface become more mobile friendly.
- Making a theme and keyboard shortcut become configurable.

8.7 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 turn, satisfies all of the objectives stated in the introduction chapter. In addition, I also believe that Phometa on this state can be a potential replacement for derivation-tree's manual-drawing so people don't have to suffer from it tedious process and error prone anymore.

Appendices

Appendix A

Example Formal System — Propositional Logic

Once you are familiar with some basic features and usability of *Phometa* from the chapter 4. 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.

A.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[20]

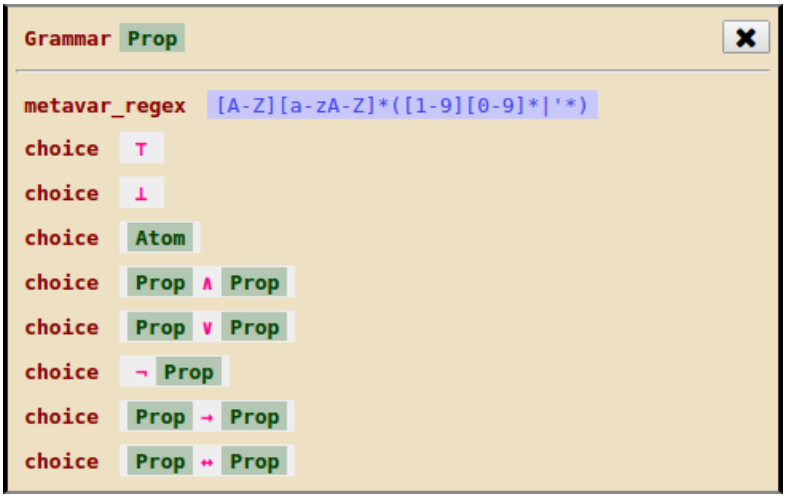


Figure A.1: Definition of `Prop`

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 A.2: Backus-Naur Form correspond to `Prop`

On the 3^{rd} 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.

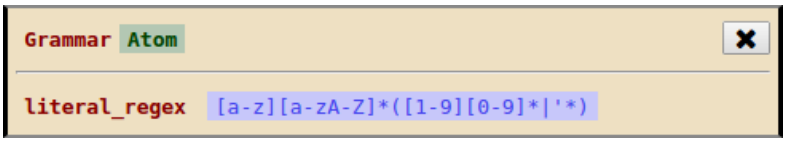


Figure A.3: Definition of `Atom`

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 A.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 A.5: Definition of `Judgement`

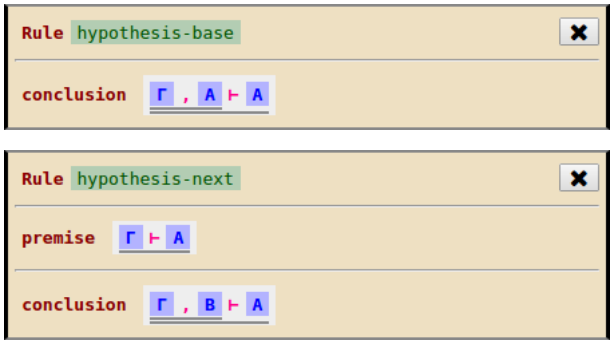
`Judgement` has a meaning of validity. For example, validity of `ε , p , q ∨ r ⊢ p ∧ q ∨ p ∧ r` means, assuming that `p` and `q ∨ r` hold then `p ∧ q ∨ p ∧ r` holds.

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`.

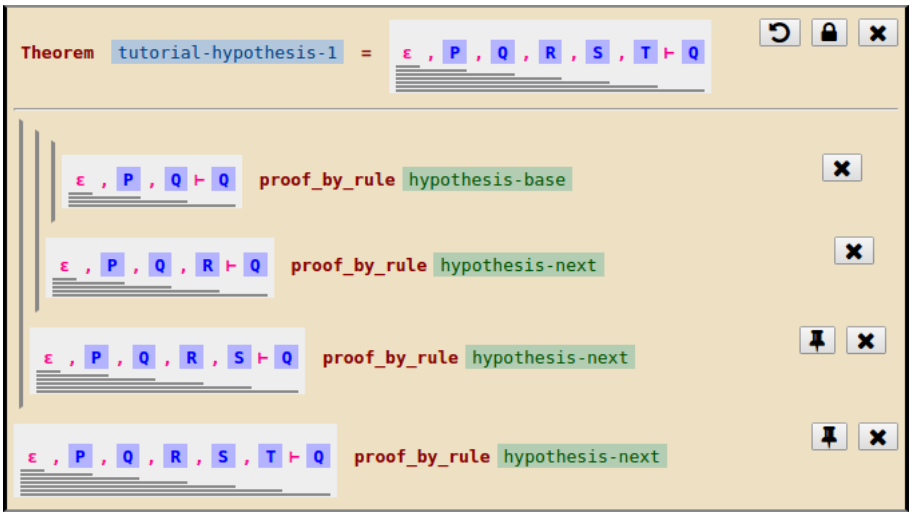
A.2 Hypothesis rules

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. $\epsilon, A_1, A_2, \ldots, A_n \vdash A_i$ where $i \in 1..n$

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



`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.



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.

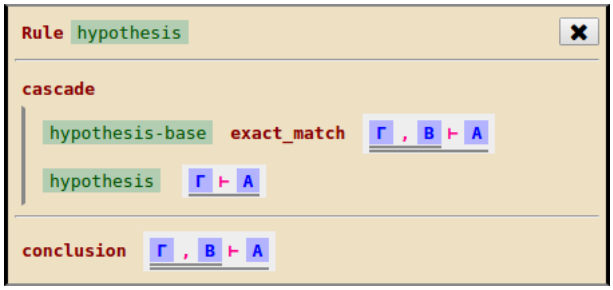
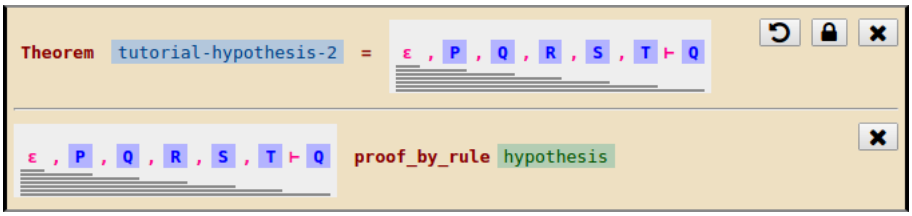


Figure A.6: rule `Hypothesis`

`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, *cascade*s 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



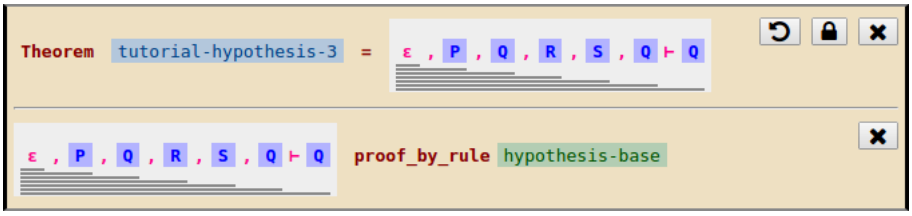
To show the process, first `hypothesis` conclusion — Γ , B ⊢ A is pattern match against goal = ε , P , Q , R , S , T ⊢ Q, this results in A = Q, B = T, and Γ = ε , P , Q , R , S. This cascade premise try to apply `hypothesis-base` with goal[3] = ε , P , Q , R , S , T ⊢ Q.

`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.



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 turn, 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.

A.3 Main Rules

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

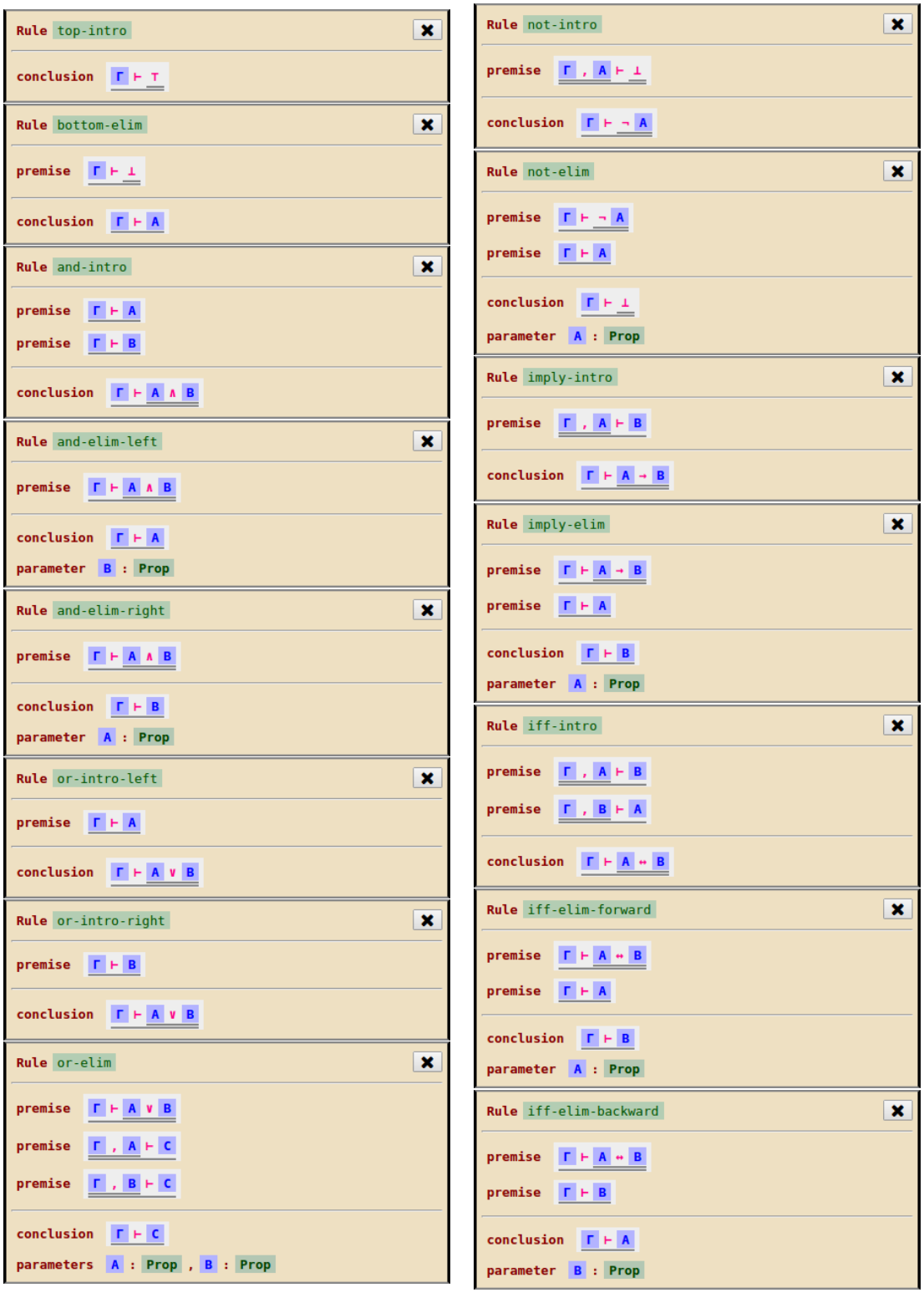
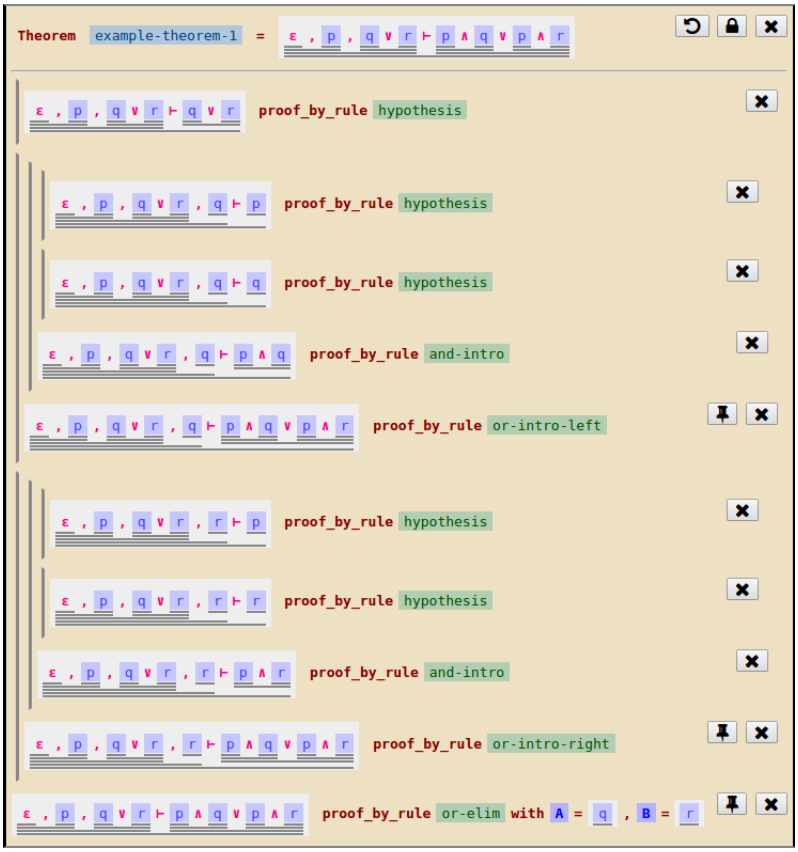
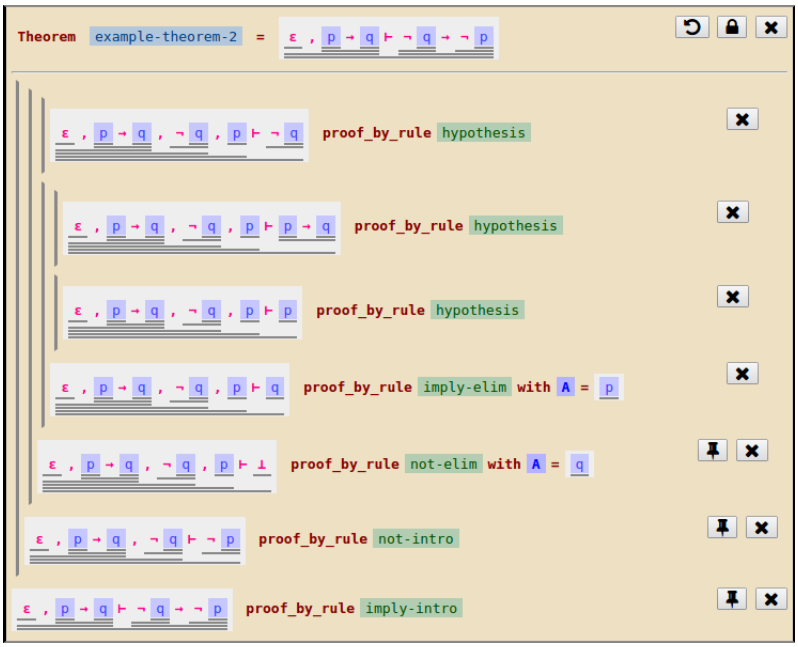


Figure A.7: Main Rules for Propositional Logic

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

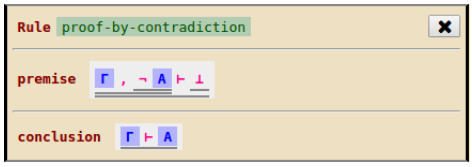




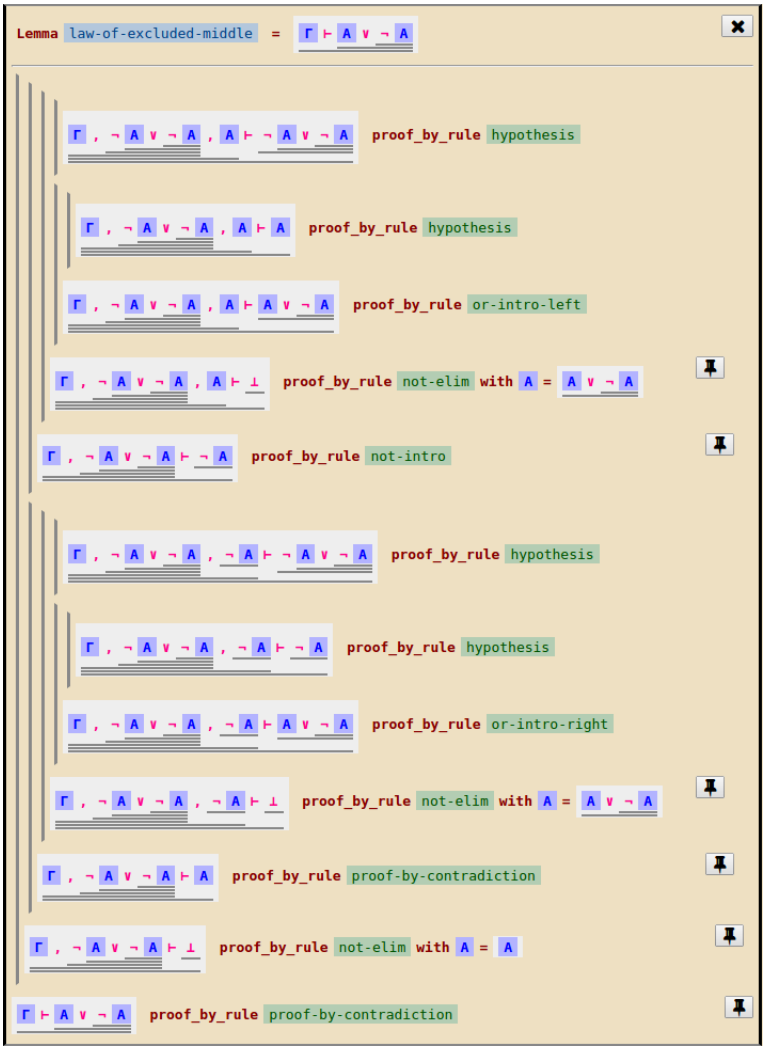
A.4 Classical Logic

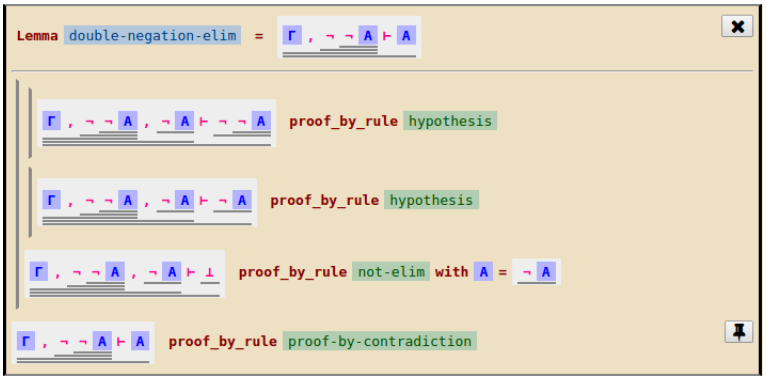
The rules so far create Intuitionistic Logic i.e. it doesn't assume that each proposition must be either true or false. Hence, cannot prove some thing like `A ∨ ¬ A`.

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.

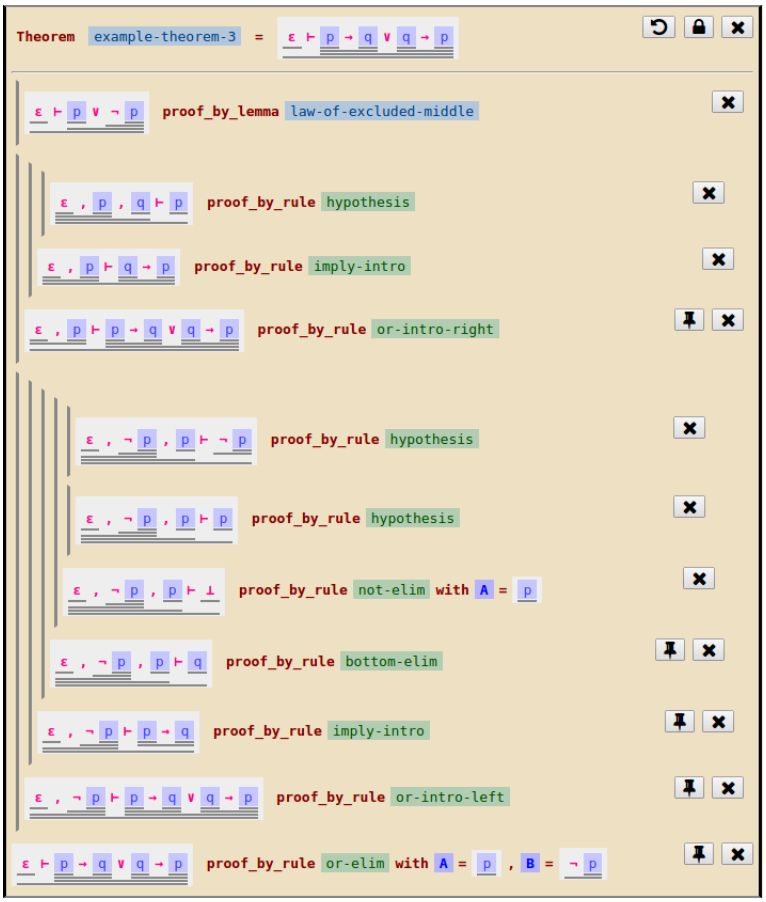


And now we can prove `law-of-exclude-middle` and `double-negation-elim` as lemmas.





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



A.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.

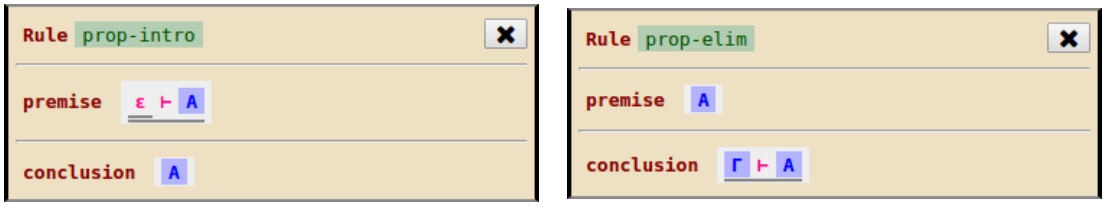
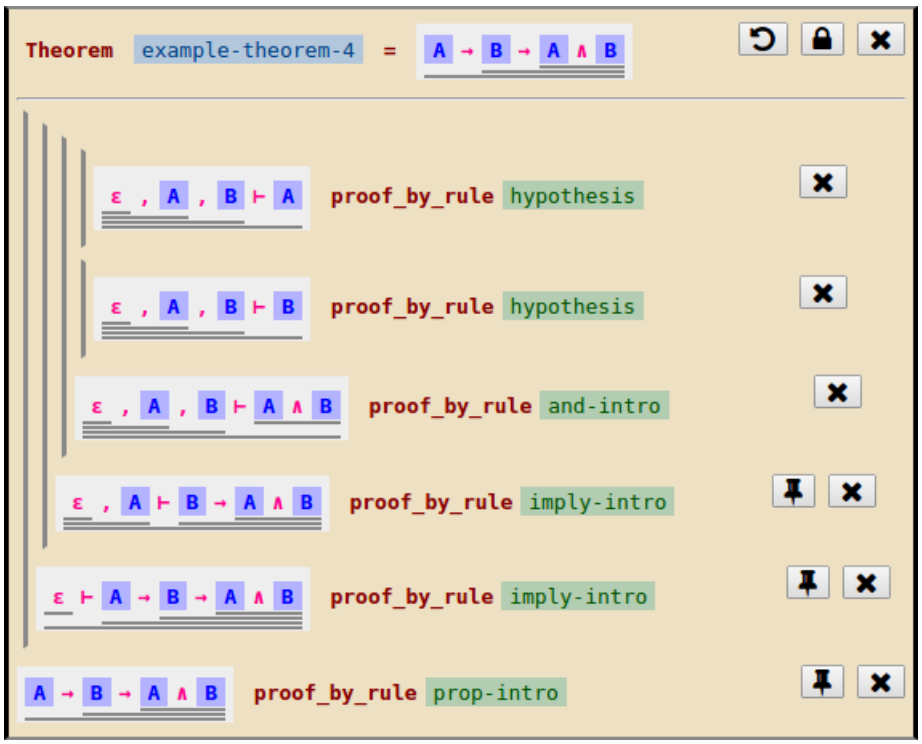


Figure A.8: `prop-intro` can be used to prove that a `Prop` is valid. And its duality, `prop-elim` can be used to prove `Judgement` when its conclusion is valid.

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.



A.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

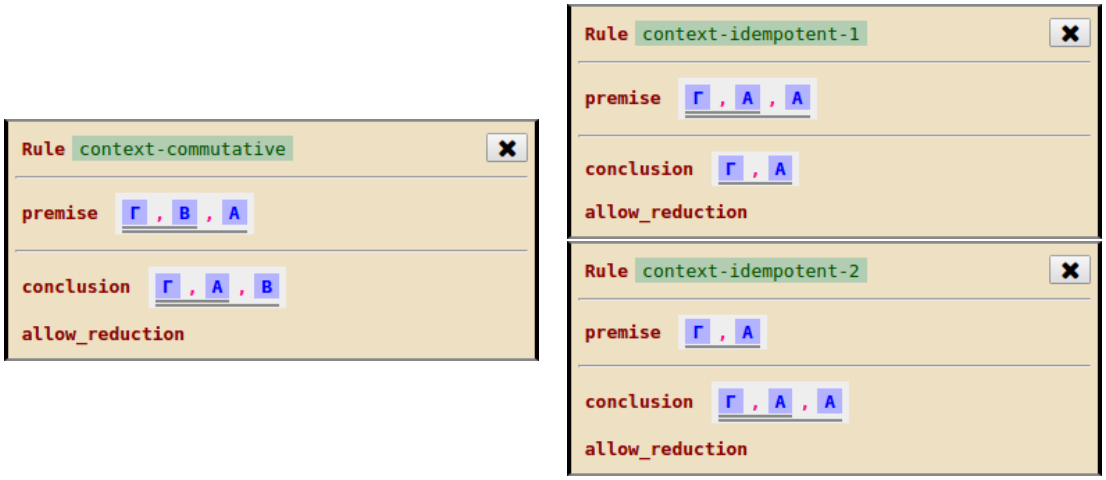
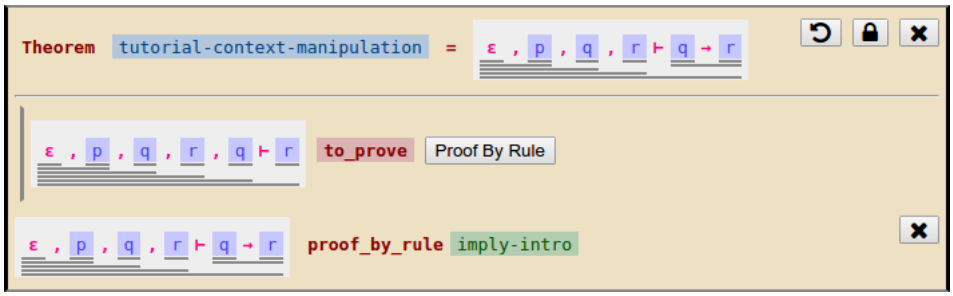


Figure A.9: Rules for context manipulation

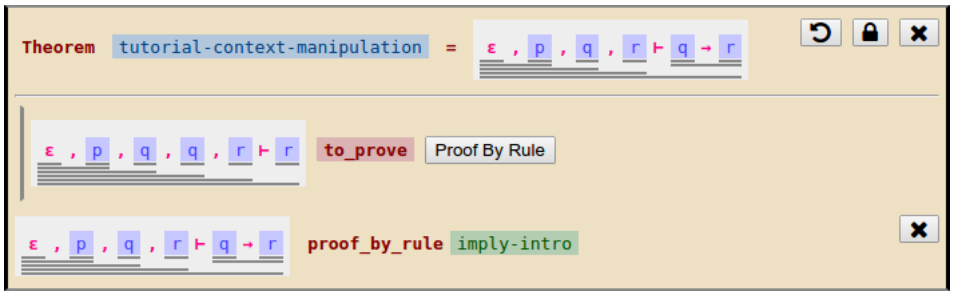
For example, let construct an unfinished theorem as the following



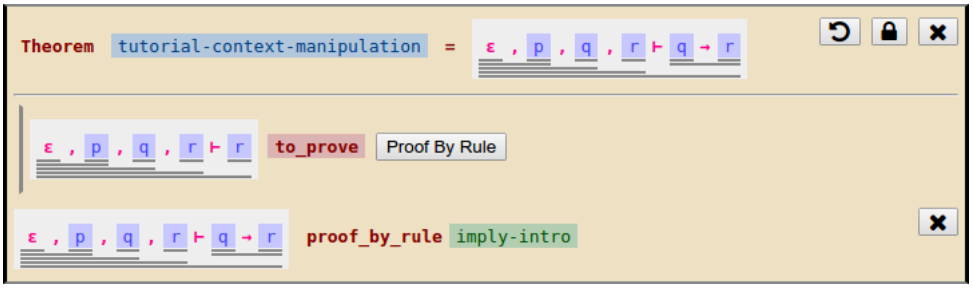
Then, if you click `ε , p , q , r , q` on the sub-goal, the keymap will look like this

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.



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.



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*[4] (sub)goal 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.

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

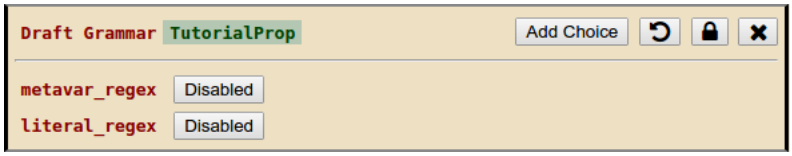
A.7 How to build Grammars and Rules

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.

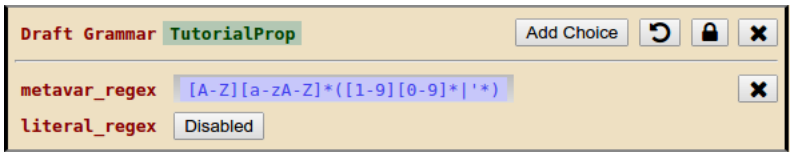
To show how to build these, I will recreate `Prop` for grammars and recreate `or-elim` and `hypothesis` for rules.

A.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.



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,

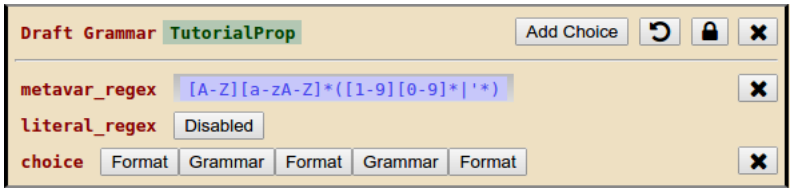


The background colour of this regular expression is grey, so you can edit it afterword by click the box again. Alternatively you can delete it using ✖ on the right of the row.

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

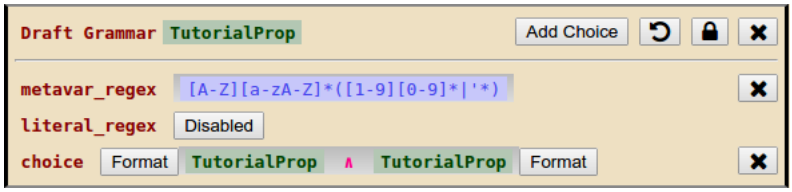
Next, we will add choices to the grammar, I will start with the one that has "∧" connector. Click "Add Choice" button on the header of this grammar, it will tern to be input box that you can write number of sub-grammars for this choice, in this case it is 2, so write it and press enter.

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



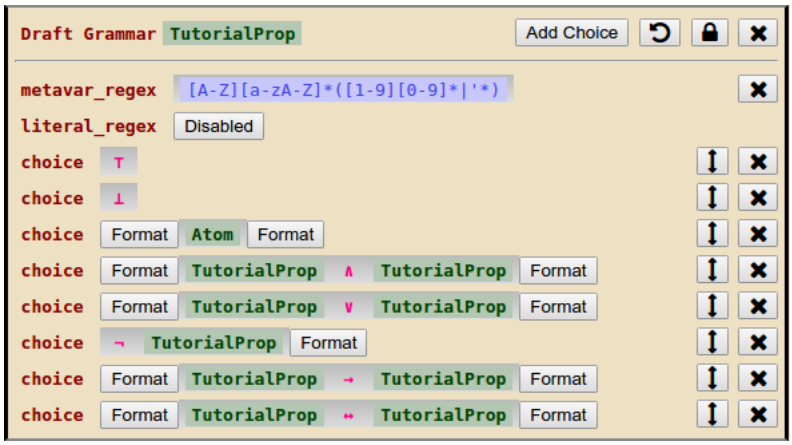
You will see a `choice` that have buttons labelled "Grammar", you can click it to specify a sub-grammar. In this case, select `TutorialProp` in both of position.

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.

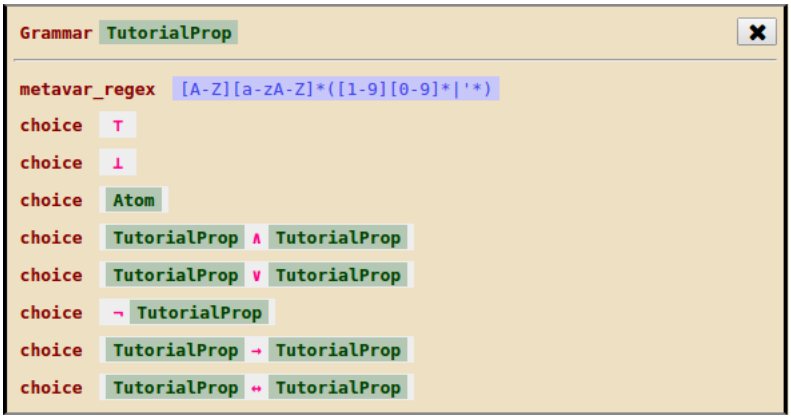


The background colour of these are gray so you can edit it similar to `metavar_regex` . For the first and last "Format" buttons, we do nothing on it as this choice make no use on those positions.

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

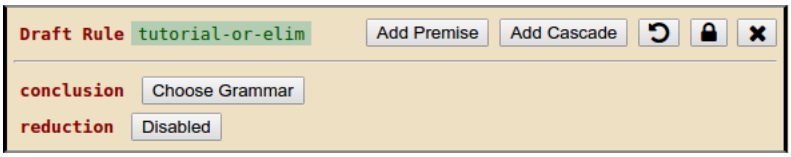


You can lock this draft grammar by clicking 🔒 on top-right corner, this will transform it as a proper grammar where you can refer it in rules and theorems.

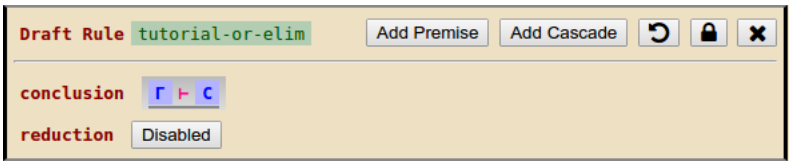


A.7.2 Recreation of `or-elim`

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



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.



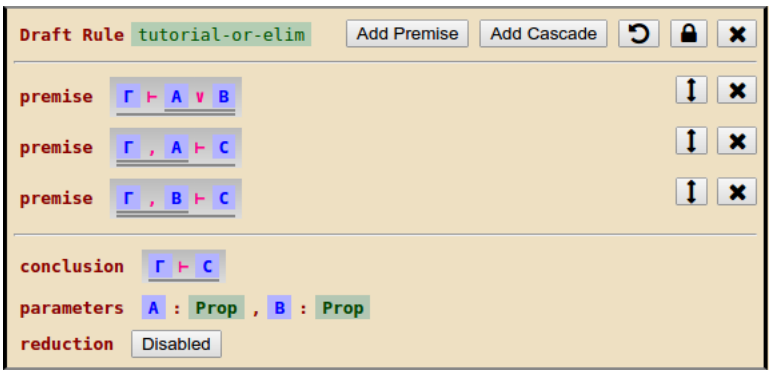
Next is `allow_reduction` it is disabled by default, we could click on button "Disable" to toggle it to "Enable", however, we don't want to rise this flag for `tutorial-or-elim` so leave it as it is.

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

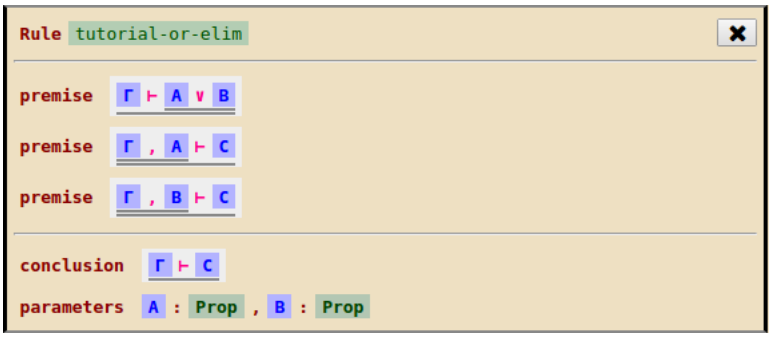
Then write a term similar to what we have done in `conclusion`

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

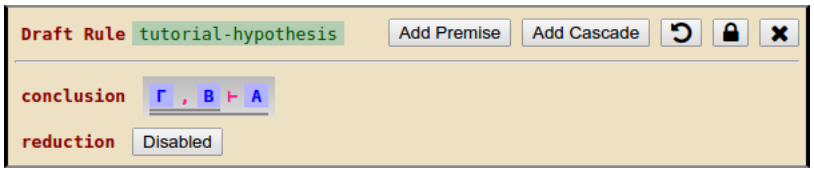


You can lock this draft rule by clicking 🔒 on top-right corner, this will transform it as a proper rule where you can refer it in theorems.

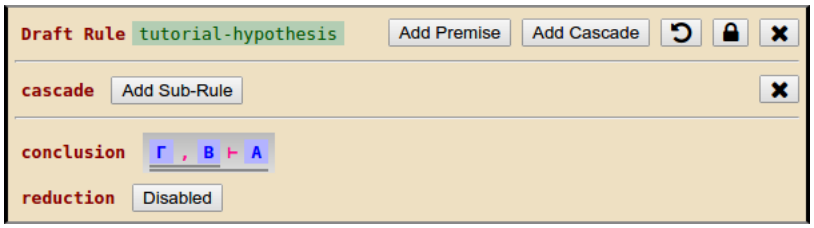


A.7.3 Recreation of `hypothesis`

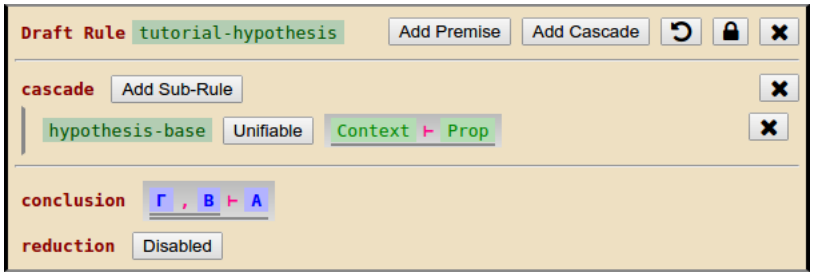
The recreation of `or-elim` shows most of features of rule construction except how to deal with cascade so we will recreate `hypothesis` to illustrate this. First, adding a rule with name "tutorial-hypothesis" on it. Then, enter a term for `conclusion` .



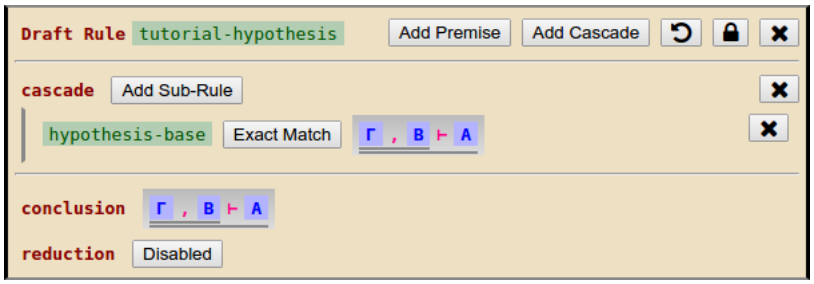
Then, add a cascade premise by clicking "Add Cascade" on the top-right corner.



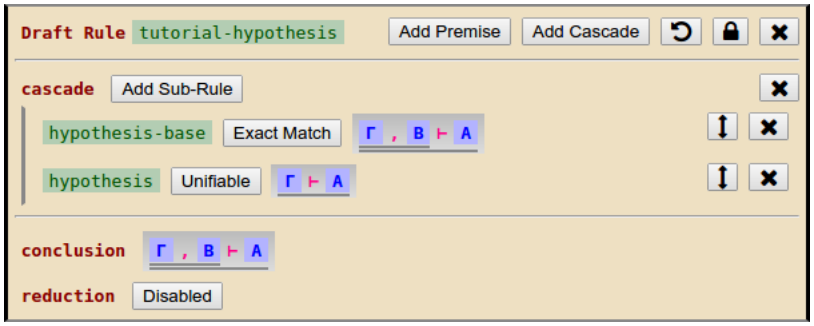
Then click "Add Sub-Rule" to add the first sub-rule, the button will transformed into input box where you can select such a rule, in this case select `hypothesis-base`



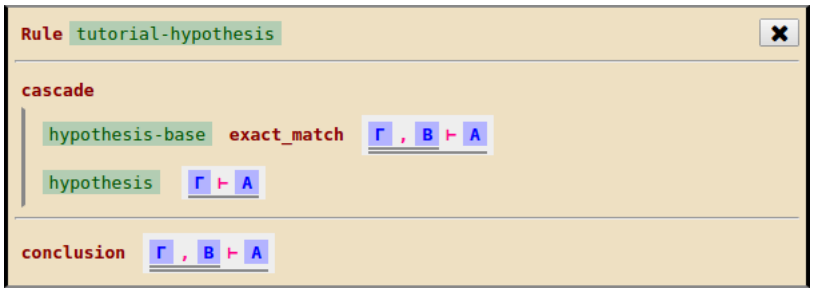
Sub-rule calling-template will appear, you can construct a term that will passed into this sub-rule. A sub-rule is unifiable by default, you can set it to `exact_match` by toggling button "Unifiable".



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



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



Appendix B

Example Formal System - Typed Lambda Calculus

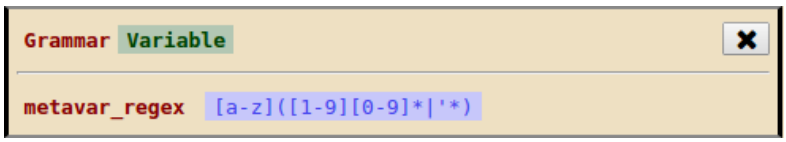
Chapters 4 and A show most of features and usability of *Phometa* already so this chapter aims to show that Phometa is powerful enough as it can even encode more complex formal system like *typed lambda calculus*. Hence, it is clear that Phometa is suitable to encode most of formal system that user can think of.

Credit: Some of material here modified from lecture note of "382 — Type Systems for Programming Languages" (third year course), Department of Computing, Imperial College London. Thank you Dr Steffen van Bakel for this.

Please note that the following formal system is just a faction of what Lambda Calculus could be. To be precise, this formal system includes β-reduction and simply types of Lambda Calculus.

B.1 Terms and Variables

Lambda Calculus stimulates computational model in functional manner. Basically, it will have a grammar `Term` which can be either a `Variable` , an anonymous function, an application, or a substitution. It can be described in Backus-Naur Form like this



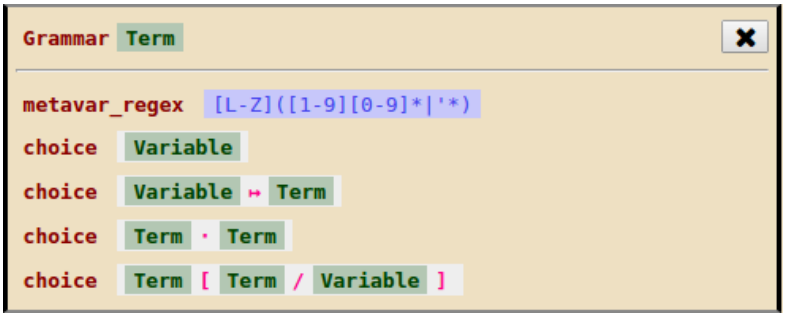


Figure B.1: Definition of `Variable` and `Term`

The first choice of `Term` represents variable which you can think as a mark point that other terms can refer to.

The second choice of `Term` represents an anonymous function i.e. `x ↦ M` means a function that get a variable `x` and return a term `M` where `M` might contain `x`. From now on, I will call as "blinder"

An anonymous function is usually represented by `λ x . M` but I use `x ↦ M` since it works better with underlines when we implement it in Phometa.

The third choice of `Term` represents an application i.e. `M · N` means apply term `M` to term `N` where `M` usually be an anonymous function[1].

The fourth choice of `Term` represents a substitution i.e. `M [ N / x ]` means a term `M` that every occurrence of free variable `x` will be replaced by a term `N`.

A variable `x` is free variable if it doesn't live inside an anonymous function that has `x` as blinder[2], otherwise, it is bounded variable.

B.2 Side Conditions

Sometime we want to build a simple formal system but its dependency relies on more complex formal system, for example, β reduction (will be defined later) needs to check whether a variable is a free variable of a curtain term or not, and checking free variable require knowledge of sets. Normally we should build a formal system regarding to set, then we can build β reduction, however this is overkill as formal system of set is much larger than β reduction. To solve this problem, we can build a grammar that construct a statement that need to be judge by the user as the following

[1] Or will become anonymous function after evaluate the term.
[2] Blinder is a variable on the left hand side of $\mapsto$ of an anonymous function

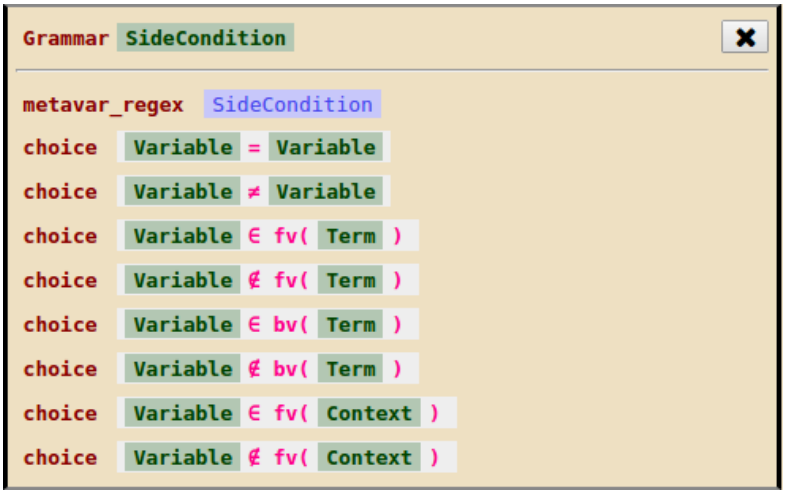


Figure B.2: Definition of `SideCondition`

Then write a single rule that allow to prove any side condition.

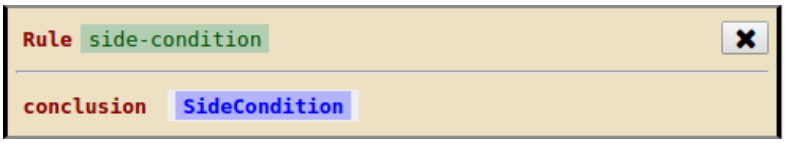


Figure B.3: Definition of `side-condition`

Every time when `SideCondition` appear in a derivation tree, user needs extra care and determine whether that side condition holds or not. If it holds then apply `side-condition` and it is done. Please note that there is no mechanism to prevent user to apply `side-condition` on a false `SideCondition` and it will result in inconsistency, in another word, by using side condition technique, we *trust* user to do the right thing.

B.3 β Reduction

Since `Term` represents computational model, so it can be *evaluated*. A step of evaluation in Lambda Calculus is called `β-Reduction` which can be defined as the following

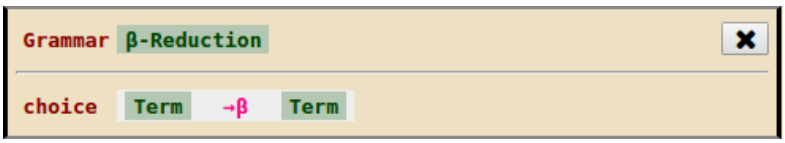


Figure B.4: Definition of `β-Reduction`

M →β N means M can reduce to N in one step, the rules to control this can be defined as the following

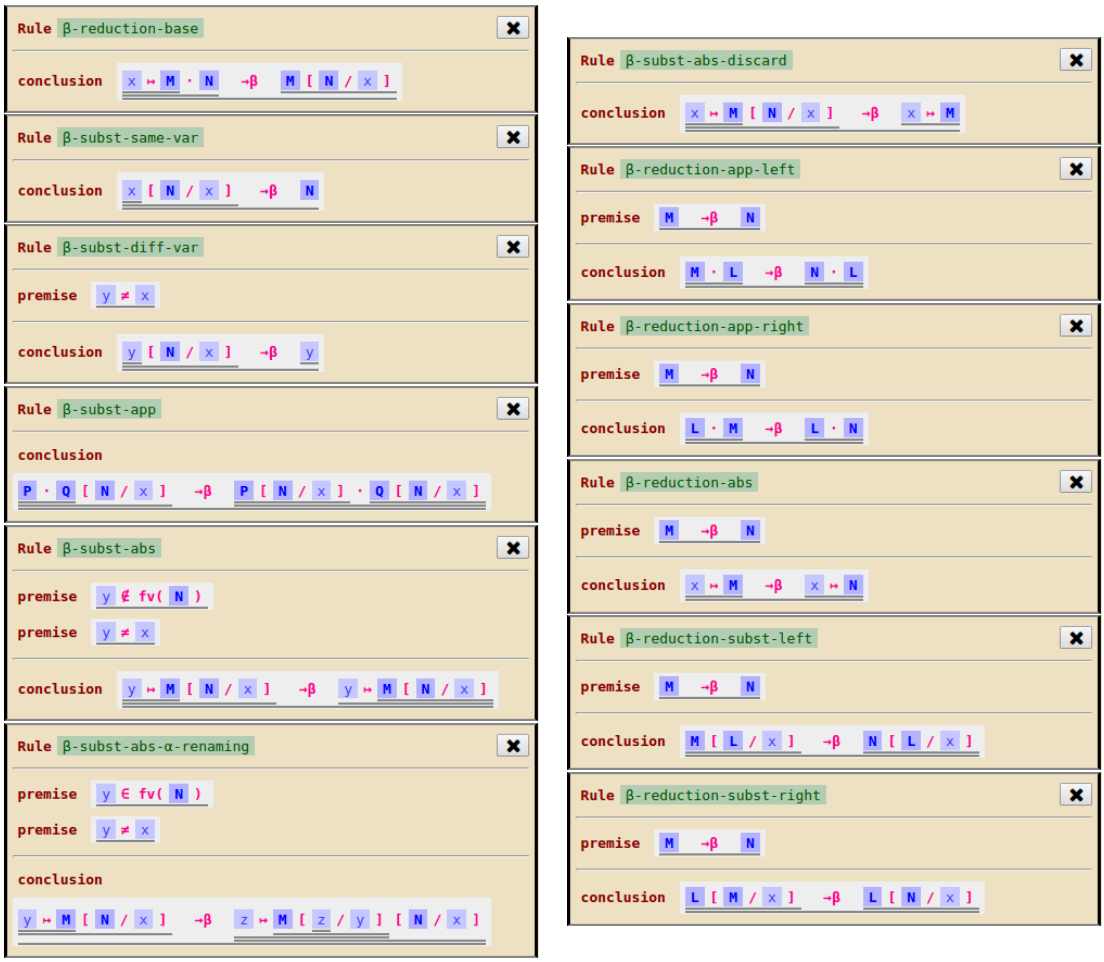


Figure B.5: Rules for β-Reduction

There is also β-ManyReduction which reflexive transitive closure of β-Reduction.

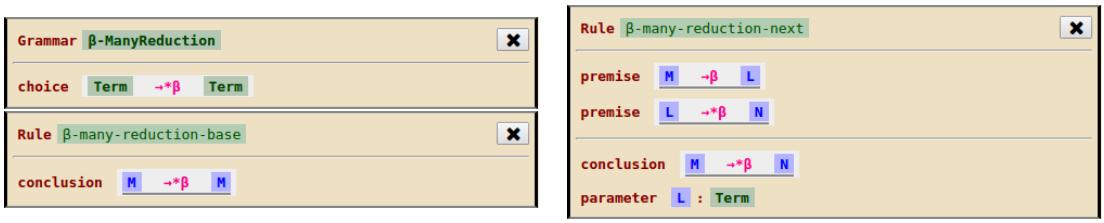


Figure B.6: Definition of β-ManyReduction and its rules

Now we are ready to build a derivation tree,

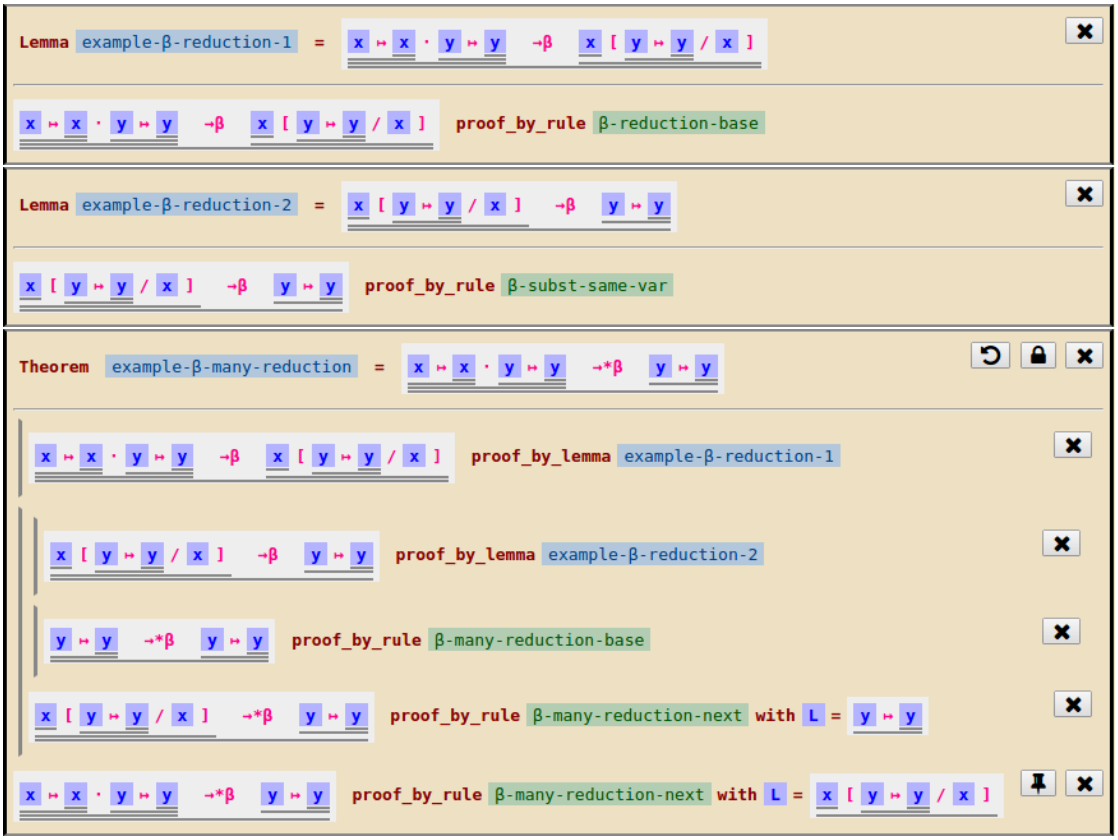


Figure B.7: Examples of derivation trees of β-Reduction and β-ManyReduction

B.4 Simply Types

Some terms can be reduced infinitely many times e.g. `x ↦ x · x · x ↦ x · x` →β `x · x [ x ↦ x · x / x ]` →β `x [ x ↦ x · x / x ] · x [ x ↦ x · x / x ]` →β `x ↦ x · x · x [ x ↦ x · x / x ]` →β `x ↦ x · x · x ↦ x · x` →β ⋯

This is not desirable as it implies a program that will never terminate. To solve this problem we can try to find a type of a term, if found, we know that the term as finite number of reductions. `Type` can be defined as the following

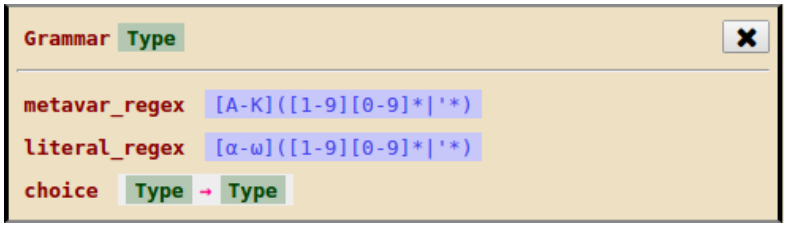


Figure B.8: Definition of `Type`

`Type` can be instantiated as

- meta variable — for generic type which can be used in another place.
- literal — for constant type similarly to Int, Bool, Char, etc.
- `A → B` — a type of anonymous function that receive input of type `A` and return output of type `B`.

To state that a term has the particular type we can use `Statement` , and in order to prove a that `Statement` holds under certain assumptions, we need `Context` and `Judgement` . These three grammars can be defined as the following

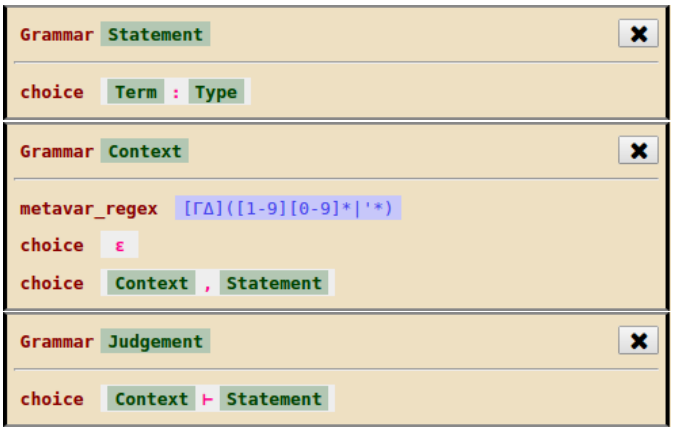


Figure B.9: Definition of grammars related to `Type`

The rules regarding to type can be defined as the following,

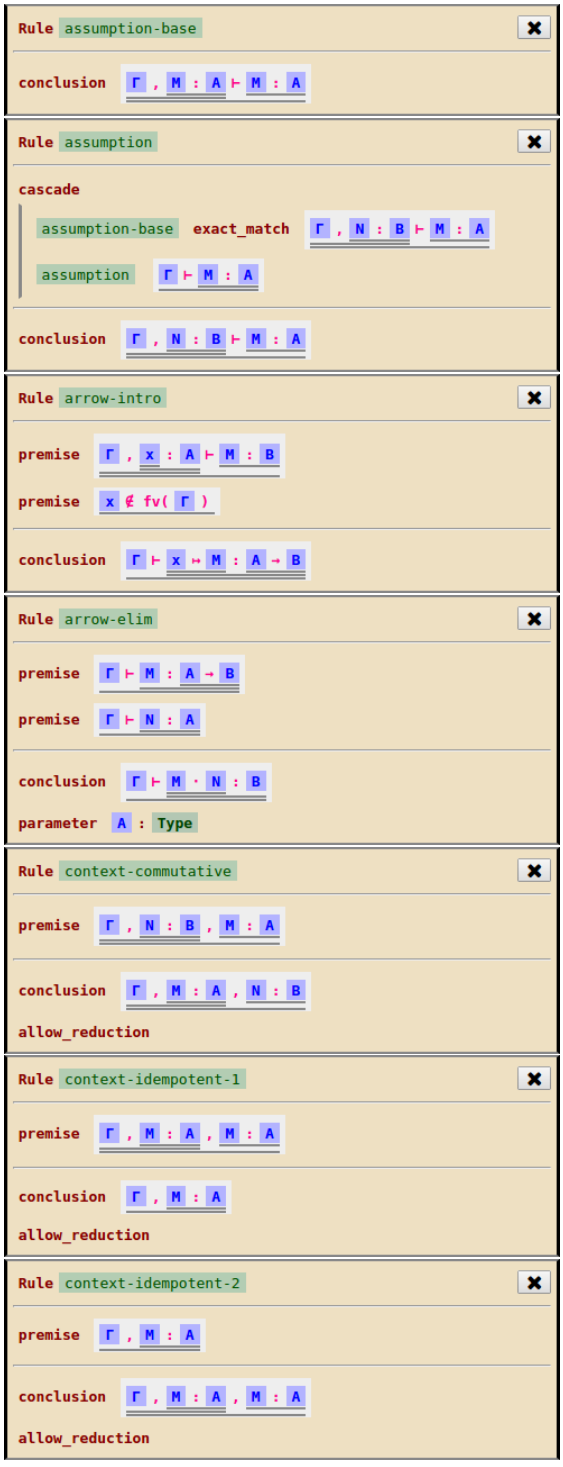


Figure B.10: Definition of rules using for type resolution.

Now we are ready to build a derivation tree,

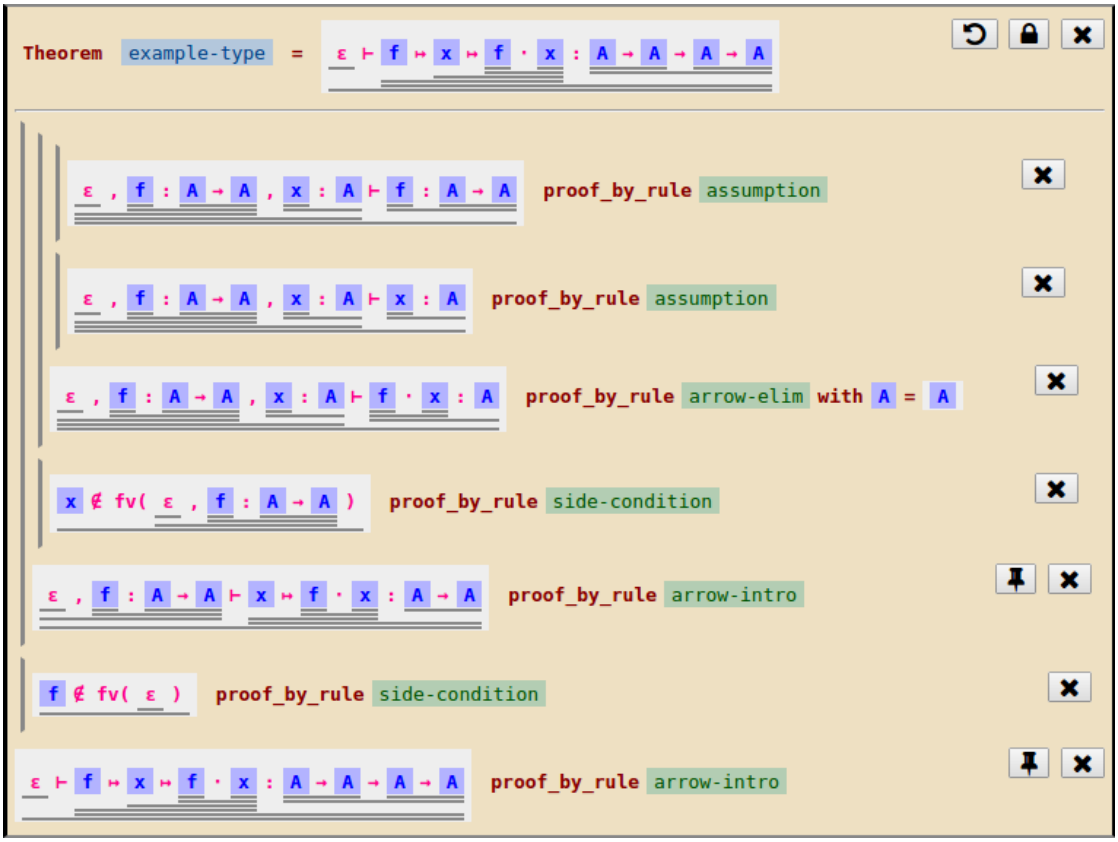


Figure B.11: Examples of derivation trees of `Judgement` .

Appendix C

Exercises on Examples Formal Systems

C.1 Simple Arithmetic

- 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 A.7 in the next chapter.

C.2 Propositional Logic

Credit: Some of material here modified from tutorial 3, 4, and 5 of "140 - Logic" (first year 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 A.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.

Appendix D

The Phometa Repository

D.1 Installation and Startup

You can download the compiled version of Phometa at

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

Once you unzip this file, you see several files as the following

- `phometa.html` — the main file containing Phometa web-based application.
- `style.css & style.css.map` — control the layout and theme of `phometa.html`
- `naive.js` — JavaScript code that will be injected to `phometa.html`
- `repository.json` — contain proofs repository, can be loaded or saved by `phometa.html`
- `logo.png` — image that will be used as favicon in `phometa.html`
- `phometa-server.py` — python script that can be executed as local server.
- `phometa-doc.pdf` — this report, included here as the tutorial and reference.

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 figure 6.1

[1] but Google Chrome is recommended

D.2 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 [2] has the structure as the following

- `build/` — contains compiled version of Phometa and PDF version of this report.
- `doc/` — contains source code and supplementary images of this report.
- `scripts/` — contains scripts that can compile 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 compiled 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.

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

D.3 Building a Compiled Version

We can build the compiled 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 compiled recursively. It can be outputted as HTML file (for standalone application) or JavaScript file (to embedded it as an element of bigger application).

D.4 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 turn, 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`.

Appendix E

Supplementary Source Code

Source code is important for implementation chapter yet it make the chapter hard to read, so we cut some of them in order to make the implementation chapter easier to read and paste it here so you can look at it if you are curious.

E.1 Pattern Matching and Unification

All codes inside this section come from `src/Models/RepoUtils.elm`.

```
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 E.1: Functions `pattern_match` and `pattern_match_multiple`.

```
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 E.2: Function `pattern_match_get_vars_dict`.

```
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 E.3: Function `vars_dict_to_pattern_matching_info`.

```
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 E.4: Function `unify`.

```
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 E.5: Other auxiliary functions for pattern matching.

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] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. San Diego, CA, USA: Academic Press Professional, Inc., 1985. ISBN: 0-12-141250-4.

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

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

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

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

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

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

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

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

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

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