Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering

Automated proof-checking of the Rose-Rosser's proof of completeness of Łukasiewicz propositional logic

Strojové ověření Rose-Rosserova důkazu úplnosti Łukasiewiczovy výrokové logiky

Master's Thesis

Diplomová práce

Author: **Bc. Jáchym Šimon**

Supervisor: **Mgr. Tomáš Lávička, Ph.D.**

Language advisor: **Mgr. Isa Dallmer-Zerbe**

Academic year: 2021/2022

Katedra: matematiky Akademický rok: 2020/2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Jáchym Šimon
Studijní program:	Aplikace přírodních věd
Studijní obor:	Matematické inženýrství
Název práce (česky):	Strojové ověření Rose-Rosserova důkazu úplnosti Łukasiewiczovy výrokové logiky
Název práce (anglicky):	Automated proof-checking of the Rose-Rosser's proof of completeness of Łukasiewicz propositional logic

Pokyny pro vypracování:

1) Seznámit se základními pojmy Łukasiewiczovy výrokové logiky a Rose-Rosserovým důkazem její standardní úplnosti (tj. tvrzení, že spojky dané logiky odpovídají jistým spojitým operacím na jednotkovému intervalu reálné osy). Zde budeme čerpat zejména z nedávné bakalářské práce [4], která vychází z původních [1], [2].
2) Dále se zdokonalovat v práci s programem Lean, ve kterém bude důkaz formalizován [5]. Zde pro získání hlubších poznatků budeme využívat i kontakt se skupinou aktivně vyvíjející Lean (internetová fóra).
3) Využít základů položených v posluchačově bakalářské práci [3], kde bylo strojově ověřené podobné tvrzení pro klasickou výrokovou logiku. To provést provést podle bodu 4–6.
4) Přeformulovat základní syntaktické definice z [3] na Łukasiewiczovu logiku, tzn. pojmy jako spojky, formule, důkaz apod.
5) V Leanu vyvinout a zdokonalit naše vlastní dokazovací taktiky (tj. podprogramy sloužící k automatickému generování důkazů) a použít je k ověření rozsáhlých syntaktických důkazů z [2].
6) Ověřit druhou polovinu Rose-Rosserova důkazu úplnosti Łukasiewiczovy výrokové logiky z [2]. Zde se pokusíme využít již dokázané výsledky z leanovské knihovny Mathlib.
7) Sepsat práci v LaTexu v anglickém jazyce.

Doporučená literatura:

1) J. Łukasiewicz, O logice trójwartościowej (On Three-Valued Logic). Ruch Filozoficzny 5, 1920, 170–171.

2) A. Rose, J. B. Rosser, Fragments of Many-Valued Statement Calculi. Trasnsactions of the American Mathematical Society 87, 1958, 1–53.

3) J. Šimon, Strojové ověření konstruktivního důkazu úplnosti klasické výrokové logiky. Bakalářská práce, FJFI ČVUT v Praze, Praha, 2019.

4) P. Fejl, Moderní čtení Rose-Rosserova důkazu úplnosti Lukasiewiczovy logiky. Bakalářská práce, FJFI ČVUT v Praze, Praha, 2017.

5) J. Avigad, L. de Moura, S. Kong, Theorem Proving in Lean. Online tutorial: https://leanprover.github.io/theorem_proving_in_lean

Jméno a pracoviště vedoucího diplomové práce:

Mgr. Tomáš Lávička, Ph.D.
Ústav teorie informace a automatizace AV ČR, v.v.i., Pod Vodárenskou věží 4, 182 08 Praha 8

Jméno a pracoviště konzultanta:

Datum zadání diplomové práce: 28.2.2021
Datum odevzdání diplomové práce: 5.1.2022
Doba platnosti zadání je dva roky od data zadání.

V Praze dne 2.3.2021

[illegible]
garant oboru

[illegible]
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-1-

[illegible]
děkan

Acknowledgment:
I would like to thank my supervisor Mgr. Tomáš Lávička, Ph.D. for his expert guidance and valuable notes and express my gratitude to Mgr. Isa Dallmer-Zerbe for her language assistance and support. Thank you.

Author's declaration:
I declare that this Masters's Thesis is entirely my own work and I have listed all the used sources in the bibliography.

Prague, May 2, 2022 Bc. Jáchym Šimon

[illegible]

Název práce:

Strojové ověření Rose-Rosserova důkazu úplnosti Łukasiewiczovy výrokové logiky

Autor: Bc. Jáchym Šimon

Obor: Matematické inženýrství

Druh práce: Diplomová práce

Vedoucí práce: Mgr. Tomáš Lávička, Ph.D. Ústav teorie informace a automatizace AV ČR, v.v.i., Pod Vodárenskou věží 4, 182 08 Praha 8

Abstrakt: Tato práce se zabývá formalizací základů Łukasiewiczovy logiky a strojovým ověřením větší části prvního známého důkazu její úplnosti, t.j. za prvé ověřením celkem 23 lemmat s rozsáhlými syntaktickými důkazy, za druhé formalizací pojmu polynomické formule a ověřením s tímto pojem spojených 16 vět. Práce popisuje hlavní body této formalizace- jedná se především o základní definice, obecné věty (např. pravidlo řezu) a implementace vlastních taktik pro automatické dokazování, např. taktiky pro automatické ověřování dlouhých syntaktických důkazů zmíněných lemmat. Formalizace je provedena v programovacím jazyce Lean a tento text je doprovázen ukázkami jak originálního kódu, tak někdy výhodnějšího pseudokódu.

Klíčová slova: Łukasiewiczova logika, strojové dokazování, Lean, důkaz úplnosti

Title:

Automated proof-checking of the Rose-Rosser's proof of completeness of Łukasiewicz propositional logic

Author: Bc. Jáchym Šimon

Abstract: This thesis addresses formalisation of the basics of Łukasiewicz logic and automated verification of majority of the first known proof of its completeness, i.e. first verification of 23 lemmas with extensive syntactic proofs, second formalisation of the notion of polynomial formula and with verification of 16 theorems connected to this term. The text describes key moments of the formalisation - mainly basic definitions, general lemmas (e.g. cut rule) and implementation of own tactics for automated proving, e.g. a tactic for automated verification of long syntactic proofs of said lemmas. The formalisation is implemented in the Lean programming language. In this text it is demonstrated by pieces of original code as well as sometimes more useful pseudocode.

Key words: Łukasiewicz logic, automated theorem-proving, Lean, proof of completeness

Contents

Introduction

Mathematical logic uses two approaches to describe its theories: syntactic and semantic. The syntactic approach provides us with a set of axioms and one or more inference rules to examine, using the notion of provability, what formulas can be derived.The semantic approach assigns numerical values (classically interpreted as truth values) to formulas and examines, what formulas can be semantically entailed. A question that immediately arises is: are those approaches equal? Is a formula semantically entailed *iff* it is provable? Equality of these approaches for a given theory is called completeness. The fact that provability implies semantic entailment is called soundness and it is the easier implication to prove. The other implication is called completeness (same as the whole equivalence), and it is, in fact, the complicated one.

The classical semantic approach is to assign one of two values (interpreted as truth and falsehood) to formulas. The first person to work with multi-valued logic instead was Jan Łukasiewicz. His three-valued logic was published in the year 1920 in the publication [3]. Ten years later an article was published by Jan Łukasiewicz and Alfred Tarski discussing logic with truth values covering the real unit interval. However, the first proof of completeness of Łukasiewicz logic was published in 1958, 38 years after its introduction. It was proven by the mathematicians Alan Rose and John Barkley Rosser using the basics of linear algebra and linear programming, and published in [4]. In 2017, Rose-Rosser's proof of completeness was reviewed and translated into modern notation by Petr Fejl in his Bachelor's Thesis [1]. However, since the proof is rather extensive (40 pages) and full of syntactic and tedious lemmas, it is difficult to be sure of its soundness when being proved with classical means (i.e., human proof-checking). For that reason, we decided to verify it using a proof-checking software. We have chosen the Lean theorem prover for this task.

The Lean theorem prover is a programming language based on type theory that is being developed at Microsoft research. The project (launched by Leonardo de Moura in 2013) is hosted publicly on GitHub and it is open source. At the time of writing this thesis the fourth version of this language (Lean 4) is being developed. Mathlib is a mathematical library available for Lean that is being maintained and continuously extended by the active Lean community. It already contains many results from various fields of mathematics.[1] When learning the language, it is possible to ask for an advice on the Lean community forum on Zulip or go through some learning materials available on YouTube [2] and on Lean's official web-page [3].

Lean is trying to bring together interactive and automated theorem proving. As in type theory, proving a proposition corresponds to providing an element (proof term) of a given type (proposition). This can be achieved either by constructing the proof term manually or via the so-called *tactic mode*, which allows writing interactive proofs using automated proving tools and methods. These methods are called tactics and serve as a guide for Lean to follow in order to produce the proof term by itself. This pos-

[1]The guide to install Lean with mathlib can be found at
`https://Leanprover-community.github.io/get_started.html`

[2]YouTube channels *Leanprover community* and *Xena Project*.

[3]All links including community web-site link are available at `https://Leanprover.github.io/`

sibility is highly utilised in the code of our formalisation. In fact, the most tedious and hence difficult to verify part of the Rose-Rosser's proof of completeness for Łukasiewicz propositional logic regards the provability in its calculi (23 pages). In this thesis, among other things, we define custom tactical commands to fully automate and hence simplify the verification of these statements.

This work proves the majority of the proof of completeness of Łukaseiwicz logic in Lean. Specifically, we formalise 23 lemmas dealing with the basic as well as more complex provability in the logic, the notion of *polynomial formula* and finally 16 theorems discussing the provability of polynomial formulas. Left to be formalised is the last section of the proof as written in [1], i.e. Farkas' lemma with two corollaries, two theorems connecting provability and the polynomial representation of formulas, and the completeness theorem itself. The formalisation is done using the Lean theorem prover version 3.39.2. This text should serve as a guide to a possible formalisation of the Łukasiewicz logic and the above-mentioned theorems, as well as a guide to our code, that can be found on the enclosed CD. The code is split into five individual files copying the structure of this text. While original snippets of code from the formalisation are used in the text, some details are left out (i.e. the usage of attributes such as `@[reducible]` and `@[simp]`). For those, the reader is referred to the original code in its full version.

In the first chapter we introduce basic definitions for the *formulas* and *proof* in Łukasiewicz logic (with inspiration from our previous work [2]), followed by two general properties of the *provability relation*: *monotonicity* and the *cut rule* theorem. Using the cut rule, we then define *proof with cut*, which helps us utilise this theorem in formal proofs. After that, we define the semantics of Łukasiewicz logic and prove its *soundness*. Finally, we define the *substitution* of an atom in a formula and prove another property of the provability relation, *structurality*. Further the derived rules of Łukasiewicz logic, which serve as a foundation for the rest of the completeness proof. Furthermore, it introduces the `proof_verifier` tactic for verification of formal proofs. After that, the way `proof_verifer` can verify congruence is discussed, and in the end, the proof by cases theorem is introduced. The second chapter starts with syntactic preparation, i.e. with more complex provability results specific to the completeness proof. In the chapter's second section, we define the notions of *polynomial* and *polynomial formula* and discuss the theorems concerning these terms. The theorems use results from the syntactic preparation section. In the end of the chapter, we demonstrate another utilisation of Lean's automated proving tools, defining a tactic that makes the application of the theorems that concern polynomial formulas easier.

Chapter 1

Łukasiewicz propositional logic

1.1 Elementary syntax and semantics of Łukasiewicz logic

The most basic objects in any propositional logic are *formulas*. Formulas are statements, which are inductively generated from a countably infinite set of elementary statements - *atoms* - using *logical connectives*. The word “elementary” denotes the fact that atoms are not allowed to contain any of the used logical connectives. In Łukasiewicz propositional logic, our basic connectives are *implication* (binary connective) and *negation* (unary connective).

In Lean, a formula can be defined using the so-called inductive type. The resulting definition of a formula is then fairly intuitive, having a total of three constructors (for atoms, for implication and for negation). We use natural numbers to generate atoms.

```
inductive Form : Type
| p : ℕ → Form  -- atoms constructor
| imp : Form → Form → Form  -- implication constructor
| neg : Form → Form  -- negation constructor

local infixr ` ⇒ ` : 80 := Form.imp
local prefix ` ~ ` : 100 := Form.neg
```

Thanks to the `infixr` and `prefix` keywords we can set the binding strength of implication and negation constructors, as well as a more natural way of notation. Now, given formulas `P` and `Q`, we can type `P ⇒ Q` and `~P` instead of `Form.imp P Q` and `Form.neg P`, respectively. In addition, we will define five derived logical connectives, that are based on implication and negation. Namely, we will define so-called disjunction, strong disjunction, conjunction, strong conjunction and equivalence, in the given order. The connectives are defined using lambda construction. Using this construction we can define functions, in this case of type `Form → Form → Form`, and state explicitly what is to be returned. We will use this construction a bit more when defining polynomials in the section 2.2.

```
local infix ` ⊔ ` : 90 := λ P Q: Form, (P ⇒ Q) ⇒ Q
local infix ` ⊕ ` : 90 := λ P Q: Form, ~P ⇒ Q
local infix ` ⊓ ` : 91 := λ P Q: Form, ~(~P ⊔ ~Q)
local infix ` & ` : 91 := λ P Q: Form, ~(P ⇒ ~Q)
local infix ` ⇔ ` : 70 := λ P Q: Form, (P ⇒ Q) ⊓ (Q ⇒ P)
```

We use capital letters $P, Q, \ldots$ to denote formulas. Using the command `variables` we can introduce these formulas and, whenever they are used in a statement, they will be automatically universally quantified over. Similarly, we introduce variables for sets of formulas which, in type theory, are defined as functions of type Form → Prop.

```
variables {P Q R : Form}
variables {Γ Δ Θ : set Form}
```

Formal proof

The notion of proof in Łukasiewicz propositional logic is given by five axioms and one rule of inference, *modus ponens*, which gives us means to prove new formulas from the five axioms. Modus ponens can be symbolically written as `(P, P ⇒ Q) -> Q`, i.e. if `P` and `P ⇒ Q` are provable, then `Q` is provable. The five axioms go as follows.

```
def A1 (P Q:Form) := P ⇒ (Q ⇒ P)
def A2 (P Q R:Form) := (P ⇒ Q) ⇒ ((Q ⇒ R) ⇒ (P ⇒ R))
def A3 (P Q:Form) := (P ⊔ Q) ⇒ (Q ⊔ P)
def A4 (P Q:Form) := (~P ⇒ ~Q) ⇒ (Q ⇒ P)
def A5 (P Q:Form) := (P ⇒ Q) ⊔ (Q ⇒ P)
```

Having our five axioms and the modus ponens rule we can now define the notion proof (also called *formal proof*). Suppose we have a set of formulas Γ (called *set of assumptions*) and a formula P. A proof of P from Γ is a sequence of formulas ending with P, where each formula is either 1) an instance of an axiom, 2) belongs to Γ, or 3) is derived using modus ponens from two formulas preceding it somewhere in the sequence. We also say that a *rule* (Γ, P) is provable in Łukasiewicz logic (or simply in Ł) if there is a proof of P from Γ. Formulas provable from an empty set of assumptions are sometimes called *theorems* of Ł.

Naturally, the proof sequence can be represented as a binary tree, where leaves are designated with axiom instances/assumptions from Γ while nodes with two children are designated with consequence of modus ponens rule.

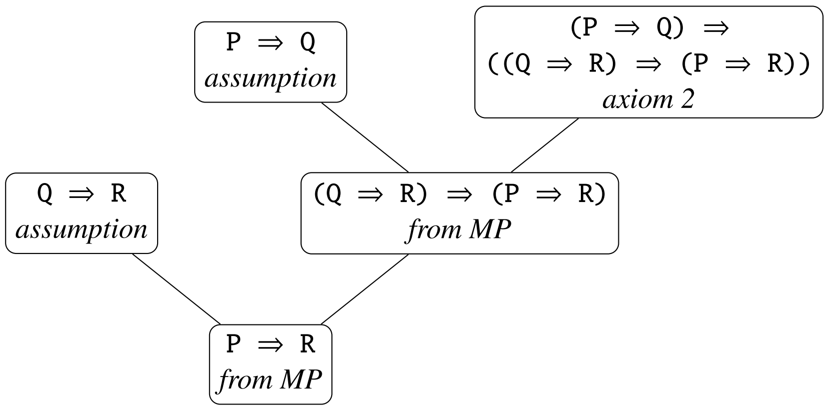

Figure 1.1: Proof of `P ⇒ R` from assumptions `P ⇒ Q` and `Q ⇒ R`.

Note that first this enforces that any node in the tree is derived directly from the roots of its children (helping with simplicity of verification), and second in this definition, no nodes with one child are used. That is, of course, because modus ponens - the only rule of inference - is binary. However, we will make use of nodes with one child in the following paragraph, when extending the definition of proof to enable the use of already proven theorems and rules of Ł.

Before formalising the proof in Lean, we define a binary tree. Again, we use the inductive type for the definition and introduce bindings and symbols for the tree's constructors.

```
inductive tree_of_form : Type
| leaf : Form → tree_of_form -- leaf constructor
| un : tree_of_form → Form → tree_of_form -- node, one child
-- node, two children
| bin : (tree_of_form × tree_of_form) → Form → tree_of_form

local postfix ` . ` : 40 := tree_of_form.leaf
local infixr ` ▶ ` : 30 := tree_of_form.un
local infixr ` ▶▶ ` : 20 := tree_of_form.bin

/- prod.mk given two trees t1 and t2 outputs a pair
   (t1, t2) of type (tree × tree), which is used above
   in the tree_of_form.bin constructor. -/
local infixr ` && ` : 25 := prod.mk
```

As an example of our notation, this is the tree from figure 1.1 written in Lean.

```
    Q ⇒ R.  &&  (P ⇒ Q. && A2 P Q R. ▶▶ (Q ⇒ R) ⇒ (P ⇒ R))  ▶▶  P ⇒ R
```

It is also useful to define a function TR of type `tree_of_form → Form` that returns the root of a tree. Here we use a special definition, which takes advantage of the inductive definition of `tree_of_form`. We further use defined symbols for `tree_of_form` constructors.

```
@[simp] def TR: tree_of_form → Form
| (P.) := P
| (pr ▶ P) := P
| (pr1 && pr2 ▶▶ P) := P
```

Now we can formalise the definition of proof. First we define `is_axiom`, which is a set (i.e. type `Form → Prop`) that for a given formula states, whether it is an instance of one of our five axioms. Then we define what it means for some tree to be a formal proof from a set of assumptions and, finally, we define *provability*, which is a statement saying that there exists a proof of some given formula from a given set of assumptions.

```
def is_axiom : set Form :=
λ P, (∃ Q R, P = A1 Q R) ∨
     (∃ Q R D, P = A2 Q R D) ∨
     (∃ Q R, P = A3 Q R) ∨
     (∃ Q R, P = A4 Q R) ∨
     (∃ Q R, P = A5 Q R)

/- We again take advantage of the inductive definition of tree_of_form. The
  last constructor is describing the usage of modus ponens. -/
def is_proof : set Form → tree_of_form → Prop
| Γ (P.) := is_axiom P ∨ P ∈ Γ
| Γ (t ▶ P) := false  -- unused tree_of_form.un constructor
| Γ (t1 && t2 ▶▶ P) := (TR t2 = TR t1 ⇒ P) ∧
                       is_proof Γ t1 ∧
                       is_proof Γ t2

def is_provable (Γ : set Form)(P : Form) : Prop :=
∃ pr, is_proof Γ pr ∧ TR pr = P

infixr ` ⊢ ` : 40 := is_provable
prefix ` ⊢ ` : 40 := is_provable {}
```

Looking closely into the third constructor of `is_proof`, it becomes apparent that it enforces a certain order of its children. When deriving `Q` from `P` and `P ⇒ Q`, we need to have `P` as root of the left child and `P ⇒ Q` as root of the right one. That is quite inconvenient, but keeps the definition simple, which is very useful when arguing about the provability relation. On the other hand, we do not want to be bound to such limitations, so in the section 1.2 this issue will be addressed.

Finally, let us mention one basic property of the provability relation, which is *monotonicity*. It is a simple, yet important, statement that follows directly from the definition of formal proof.

```
lemma monotonicity : Γ ⊢ P → Γ ⊆ Δ → Δ ⊢ P := sorry
```

The cut rule and proof with cut

As mentioned earlier, when writing out proofs, we would like to have the ability to use rules and theorems of Łalready proven. This ability is granted by the *cut rule*. The rough idea of the cut rule is simple: assume we have a proof `prf`. Wherever in `prf` we would use some already proven formula `P` (therefore breaking the formal definition of proof), we can instead insert the whole proof of `P` into `prf`. The `prf` extended with the formal proof of `P` will then follow the formal proof definition.

Below is shown how the cut rule looks in our formalisation. We are mainly interested in two special cases: `binary_cut` and `unary_cut`.

```
def symmetric_provability (Γ : set Form) (Δ : set Form)
: Prop := ∀ P ∈ Δ, Γ ⊢ P

local infixr ` ⊩ ` : 40 := symmetric_provability
local prefix ` ⊩ ` : 40 := symmetric_provability {}

theorem cut_rule: Γ ⊩ Θ → Δ ∪ Θ ⊢ Q → Δ ∪ Γ ⊢ Q := sorry

lemma binary_cut : Γ ⊢ P → Γ ⊢ Q → {P, Q} ⊢ R → Γ ⊢ R := sorry

lemma unary_cut : {P} ⊢ Q → Γ ⊢ P → Γ ⊢ Q := sorry
```

The `cut_rule` theorem can be proven using the following lemma, whose proof copies the exact steps described above.

```
lemma cut_rule_help (hp : Γ ⊩ Θ)
: ∀ (pt : tree_of_form), is_proof (Δ ∪ Θ) pt → Δ ∪ Γ ⊢ TR pt :=
λ t, match t with
| (Q.) := /- If Q is an axiom- follows from monotonicity. If Q is assumption
  then
              1) Q ∈ Δ - trivial or 2) Q ∈ Θ - follows from hp. -/
| (pt ▶ Q) := /- Such proof can not exist. -/
| (t1 && t2 ▶▶ Q) := /- From inductive hypothesis and modus ponens. -/
end
```

On paper, this theorem would be enough and no other constructions would be needed, because there we can just use the cut rule silently without any explicit justification. In Lean, however, we have to either explicitly state it, whenever we are using the cut rule (which is often), or as we did, create an extension of the formal proof that will include the use of the cut rule in its very definition and therefore justify its usage only once.

But how to extend the proof definition? As the proof is represented by a binary tree, the extension will be, too. We can split the provability rules into four categories based on the number of assumptions: `⊢ P`, `{P} ⊢ Q`, `{P, Q} ⊢ R`, and more than two assumptions. It is already apparent how to fit those into the tree representation. All we need to do, is to allow theorems into the `tree_of_form.leaf` constructor, one assumption rules into the previously unused `tree_of_form.un` constructor and two assumption rules into the `tree_of_form.bin` constructor. The apparent downside of this representation is that we will not be able to use rules with three and more assumptions. However, since this is not of relevance to our application, therefore we do not mind that.

Thanks to `MP` and `axiom_is_provable` we can define the extension using provability such that the definition is very similar to the original one. Note, that first the definition still requires each formula to be derived directly from its children and second the definition has a similar issue as `is_proof` regarding the order of children. More about that in the next section.

```
lemma axiom_is_provable : is_axiom P → Γ ⊢ P := sorry

lemma MP : {P, P ⇒ Q} ⊢ Q := sorry

def is_proof_cut: set Form → tree_of_form → Prop
| Γ (P.):= P ∈ Γ ∨ ⊢ P
| Γ (pr ▶ P):= {TR pr} ⊢ P ∧ is_proof_cut Γ pr
| Γ (t1 && t2 ▶▶ P) := {TR t1, TR t2} ⊢ P ∧
                        is_proof_cut Γ t1 ∧
                        is_proof_cut Γ t2

def is_provable_cut (Γ: set Form)(P: Form) : Prop :=
∃ pr, is_proof_cut Γ pr ∧ TR pr = P

infixr ` ⊢cut ` : 40 := is_provable_cut
prefix ` ⊢cut ` : 40 := is_provable_cut {}
```

To justify our extension, we show, that both definitions are equal, i.e. for any P and Γ is `Γ ⊢cut P ↔ Γ ⊢ P`. In this case, we do not actually need the whole equivalence to work with proofs with cut, but theorem `provable_no_cut` suffices. In order to prove `Γ ⊢ P` (which is always our goal), we can prove `Γ ⊢cut P` instead and the goal will follow from `provable_no_cut`. Thanks to the `binary_cut` and `unary_cut` lemmas the proof of the theorem is easy.

```
theorem provable_no_cut : Γ ⊢cut P → Γ ⊢ P :=
  have lem: ∀ t, is_proof_cut Γ t → Γ ⊢ TR t,
  from λ t, match t with
  | Q. := /- Proof consisting of only Q did not use cut. -/
  | t ▶ Q := /- From ind. hypothesis. and unary_cut. -/
  | t1 && t2 ▶▶ Q := /- From ind. hyp. and binary_cut. -/
by{intro a, cases a with t h, convert lem t h.1, simp[h.2]}
```

Semantics and proof of soundness

Opposing the syntax side of the logic, stands the semantics. In semantics we discuss the truthfulness of formulas and we do so using *truth evaluation*. Truth evaluation is a function that gives *truth values* to individual atoms. Using predefined interaction rules between truth values and logical connectives, we can then derive the truth value of any formula. While in classical propositional logic the truth values can only be from $\{0, 1\}$ (i.e. false or true), in Łukasiewicz propositional logic we allow any rational number between 0 and 1. That gives us an infinite amount of possible truth values. Of course, the interaction rules for logical connectives need to be defined accordingly, so that the truth value of any formula stays between 0 and 1.

Special role among all formulas play those, whose truth value is equal to 1 under any truth evaluation. These formulas are called *tautologies*. If a formula has truth value 1 every time all formulas from Γ have truth value 1, then it is a *semantic consequence* of Γ.

We start with a definition of the truth evaluation on atoms (`eval` definition), and then extend it by stating the rules of interaction with logical connectives (`val` definition). Note lemma `v_0_1`, that shows that the extension is valid, i.e. the truth value of any formula really stays between 0 and 1. In the end, we define tautology and semantic consequence.

```
/- Rational closed interval <0, 1>. Would not be needed, but looks nicer. -/
def UnitQ := (@Icc ℚ _ 0 1)
/- Evaluation on atoms. -/
def eval := ℕ → UnitQ
/- Extended evaluation. -/
def val (v : eval): Form → ℚ
| (p n) := v n
| (A ⇒ B) := min (1:ℚ) (1 - (val A) + (val B))
| ~A := (1:ℚ) - (val A)

postfix ` ∗ ` : 100 := val

lemma v_0_1 : ∀ v: eval, ∀ P: Form, v∗ P ∈ UnitQ := sorry

def is_taut (A:Form) : Prop := ∀ {v:eval}, v∗ A = (1:ℚ)

def is_sem_conseq (Γ: set Form)(Q: Form) : Prop :=
∀ v: eval, (∀ P ∈ Γ, v∗ P = 1) → (v∗ Q = (1:ℚ))

prefix ` ⊨ ` : 40 := is_taut
infix ` ⊨ ` : 40 := is_sem_conseq
```

So far, the connection between syntax and semantics has not yet been established. It would be nice to know though, that there are some restrictions as to what is provable with regards to the semantics. It would not be a good syntactic system if we could prove formulas that are semantically false. Thus we will now prove the soundness of our logic.

```
lemma soundness_empty: ⊢ Q → ⊨ Q := sorry
```

The proof of `soundness_empty` is based on the fact that our five axioms are tautologies and that modus ponens, as the only rule of inference, preserves this property. Therefore, building proofs with only axioms then can not lead to anything else but a tautology. This is generalised in the `soundness` lemma using a similar logic with the notions of provability from assumptions and semantic consequence.

```
lemma soundness: Γ ⊢ Q → Γ ⊨ Q := sorry
```

Thus, the first step in proving the `soundness` is to verify that all of the five axioms are tautologies. We will show how that is done, as it is a nice example of the usage of Lean's automated proving tools. These proofs are technical, based on going through all the options of evaluations of involved formulas. For example, proving that `A1` is a tautology, we first unfold the definition of evaluation `val`

```
v∗(A1 P Q) = v∗(P ⇒ (Q ⇒ P)) = min 1 (1 - v∗P + min 1 (1 - v∗Q + v∗P))
```

and then discuss the options `v∗ P ≥ v∗ Q` and `v∗ P < v∗ Q` while from `v_0_1` we know, these values are between 0 and 1. This gets more complicated, when three formulas are involved (as and only as in the case of the second axiom).

The process can be generalised using Lean's automated proving tools (above-mentioned *tactics*). Our tactic `ax_taut` is able to verify that all our axioms are tautologies. The general strategy of the tactic is the following: the tactic unfolds the definition `val` and uses the lemma `v_0_1` to get restrictions on the truth values. The goal is the split using the definitions of `min` and `max` as `if`... `then`... `else`... statements. Remaining inequalities can be solved by Lean's tactic `linarith` (short from linear arithmetic).

```
meta def ax_taut: tactic unit :=
`[intros, unfold1 val, simp,
  have a: v*P ≥ 0 ∧ v*P ≤ 1, exact (v_0_1 v P),
  have b: v*Q ≥ 0 ∧ v*Q ≤ 1, exact (v_0_1 v Q),
  try{have c: v*R ≥ 0 ∧ v*R ≤ 1, exact (v_0_1 v R)},
  simp[*, max_def, min_def],
  split_ifs, all_goals{linarith}]

variable {v: eval}

lemma A1_taut : ∀ P Q, v*(A1 P Q) = 1 := by ax_taut
lemma A2_taut : ∀ P Q R, v*(A2 P Q R) = 1 := by ax_taut
lemma A3_taut : ∀ P Q, v*(A3 P Q) = 1 := by ax_taut
lemma A4_taut : ∀ P Q, v*(A4 P Q) = 1 := by ax_taut
lemma A5_taut : ∀ P Q, v*(A5 P Q) = 1 := by ax_taut
```

These lemmas can be put together into the lemma `axiom_taut`. We can now prove the helping lemma `sound_help` and the `soundness` itself.

```
lemma axiom_taut: ∀ P, is_axiom P → (v* P = 1) := sorry

/- Help lemma proven by induction over tree_of_form. -/
lemma sound_help (hΓ: ∀ (P: Form), P ∈ Γ → (v* P = 1))
: ∀ (prf: tree_of_form), is_proof Γ prf → (v* (TR prf) = 1)
| (Q.) :=
  begin
    /- Q is an axiom. Follows from axiom_taut. -/
  end
| (pt ▶ Q) :=
  begin
    /- Unused constructor that returns false. Anything follows from false. -/
  end
| (t1 && t2 ▶▶ Q) :=
  begin
    have t1_taut: v* (TR t1) = 1, -- from induction hypothesis
    have t2_taut: v* (TR t1 ⇒ Q) = 1, -- from ind. hyp.
    /-
    From t2_taut we have
      v* (TR t1 ⇒ Q) =  min 1 (1 - v* (TR t1) + v* Q) =
        = min 1 (1 - 1 + v* Q) = min 1 (v* Q) = 1
    and since v* Q ≤ 1 we have v* Q = 1.
    Finally Q = TR (t1 && t2 ▶▶ Q).
    -/
  end

lemma soundness: Γ ⊢ Q → Γ ⊨ Q :=
begin
  intros prov v hΓ,
  rcases prov with ⟨prf, ⟨h1, h2⟩⟩,
  rw ←h2,
  /- Current state (without variables) :
  hΓ: ∀ (P : Form), P ∈ Γ → v* P = 1
  h1: is_proof Γ prf
  h2: TR prf = Q
  ⊢ v* (TR prf) = 1 -/
  exact correct_help hΓ prf h1,
end
```

Substitution and structurality

The last thing left to define is the substitution. Substitution is a function that takes formula P, some atom x, and formula Q, and substitutes all occurrences of x in P with Q. For this, we make use of the inductive definition of Form and the if...then...else... construction to introduce a simple definition of substitution (p is the atom constructor of Form).

```
  def subst (n: ℕ)(Q: Form): Form → Form
  | (p m) := if (m = n) then Q else p m
  | (A ⇒ B) := subst A ⇒ subst B
  | (~A) := ~ subst A

  notation R `[` n `,` Q `]` := subst n Q R
```

Here is an example of how such substitution looks, in this case swapping all occurrences of atom p 1 for formula P.

```
  example : ((p 1 ⇒ p 2) ⊕ ~(p 1))[1, P] = (P ⇒ p 2) ⊕ ~P := by simp
```

Useful lemma connected to substitution is *structurality*, which states that provability is in some sense invariant under substitution. Before stating the lemma, we define substitution on a set of assumptions.

```
  def subst_set (n : ℕ) (Q : Form): set Form → set Form :=
  λ Γ, {P | ∃ R ∈ Γ, P = R[n, Q]}

  notation Γ `[` n `,` Q `]` := subst_set n Q Γ

  lemma structurality: Γ ⊢ P → Γ[n, Q] ⊢ P[n, Q] := sorry
```

The idea of the proof of structurality is the following: we take the formal proof of Γ ⊢ P and show, that if we do a substitution of Q for p n in the whole proof sequence and the assumptions, the property of being a formal proof will be preserved. In order to do this we introduce one helping definition and three lemmas. First lemma states that if we do a substitution in an axiom, the resulting formula will still be an axiom.

```
  lemma axiom_structurality {n : ℕ}
  : is_axiom P → is_axiom (P [n, Q]) :=
  begin
  /- Thanks to the subst definition we have the following:
  if  P = A1 R S = (R ⇒ S ⇒ R)
  then  P[n, Q] = (R ⇒ S ⇒ R)[n, Q] =
        (R[n, Q] ⇒ S[n, Q] ⇒ R[n, Q]) =
        A1 (R[n, Q]) (S[n, Q]).
  Similarly for the other axioms. -/
  end
```

For the remaining two lemmas, we need to define a substitution on a proof, which is a substitution on all formulas in the proof sequence.

```
def subst_prf (n : ℕ) (Q : Form) : tree_of_form → tree_of_form
| (P.) := P[n, Q].
| (t ▶ P) := (subst_prf t) ▶ P[n, Q]
| (t1 && t2 ▶▶ P) := (subst_prf t1) && (subst_prf t2) ▶▶ P[n, Q]
```

Now that we can perform substitution on a set of assumptions (`subst_set`) and on a formal proof, we can prove the other two helping lemmas. Lemma `subst_TR` is self-explanatory and easy to prove. Finally, lemma `subst_preserve_proof` shows that the formal proof is preserved under substitution. Proof of this lemma is also not complicated.

```
lemma subst_TR
: ∀ t : tree_of_form, TR (t[n,Q]) = (TR t)[n,Q] := sorry

lemma subst_preserve_proof
: ∀ t : tree_of_form, is_proof Γ t → is_proof (Γ[n, Q]) (t[n, Q])
| (P.) := /- From axiom_structurality and subst_set definition. -/
| (t ▶ P) := /- Such proof can not exist. -/
| (t1 && t2 ▶▶ P) := /- From induction hypothesis and subst_TR. -/
```

Having these results we can then easily prove the `structurality` lemma.

1.2 Derived rules of Łukasiewicz logic and proof by cases

Having introduced the logic as a whole and having shown some useful general theorems, as well as the soundness, we can finally proceed to the preparation of the completeness theorem. In this section we will be discussing derived rules and theorems of Łukasiewicz logic, i.e. some basic results about the provability relation. In the formalisation, these are lemmas 14 to 33 and they serve as a foundation for more complex results about the provability relation in the section 2.1.

First, we will show an example of a derived rule and introduce our tactic for formal proof verification. Second, we will teach Lean's tactic `simp` that equivalence is a congruence relation in Łukasiewicz logic. In other words, we make our tactic capable of recognising that an inference was used, that can be symbolically written as `{P ⇔ Q} ⊢ R(P) ⇔ R(Q)`. Lastly, we will mention proof by cases theorem, that requires some derived rules to be proven.

Derived rules and `proof_verifier` tactic

A simple example of a derived rule can be the very first one, i.e. `lemma14_1`. In general, each rule consists of its statement, in this case `{P ⇒ Q, Q ⇒ R} ⊢ P ⇒ R`, its formal proof always introduced as `prf`, and a Lean proof verifying that `prf` is indeed a formal proof of a given rule. This verification is provided by our `proof_verifier` tactic. While the formal proof of `lemma14_1` is short and simple, that is not true for all rules.

```
@[simp] lemma lemma14_1 : {P ⇒ Q, Q ⇒ R} ⊢ P ⇒ R :=
  let prf := Q ⇒ R. &&
             (P ⇒ Q. && A2 P Q R. ▶▶ (Q ⇒ R) ⇒ (P ⇒ R))
              ▶▶ P ⇒ R
  in by proof_verifier
```

Let us discuss the `proof_verifier`. In general, there are two main principles we are using when writing out formal proofs: there are the cut rules discussed in the section 1.1 and congruence, i.e. if two formulas are equivalent (i.e. `⊢ P ⇔ Q`), then they are interchangeable in any formula. The `proof_verifier` is able to deal with both. In fact, the latter one is a special case of the first. We will start with building the tactic for verification of proofs with cut, using parts of `lemma14_1` as an example. Note, that this lemma does not technically include any usage of cut. Thanks to the lemma `MP` and theorem `provable_no_cut` from the section 1.1, we can perceive any proof as a proof with cut. Once again, the definition of `is_proof_cut`:

```
@[simp] def is_proof_cut: set Form → tree_of_form → Prop
| Γ (P.):= P ∈ Γ ∨ ⊢ P
| Γ (pr ▶ P):= {TR pr} ⊢ P ∧ is_proof_cut Γ pr
| Γ (t1 && t2 ▶▶ P) := {TR t1, TR t2} ⊢ P ∧
                       is_proof_cut Γ t1 ∧
                       is_proof_cut Γ t2

def is_provable_cut (Γ: set Form)(P: Form) : Prop :=
∃ pr, is_proof_cut Γ pr ∧ TR pr = P
```

Each formula in a proof with cut can either be an assumption, a theorem of Łor an application of an already proven derived rule of Łwith one or two assumptions (as mentioned, this includes application of modus ponens). The job of `proof_verifier` is to separately verify all these steps in given proof, i.e. verify the origin of all the individual formulas.

Two things playing a major role in the verification are the tree representation of a proof and Lean's inbuilt tactic `simp` (or simplifier). Firstly, thanks to the tree representation, each formula in a proof is derived directly from its children (except of the leaf formulas, naturally). In addition, the position of a formula in a proof tree immediately reveals its supposed origin in the proof. For example, having a proof of shape `(t1 && t2 ▶▶ P)`, we know P is a result of an application of some rule with two assumptions of the shape `{TR t1, TR t2} ⊢ P`. So in order to verify `P`, we need to check whether such a rule is provable (i.e. whether we already proved it). Since we know the formulas `TR t1`, `TR t2` and `P`, this is easy, because it can be done by the already mentioned `simp` tactic. In the same way, having a proof of shape `(P.)`, we know that P is either an assumption or an application of a rule of shape `⊢ P`. Again, both options (in fact this disjunction as a whole) can be checked by `simp`.

We here use `lemma14_1` as an example. The formal proof of this lemma consists of five steps, each step corresponding to one example below. All of these examples, i.e. all individual steps in the proof, can be verified by `simp` (in this case making use of `MP` and `A2_provable`).

```
/- Modus ponens as a derived rule. -/
@[simp] lemma MP : {P, P ⇒ Q} ⊢ Q := sorry

/- Axiom 2 is provable.  A2 P Q R = (P ⇒ Q) ⇒ ((Q ⇒ R) ⇒ (P ⇒ R)) -/
@[simp] lemma A2_provable: ⊢ A2 P Q R := sorry

/- The formula Q ⇒ R is a leaf in the proof. It is an assumption. -/
example: Q ⇒ R ∈ ({P ⇒ Q, Q ⇒ R} : set Form) ∨ ⊢ Q ⇒ R := by simp

/- The formula Q ⇒ R too. -/
example: P ⇒ Q ∈ ({P ⇒ Q, Q ⇒ R}: set Form) ∨ ⊢ P ⇒ Q := by simp

/- The formula A1 P Q R is a leaf. It is a direct application of
  A2_provable. -/
example : A2 P Q R ∈ ({P ⇒ Q, Q ⇒ R}: set Form) ∨ ⊢ A2 P Q R := by simp
```

```
/- The formula (Q ⇒ R) ⇒ (P ⇒ R) comes from an application of MP. -/
example : {P ⇒ Q, A2 P Q R} ⊢ (Q ⇒ R) ⇒ (P ⇒ R) := by simp

/- The formula P ⇒ R too. -/
example: {Q ⇒ R, (Q ⇒ R) ⇒ (P ⇒ R)} ⊢ P ⇒ R := by simp
```

The simplifier is a powerful tactic. It tries to simplify (or even close if possible) the current goal using a list of various definitions, lemmas and theorems. By using the tag `@[simp]` we can broaden this list (other option is to give `simp` things in square brackets, like this `simp[A2_provable]`). All derived rules in our code are tagged for `simp`. Overall, `simp` can verify all individual steps in a proof on its own. We can, however, make `simp` even stronger by tagging the definitions of tree root `TR` and `is_proof_cut`, too. Having these definitions, the tactic is able to deconstruct the proof and verify all individual steps. As a result, we can prove `lemma14_1` with three commands.

```
@[simp] lemma lemma14_1 : {P ⇒ Q, Q ⇒ R} ⊢ P ⇒ R :=
  let prf := Q ⇒ R. &&
            (P ⇒ Q. && A2 P Q R. ▶▶ (Q ⇒ R) ⇒ (P ⇒ R))
             ▶▶ P ⇒ R
  in by {
  /- We will verify provability with cut. -/
  apply provable_no_cut,
  /- prf is the formal proof. -/
  existsi prf,
  /- Current goal:
    (is_proof_cut {P ⇒ Q, Q ⇒ R} prf) ∧ (TR prf = P ⇒ R).
    Can be solved by simp. -/
  simp}
```

The last thing we have to pay attention to, is the ordering of children in a proof tree. As we mentioned when introducing the proof with cut in the section 1.1, the definition `is_proof_cut` requires certain ordering of children in its tree representation. To be specific, if a formula `P` is derived from a lemma `{Q, R} ⊢ P`, it is required that `Q` is the left child and `R` is the right child of the node with `P`. This can be overcome by introducing a simple lemma `equal_assumptions`.

```
lemma equal_assumptions: {P, Q} ⊢ R → {Q, P} ⊢ R := sorry
```

Now, if `simp` does not manage to verify some formula in a proof because of the ordering, we can use this lemma (effectively swapping the assumptions) and let `simp` try again.

So far `simp` can only apply rules and lemmas tagged with `@[simp]`. Sometimes however, while proving a rule we need to introduce some additional mid-proof statements and we want `simp` to use them. For those situations we add a star symbol `simp *`, that allows `simp` to use all hypotheses available while proving a given goal. The `proof_verifier` then looks like this:

```
meta def proof_verifier : tactic unit :=
`[/- We will verify the goal as provable with cut. -/
 apply provable_no_cut,
 /- Tell Lean prf is the proof. -/
 existsi prf, split,
 /- Let simp solve all the goals it can. -/
 all_goals{simp *},
 /- Swap assumptions of all remaining goals and let simp try again. -/
 repeat{solve1{apply equal_assumptions, by simp *} <|> split}]
```

`proof_verifier` meets congruence

As we have shown that the `proof_verifier` is able to verify proofs with cut, let us discuss the congruence (or substitution). We do not actually have to change anything about the `proof_verifier` tactic to make it able to verify congruence. In this section, we will teach `simp` how to deal with congruence by supplying it with some more lemmas.

An example of congruence could be the following statement `{~~P ⊔ ~~Q} ⊢ P ⊔ Q`, where using lemma `lemma23_poml`, the exchange of `~~P` and `~~Q` for `P` and `Q` is performed. How does `simp` verify this example will be discussed below.

```
@[simp] lemma lemma23_poml: ⊢ (~~P ⇔ P) := sorry

example: {~~P ⊔ ~~Q} ⊢ P ⊔ Q := by simp
```

The verification of congruence is based on so-called *congruence lemmas*. These lemmas show that such exchange is at all possible under our logical connectives.

```
@[simp] lemma congr_imp_left: Γ ⊢ P ⇔ Q → Γ ⊢ (P ⇒ S) ⇔ (Q ⇒ S) := sorry

@[simp] lemma congr_imp_right: Γ ⊢ P ⇔ Q → Γ ⊢ (S ⇒ P) ⇔ (S ⇒ Q) := sorry

@[simp] lemma congr_imp_prem:
Γ ⊢ P ⇔ Q → Γ ⊢ S ⇔ R → Γ ⊢ (S ⇒ P) ⇔ (R ⇒ Q) := sorry

@[simp] lemma congr_neg: Γ ⊢ (P ⇔ Q) → Γ ⊢ ~P ⇔ ~Q := sorry
```

Note, that using these lemmas we could prove more general statement: having formulas P, Q and R we can swap all occurrences of P in R with Q, which could be written as `{P ⇔ Q} ⊢ R(P) ⇔ R(Q)`. Although this is useful to justify congruence on paper, it is a bit too general for our automated verification. The problem is, that the general statement does not give Lean many hints as to where exactly the substitution happened and how to verify it, which leads to a lot of inefficiency. Using these four lemmas directly means that `simp` can deconstruct involved formulas (effectively performing the proof of the general statement, if applied on given formulas) and verify the congruence.

In the code, we have a few helping lemmas, that are aiding with the verification of congruence and its efficiency. One of those lemmas is `MP_ekv1`, which is directly used in the verification of the above example `{~~P ⊔ ~~Q} ⊢ P ⊔ Q`.

```
@[simp] lemma MP_ekv1: ⊢ (P ⇔ Q) → {P} ⊢ Q := sorry
```

Given a goal of form `{P} ⊢ Q`, where `Q` is derived from `P` using substitution, `simp` applies `MP_ekv1` and attempts to solve a goal of form `⊢ (P ⇔ Q)` instead. In our example, it amounts to `⊢ (~~P ⊔ ~~Q) ⇔ (P ⊔ Q)`. This goal can be then further simplified using congruence lemmas as demonstrated below.

We will now deconstruct the individual steps `simp` does in order to verify our example. It is only making use of the `congr_imp_prem` lemma, but it is easy to imagine that the usage of the remaining congruence lemmas would be similar. We can use the command `set_option trace.simplify.rewrite true`, which makes Lean print all successful applications of lemmas, theorems and definitions `simp` executed in a given occasion. This is a summary of what is printed for our example. (Note, that `P ⊔ Q = (P ⇒ Q) ⇒ Q` for any `P` and `Q`.)

```
GOAL:  {(~~P ⇒ ~~Q) ⇒ ~~Q} ⊢ (P ⇒ Q) ⇒ Q
-> simp applies MP_ekv1
GOAL:  ∅ ⊢ ((~~P ⇒ ~~Q) ⇒ ~~Q) ⇔ ((P ⇒ Q) ⇒ Q)
-> simp applies congr_imp_prem

   GOAL 1:  ∅ ⊢ (~~P ⇒ ~~Q) ⇔ (P ⇒ Q)
   -> simp applies congr_imp_prem
      GOAL 1a:  ∅ ⊢ ~~P ⇔ P
      -> simp applies lemma23_poml
      GOAL 1b:  ∅ ⊢ ~~Q ⇔ Q
      -> simp applies lemma23_poml

   GOAL 2:  ∅ ⊢ ~~Q ⇔ Q
   -> simp applies lemma23_poml
```

An example of one more helping lemma can be `congr_bin`.

```
@[simp] lemma congr_bin: {P ⇒ Q, Q ⇒ P} ⊢ S ⇔ R → {P ⇔ Q, S} ⊢ R := sorry
```

Here, the substitution is happening between formulas `S` and `R`. Specifically, we are either exchanging `P` for `Q`, or `Q` for `P`, in either `S` or `R`. None of this has to be specified, thanks to the equivalency `P ⇔ Q` being split into two implications and the `equal_assumptions` lemma. `congr_bin` is very similar to the `MP_ekv1`, but unlike `MP_ekv1`, it allows us to use also equivalence formulas, that were proven within the same formal proof.

Note that the form of the rules introduced in this section is not arbitrary, but was carefully selected to optimise the time required for the proof to be found. E.g., there are different simp lemmas to be used for theorems, unary and binary rules. Finally, we note that the order of lemmas is important as well, since `simp` first tries to use the lemmas written later in the code (so lemmas without assumptions are used first).

Proof by cases

After proving some beginning results, namely `lemma14_3_pom3`, `lemma14_5_pom3_rule` and `lemma15rule`, we can prove the proof by cases theorem. This theorem becomes quite important in the course of completeness proof. The proof of proof by cases is quite technical and arguably not too interesting, therefore it is not described here. The interested reader is referred to the full code.

```
lemma lemma14_3_pom3 : {P} ⊢ (P ⊔ Q) := sorry

lemma lemma14_5_pom3_rule : {P ⊔ P} ⊢ P := sorry

lemma lemma15rule : {P ⊔ Q, (P ⇒ R) ⊔ Q} ⊢ R ⊔ Q := sorry

theorem proof_by_cases (hp : Γ ∪ {R} ⊢ T) (hq: Δ ∪ {S} ⊢ T)
: Δ ∪ Γ ∪ {R ⊔ S} ⊢ T := sorry
```

Chapter 2

Proof of completeness

2.1 Syntactic preparation

The syntactic preparation consists of theorems 34 to 40. Those are complex statements about the provability relation not too different from the derived rules of Łukasiewicz logic in the previous section. However, compared to those rules, these theorems are longer, more complex and most have additional hypotheses. An example of such a theorem can be `theorem36`.

```
theorem theorem36
(h1: ⊢ ~V ⊔ W)
(h2: ⊢ R ⇔ ((V ⊕ Z) & W))
(h3: ⊢ (((W ⊕ Z) & X)) ⇔ S)
(h4: ⊢ T ⇔ ((V ⊕ Y) & W))
(h5: ⊢ (((W ⊕ Y) & X)) ⇔ U ) :
⊢ ((R ⊕ Y) & S) ⇔ ((T ⊕ Z) & U) :=
have a: ⊢ (R ⊕ Y) & (W ⊕ Z) ⇔ (T ⊕ Z) & (W ⊕ Y), from lemma35 h2 h4 h1,
let prf := (((R ⊕ Y) & (W ⊕ Z) ⇔ (T ⊕ Z) & (W ⊕ Y). ▶
          (((R ⊕ Y) & (W ⊕ Z) & X) ⇔ ((T ⊕ Z) & (W ⊕ Y) & X))) ▶
          (((R ⊕ Y) & ((W ⊕ Z) & X)) ⇔ ((T ⊕ Z) & ((W ⊕ Y) & X)))) ▶
          (((R ⊕ Y) & S) ⇔ ((T ⊕ Z) & U))
in by proof_verifier
```

With these theorems we are getting a step closer to the proof of completeness, with most of them used to prove statements about *polynomial formulas* in the next section.

2.2 Polynomial formulas

Having the syntactic preparation theorems, we can prove more results about the provability relation, this time focusing on a special type of formulas. Those are theorems 44 - 60 and they play a direct role in the proof of completeness. We will refer to them as the *PF theorems*. These formulas are called *polynomial formulas* and they are recursively generated by linear functions with multiple variables, which we call *polynomials*. The recursion is executed using *degree* of a polynomial, which is a sum of its coefficients in absolute value. We will first define polynomials, then define a set of polynomial formulas and discuss well founded recursion, which allows us to perform recursion over degree of polynomial. After that, we will show some example theorems and in the end talk about a tactic that makes applying these theorems easier.

As mentioned, the polynomials are linear functions with multiple variables of shape $a + \sum_{i=0}^{n-1} b_i x_i$, where given n natural, a, b_0, b_1, …, b_{n-1} are integer coefficients, and x_0, x_1, …, x_{n-1} are rational variables. Since these functions are characterised only by their coefficients, our Lean definition uses only those. We divide the coefficients into the absolute coefficient a and the rest, i.e. b_0, b_1, …, b_{n-1}. The first coefficient a is one integer, type $\mathbb{Z}$. The rest of the coefficients can be represented as a finite series of length n. In Lean, this kind of series would be defined using `fin n`. The type `fin n` is a subtype of natural numbers that consists of numbers strictly smaller than `n`. Thus, the series can be represented as type `fin n → ℤ`. We put this together into a product type. (Of course, the `n` natural needs to be present in the definition, too.)

```
def poly_fun (n: ℕ) := ℤ × (fin n → ℤ)
```

Next, we define two special polynomials: a constant polynomial and a polynomial `xj`. The constant polynomial is a polynomial with all coefficients except the absolute one equal to zero. The polynomial `xj` is an injection of `fin n` into `poly_fun n`, i.e. it is a polynomial, that has all coefficients equal to zero except for coefficient $b_i = 1$ for some i. We also need a function that returns the value of a polynomial and, of course, a function that returns its degree. Since the `poly_fun` is a product, we can use the notation `⟨..., ...⟩` to define individual components $\mathbb{Z}$ and `(fin n → ℤ)` and `.1` and `.2` to access them. Note that the `val` function takes in as variables the truth values of atoms under evaluation `v`. This is the only application we need.

```
/- Constant polynomial f(x) = a. -/
def cons (a: ℤ): poly_fun n := ⟨a, (λ i, 0)⟩

/- Polynomial f(x) = xj. -/
def xj (j: fin n): poly_fun n :=
⟨0, (λ i, if i = j then 1 else 0)⟩

/- Value of a polynomial. -/
def poly_fun.val (v: eval): poly_fun n → ℚ :=
λ f, f.1 + Σ i, f.2 i * (v i)

/- Degree of a polynomial. nat_abs is an absolute value on integers. -/
def poly_fun.degree (f: poly_fun n): ℕ :=
Σ i, (f.2 i).nat_abs

/- Symbols and notation. -/
prefix `σ` : 100 := poly_fun.degree
prefix `C` : 100 := poly_fun.cons
notation v `*` := poly_fun.val v
```

Note, that the degree, being a sum of absolute values of integers, is always a natural number. Also, if it is equal to zero for some polynomial `f`, then `f` is a constant polynomial.

In order to work with `poly_fun`, it would be nice to know that the type has some algebraic properties. Using the fact that the type is based on $\mathbb{Z}$, Lean automatically defines an addition, subtraction and multiplication on `poly_fun` using these operations on $\mathbb{Z}$. These new operations are defined point-wise. On top of that, with these operations, `poly_fun` inherits the ring properties of $\mathbb{Z}$. Therefore, with no additional work, we can, using the `ring` tactic, prove for example (`C 1` is a constant polynomial with absolute coefficient equal to 1).

```
variables (n: ℕ)(f g h: poly_fun n)

example : f + g - (C 1 + h) = - C 1 + g - h + f := by ring
```

Let us move on to the definition of a set of polynomial formulas. Assume a polynomial `(f: poly_fun n)`. Set `PF f` of polynomial formulas generated by `f` is defined recursively on `σ f`, i.e. the degree of `f`. The definition of `PF` is quite complicated, but will hopefully be understandable with a few comments: we assume that `σ f = 0` and get two possible shapes of `PF` based on the sign of the absolute coefficient of `f`. In the induction step, we do a union over all positive and negative coefficients of `f`, each of them generating a special set (note that since `σ f ≠ 0` at least one such coefficient must exist). These sets are where the recursion is hidden. Remember, that `f.1` (type ℤ) is the absolute coefficient of `f`,that `f.2` (type `fin n → ℤ`) is the sequence of remaining coefficients, and that `p` is the atom constructor of `Form`. Note the overall notation and `using_well_founded` sequence at the end of the definition.

```
def PF: ∀ f: poly_fun n, set Form
| f :=
  /- Case σ f = 0. -/
  if h: σ f = 0 then begin
    by_cases (f.1: ℚ) > 0,
    -- in case that f.1 > 0
    exact {K | ∃ i ≤ n, K = p i ⇒ p i},
    -- in case that f.1 ≤ 0
    exact {K | ∃ i ≤ n, K = ~(p i ⇒ p i)} end
  /- Induction step, σ f ≠ 0 (i.e. σ f > 0). -/
  else begin
    -- union over all i: fin n
    exact ⋃ i,
    begin
    -- for each (f.2 i) coefficient we get a set
    by_cases 0 < (f.2 i),
    -- for every (f.2 i) > 0
    exact {K | ∃ Q ∈ PF (f - xj i) ,
               ∃ R ∈ PF (f + C 1 - xj i),
               K = (Q ⊕ p i) & R},
    by_cases h_1: f.2 i = 0,
    -- for every f.2 i = 0
    exact {},
    -- for every f.2 i < 0
    exact {K | ∃ Q ∈ PF (f - C 1 + xj i),
               ∃ R ∈ PF (f + xj i),
               K = (Q ⊕ ~p i) & R}
    end
  end
 using_well_founded {rel_tac :=
   λ _ _, `[exact ⟨ _, measure_wf (@poly_fun.degree n)⟩],
   dec_tac := `[simp [*, measure, inv_image]]}
```

The `using_well_founded` tactic allows us to do recursive definitions and inductive proofs using a custom well-founded relation. In our case the relation is the measure of `poly_fun.degree`, i.e. `λ f g, σ f < σ g`. The relation used is specified in the `rel_tac := ...` part of `using_well_founded`. In fact, the relation is implicit in term `measure_wf (@polyfun.degree n)`, which is a proof of the fact that measure `σ` is, indeed, well-founded. The `dec_tac := ...` part specifies a tactic that should be used to determine whether the inductive applications inside the definition are decreasing, hence reaching the initial step `σ f = 0` in a finite amount of steps. To be specific, the tactic `simp [*, measure, inv_image]` should be able to find proof of `σ (f - xj j) < σ f` given `f.2 j > 0`, so that the recursive usage of `PF (f - xj j)` is justified (similarly for the other three recursive calls of `PF` in the definition). To prove these inductive statements, we introduce several "degree simp lemmas". Two of these lemmas are stated below. There are different variants of these lemmas in the code to fit all instances we encountered. Note that one can, similarly as before, list all `simp` lemmas used to prove these induction assumptions by using `set_option trace.simplify.rewrite true`.

```
variables{n: ℕ}{j: fin n}

@[simp] lemma sum_abs_lt_degree (hb : f.2 j < 0)(hf : σ f = m + 1)
: σ (f + xj j) = m := sorry

@[simp] lemma sum_abs_gt_degree (hb : f.2 j > 0)(hf : σ f = m + 1)
: σ (f - xj j) = m :=  sorry
```

While proving theorems with polynomial formulas it is often required to take an arbitrary element from some `PF f`. To be able to do this, we need to know that such an element exists, i.e. that the set `PF f` is not empty. On paper, it is "apparent" for any `f`, in Lean, we have to prove it. The instance `nonempty_pf` can be proven by induction on the degree. (The `dec_tac` is slightly changed compared to the `PF` definition for internal reasons of the proof.)

```
instance nonempty_pf: ∀ f: poly_fun n, nonempty (PF f)
| f := sorry
using_well_founded {rel_tac :=
  λ _ _, `[exact ⟨ _, measure_wf (poly_fun.degree)⟩],
  dec_tac :=`[simp at hi, simp [*,measure,inv_image]]}
```

We can easily rephrase this into a lemma of more useful shape.

```
lemma exists_form_in_pf (f: poly_fun n): ∃ P: Form, P ∈ (PF f) := sorry
```

Example theorem

A simple example of a theorem from this part of the completeness proof could be lemma 45, whose statement is the following:

```
∀ (f: poly_fun n.succ), ∀ (P Q ∈ PF f), ⊢ (P ⇔ Q)
```

Generally, the theorems discuss provability results about polynomial formulas, in this case two formulas which both belong to the `PF` set generated by the same function `f`. Overall, the theorems follow the pattern

```
∀/-some poly_funs-/, ∀/-formulas from some PF sets-/, /-provability result-/.
```

Most theorems from this part are proven by induction on degree. Such proofs then usually go the following way: the starting case `σ _ = 0` is proven using derived rules and previous PF theorems, utilising the knowledge of how `PF _` looks given that `σ _ = 0`. The induction step then creates some mid-proof provability statements using previous PF theorems and induction hypothesis. The proof is often closed by applying some theorem from the syntactic preparation section 2.1. As an example we can use the aforementioned lemma 45. The proofs of lemmas 45 and 46 are intertwined, therefore the lemmas have to be put into a single statement `lemma45_46`. Here, we will focus on lemma 45. We will start with sketching the proof of the starting case `σ f = 0`, which for lemma 45 involves two simple formal proofs:

```
lemma lemma45_zero:
∀ f: poly_fun n.succ, σ f = 0 →
∀ P Q ∈ PF f, ⊢ (P ⇔ Q) :=
begin
  /- Thanks to σ f = 0 we know what P ∈ PF f and Q ∈ PF f look like. -/
  -- if f.1 > 0, then P = (p i ⇒ p i) and Q = (p j ⇒ p j) for some i and j
  let prf := p i ⇒ p i. && p j ⇒ p j. ▶▶
             (p i ⇒ p i) ⇔ (p j ⇒ p j),  -- i.e. P ⇔ Q
  by proof_verifier,
  -- if f.1 ≤ 0, then P = ~(p i ⇒ p i) and Q = ~(p j ⇒ p j)
  let prf := (p i ⇒ p i. ▶ ~~(p i ⇒ p i)) &&
             (p j ⇒ p j. ▶ ~~(p j ⇒ p j))
             ▶▶ ~(p i ⇒ p i) ⇔ ~(p j ⇒ p j),  -- i.e. P ⇔ Q
  by proof_verifier
end
```

Next, we can sketch the rest of the proof of lemma 45. We focus on one sub-part of the proof, the case `f.2 j > 0, f.2 k > 0, i ≠ k` as you can see below. Unfolding `PF` definition, we can derive the shapes of `P ∈ PF f` and `Q ∈ PF (f + C 1)`. Then, using the `exists_form_in_pf` lemma, we introduce new polynomial formulas into the proof. After that, we prove helping statements `h1`, `h2`, `h3`, `h4`, `h5`, in this case using only the inductive hypotheses, and finally close the goal by applying `theorem36`.

```
lemma lemma45_46 : ∀ (f: poly_fun n.succ),
(∀ (P Q ∈ PF f), ⊢ (P ⇔ Q)) ∧  -- lemma 45
(∀ (P ∈ PF f)(Q ∈ PF (f + C 1)), ⊢ ~P ⊔ Q)  -- lemma 46
| f :=
begin
  /- Case σ f = 0 comes from lemma45_zero. -/
  /- Case σ f ≠ 0. From unfolding P ∈ PF f and Q ∈ PF (f + C 1) we get the
 following formulas:
  hR: R ∈ PF (f - xj j)
  hS: S ∈ PF (f + C1 - xj j)
  hT: T ∈ PF (f - xj k)
  hU: U ∈ PF (f + C1 - xj k)

  hhP: P = (R ⊕ p j) & S
  hhQ: Q = (T ⊕ p k) & U -/
  -- if j = k
  sorry,
  -- if j ≠ k
  -- case f.2 j > 0, f.2 k > 0
  cases exists_form_in_pf (f + C 1 + C 1 - xj j- xj k) with Z hZ,
  cases exists_form_in_pf (f - xj j- xj k) with V hV,
  cases exists_form_in_pf (f + C 1 - xj j- xj k) with W hW,
  /- Formulas Z, V and W were introduced.
  hZ: Z ∈ PF (f + C1 + C1 - xj j - xj k)
  hV: V ∈ PF (f - xj j - xj k)
  hW: W ∈ PF (f + C1 - xj j - xj k) -/
  have h1: ⊢ ~V ⊔ W,  -- from the induction hypothesis of lemma 46
  have h2: ⊢ (R ⇔ ((V ⊕ p k)&W)),  -- from the ind. hyp. of lemma 45
  have h3: ⊢ (T ⇔ ((V ⊕ p j)&W)),  -- likewise
  have h4: ⊢ (((W ⊕ p k)&Z) ⇔ S),  -- ...
  have h5: ⊢ (((W ⊕ p j)&Z) ⇔ U),  -- ...
```

```
    rw [hhQ, hhP],
    exact theorem36 h1 h2 h4 h3 h5,
    -- remaining cases and the proof of lemma 46
    all_goals{sorry}
  end
  using_well_founded {rel_tac :=
    λ _ _, `[exact ⟨ _, measure_wf (@poly_fun.degree n.succ)⟩],
    dec_tac :=`[simp [*, measure, inv_image]]}
```

As can be seen, the proofs are often split into several cases, based on the definition PF. These cases, however, are often proven in a very similar manner, exchanging maybe two formulas or some polynomials. We can take advantage of this using the `let := ...`, `all_goals{}` and `work_on_goal n {}` tactics. The `all_goals { t }` applies the tactic `t` to all goals and succeeds only if all applications succeed. The `work_on_goal n {t}` tactic applies `t` to the nth goal (numbered from 0). The following example demonstrates how we can use those to solve multiple similar goals at once.

```
  example (n m : ℕ): n ≤ n + 1 ∧ m ≤ m + 1 :=
  begin
    split,
    /- GOAL 0: n ≤ n + 1, GOAL 1: m ≤ m + 1 -/
    work_on_goal 0 {let m':= n},
    work_on_goal 1 {let m':= m},
    /- Solve both goals using the "placeholder" m'. -/
    all_goals{exact nat.le_succ m'}
  end
```

This technique is utilised in several PF theorems.

One more theorem worth mentioning is the `theorem54`. In one step of the proof of `theorem54` we have a `f : poly_fun n` and are asked to take a coefficient of `f` that is equal to zero. Since we know nothing about the coefficients of `f`, we have to assume that all coefficients are non-zero. In that case, we would like to perceive `f` as a `poly_fun (n + 1)` with the last coefficient equal to zero. This extension is defined in the `pfcon` definition, which projects a `poly_fun n` into `poly_fun (n + 1)` by adding the nth coefficient (equal to zero) while keeping all other coefficients untouched.

```
  def pfcon : poly_fun n → poly_fun n.succ :=
  begin
    intro f,
    split, exact f.1,
    intro i,
    by_cases hh: i.val < n,
    exact f.2 ⟨i.val, hh⟩,
    exact 0
  end
```

This definition and its practical use of course require some basic statements such as e.g. `∀ f, σ (pfcon f) = σ f` and `∀ f P, P ∈ PF f → P ∈ PF (pfcon f))`, which can be found in the code.

Having this extension, we can use `pfcon f` instead of `f` and take the last coefficient, which is equal to zero. This extension is then used in the induction hypothesis, which forces a change in the measure we use in `using_well_founded`. The problem with the original measure is that it is a measure on `poly_fun n` for previously fixed `n`, therefore the induction hypothesis only works for `poly_fun n`. Because `f` is `poly_fun n`, our extension `pfcon f` is of type `poly_fun (n + 1)`. We can, however, alter the measure to take the `n` as well, since the relation is well founded over all `poly_funs` across all ns. This makes the induction hypothesis depend on `n`, enabling us to use it for a `poly_fun (n + 1)`. The theorem with this altered measure looks like this (the `dec_tac` is again slightly changed for internal reasons of the proof.):

```
theorem theorem54 :
∀ n : ℕ, ∀ g f: poly_fun n.succ,
∀ (P ∈ PF f)(Q ∈ PF g)(R ∈ PF (f + g))(S ∈ PF (f + C 1))(T ∈ PF (g + C 1)),
{S} ⊢ T ⇒ ((P ⊕ Q) ⇒ R)
| n g := sorry

using_well_founded
  {rel_tac := λ _ _, `[exact ⟨ _, measure_wf (λ ⟨n, g⟩, σ g)⟩],
  dec_tac := `[simp [*, measure, inv_image, ←sub_add, -prod.mk_add_mk,
             -prod.mk_sub_mk]]}
```

apply_PF tactic

As discussed above, the theorems in this part follow the pattern

```
∀/-some poly_funs-/, ∀/-formulas from some PF sets-/, /-provability result-/.
```

Any time we are applying these theorems, we need to provide the `poly_funs` and the formulas, alongside corresponding propositions of type `_ ∈ PF _`. Since this happens a lot in the code and we always have these propositions ready, it would be nice to have it automated. Unfortunately, there are a few smaller complications.

First is that when applying these theorems, Lean is not always able to fill in the input `poly_funs` correctly, as they are not always simple. This is something we can not really deal with, and we have to sometimes explicitly tell Lean which `poly_funs` to use. The second is, that for example the sets `PF (f - g)` and `PF (-g + f)` are not the same in Lean. The following example can be proven using the `convert` command, which creates a new goal for the difference between the original goal and the provided hypotheses (in this case `h1`).

```
example (h1: P ∈ PF (f - g)): P ∈ PF (-g + f) :=
begin
  /- Current goal: P ∈ PF (-g + f) -/
  convert h1 using 2,
  /- Current goal: -g + f = f - g -/
  ring,
end
```

The `convert` tactic works well, but it requires the name of the used hypothesis stated explicitly, which complicates its application in an automated process. In order to deal with this, we can define a special tactic that is able to find the correct hypothesis in the local context and return it. Suppose our goal is `P ∈ PF (-g + f)` and in the local context there is, among others, a hypothesis of shape `P ∈ PF (f - g)`. Given the goal, we can identify the correct hypothesis thanks to the involved formula, in this case `P`. In other words, we are looking for a proposition of shape `P ∈ PF _`. This relies on the fact, that in all instances of the code in which the PF theorems are used, there is always only one hypothesis of shape `P ∈ PF _` for any `P`.

This is how the general strategy of our hypothesis seeking tactic looks like. It has two inputs: the involved formula as an `expr`, to be able to identify the right hypothesis and the local context (i.e. all currently available hypotheses) as a `list expr`. The tactic goes through the local context list and tries to find a proposition of type `_ ∈ PF _`. If it succeeds, it checks the involved formula and tries to unify it with the input formula. The tactic then either returns the hypothesis, if the formulas match, or moves on to the next hypotheses from the list. If no hypothesis is found, the tactic fails.

```
open interactive.types (texpr)
open tactic

meta def tactic.interactive.find_PF (P : expr) : list expr → tactic expr
| [] := fail "failed to find hypothesis"
| (H::Hs) := do HH ← infer_type H,
  match HH with
  | `(%%Q ∈ _) := (unify P Q >> `[return (H)]) <|>
                  tactic.interactive.find_PF Hs
  | _ := `[tactic.interactive.find_PF Hs]
  end
```

We can now use this tactic to look for hypotheses for `convert`. The last thing we have to do is to identify the formula involved in the goal, so that `find_PF` knows what to look for. That can be done by matching the goal with a proposition `_ ∈ PF _` and naming the individual arguments. All this can be summarised in a simple tactic `test_tac`.

```
meta def tactic.interactive.test_tac : tactic unit :=
`[do `(%%j ∈ PF %%t) ← target,
  ctx ← local_context,
  hh ← tactic.interactive.find_PF j ctx,
  `[convert %%hh using 2, ring]]
```

Now we can go back to our example and have `text_tac` provide the proof.

```
example (h1: P ∈ PF (f - g)) : P ∈ PF (-g + f) := by test_tac
```

Since the ultimate reason for this is application of the PF theorems, we can alter the `test_tac` a bit, so that it can take a theorem/lemma it is supposed to use as an input.

```
open interactive (parse)

meta def tactic.interactive.test_tac (H : parse texpr): tactic unit :=
do w ← tactic.i_to_expr H,
`[apply %%w; try{assumption};
  do `(%%j ∈ PF %%t) ← target,
  ctx ← local_context,
  hh ← tactic.interactive.find_PF j ctx,
  `[convert %%hh using 2, ring]]
```

We can then use the `test_tac` as in the following example, which is an application of `lemma45_46` we used as an example of a PF theorem. As you can see, in this case, Lean is able to correctly fill in the `poly_fun`.

```
variable{n: ℕ}

example (g h : poly_fun n.succ)
(h1: P ∈ PF (g - h + C 1))(h2: Q ∈ PF (C 1 - h + g))
: ⊢ (P ⇔ Q) := by test_tac (lemma45_46 _).1
```

Note that while the idea is the same, the `test_tac` provided here is not the exact tactic we use in our code, since there are more small things to account for, when applying the PF theorems.

Conclusion

The Rose-Rosser's proof of completeness of Łukasiewicz propositional logic can be divided into three parts. In this work, a possible formalisation of the first two parts, which constitute the majority of the completeness proof, was presented. Specifically, we introduced all the logic's basic definitions (syntax, semantics) and proved some of its fundamental theorems (cut rule, soundness, structurality, etc.). Using Lean's automated proving tools, we created the `proof_verifier` tactic to automate the verification of formal proofs in the logic. In this work, the tactic is used to automatically verify 23 basic syntactic lemmas and more complex syntactic theorems. Despite the complexity of the formal proofs, the verification procedure in our code is a matter of tens of seconds. That is mainly thanks to the efficient tree representation of the formal proof.

Further, we formalised the notion of a polynomial and a polynomial formula and verified 16 complex theorems discussing these formulas. While proving these theorems, we utilised previous provability results as well as the `proof_verifier` tactic. Also, we used Lean's apparatus for performing well-founded recursion and induction using a custom measure needed to prove most of these theorems. In the end, a simple tactic was defined to make applying these theorems easier, going a step deeper into the realm of Lean's automated proving.

Overall, the Lean code of this formalisation consists of five files totalling around ~3600 lines and ~320 individual lemmas and theorems, that successfully formalise said parts of the completeness proof of Łukasiewicz propositional logic. Left to formalise is the Farkas' lemma with two corollaries, two theorems further connecting the polynomial representation of formulas and provability, and finally, the proof of completeness. We intend to complete the formalisation in the future.

Bibliography

[1] P. Fejl, Moderní čtení Rose-Rosserova důkazu úplnosti Łukasiewiczovy logiky. Bakalářská práce, FJFI ČVUT v Praze, Praha, 2017.

[2] J. Šimon, Strojové ověření konstruktivního důkazu úplnosti klasické výrokové logiky. Bakalářská práce, FJFI ČVUT v Praze, Praha, 2019.

[3] Łukasiewicz, O logice trójwartościowej (On Three-Valued Logic). Ruch Filozoficzny 5, 1920, 170–171.

[4] A. Rose, J. B. Rosser, Fragments of Many-Valued Statement Calculi. Transactions of the American Mathematical Society 87, 1958, 1–53.

[5] J. Avigad, L. de Moura, S. Kong, Theorem Proving in Lean. Online tutorial: `https://leanprover.github.io/theorem_proving_in_Lean`