

Obsah

1	Úvod	2
2	Curry-Howardov izomorfizmus	3
1	Intuicionistická logika	3
2	λ -kalkulus	5
3	Typovo jednoduchý λ -calculus	7
4	Curry-Howardov izomorfizmus	8
3	Lean dokazovací asistent	10
1	Mathlib	10
2	Vývojové prostredie	11
3	λ -kalkulus	12
3.1	Konštanty, aplikácie	12
3.2	Funkcie	13
3.3	Typy	14
3.4	Premenné	15
3.5	Kontext, priestor mien	15
4	Dokazovanie	16
4.1	Dopredné dokazovanie	16
4.2	Spätné dokazovanie	17
5	Abstraktné štruktúry v Lean-e	18
5.1	Induktívne štruktúry	19
5.2	Jednoduché štruktúry	20
5.3	Typové triedy	21
4	Teória čiastočného usporiadania	23
1	Zväz	24
2	Modulárne zväzy	25
3	Podzväz	28

Kapitola 1

Úvod

Dokazovací asistenti obsahujú interaktívne prostredie ktoré na základe vstupu používateľa tvoria formálne dôkazy. Táto diplomová práca sa zaoberá dokazovacím asistentom Lean ktorý je založený na teórii závislostných typov. V úvode práce uvádzame teoreticky nevyhnutú časť intuicionistickej logiky a jednoduchej teórie typov aby sme ukázali prepojenie medzi logikou a výpočtovými modelmi ktoré je známe ako Curry-Howardova izomorfizmus.

Druhá kapitola opisuje prostredie Lean-u, jeho základnú syntax a prácu s typmi ktoré sú jeho základným prvkom. Poskytujeme pohľad na dva spôsoby dokazovania. Dopredným spôsobom ktoré je priamym poskytnutím λ -výrazu, a spätným v taktickom móde ktoré naplno využíva interaktívne prostredie. Kapitolu uzatvárame opisom syntaxe využívanej na budovanie štruktúr používanej pri tvorbe tých matematických a ich hierarchie.

V poslednej kapitole rozoberáme našu implementáciu vety o izomorfizme modulárnych zväzov. Opisujeme základné definície a štruktúry týkajúce sa teórie čiastočného usporiadania implementovaných v rámci komunitného projektu mathlib. Na záver podrobne opisujeme náš príspevok štruktúry podzväzu do tejto knižnice aj s jeho atribútmi.

Kapitola 2

Curry-Howardov izomorfizmus

1 Intuicionistická logika

Konstruktivizmus je jeden z filozofických smerov ktorý hovorí o tom že vedmosti sú tvorené a mali by byť zahrnuté k doteraz poznanému. Tento filozofický smer mal vplyv aj na matematiku kde sa na jeho základe vytvorilo viacero "škôľako finitizmus, predikativizmus, intuicionizmus. *Intuitionizmus* ako jeden z nich je teda konštruktívny prístup k matematike v duchu učenia Brouwera(1881-1966) a Heytinga(1898-1980). Filozofickým základom tohto prístupu je princíp že matematika je výtvorom mentálnej činnosti a nepozostáva z výsledkov formálnej manipulácie symbolov ktoré sú iba sekundárne. Jedným z princípov je tak odmietnutie postulátu zákona vylúčenia tretieho z klasickej logiky.

$$p \vee \neg p \tag{2.1}$$

V prípade dôkazu sporom nad výrokom p je možné dokázať existenciu p , čo je z konštruktívneho pohľadu nezmysel pretože uvažujeme nad pravdivosťou výroku nezávisle od uvažovaného tvrdenia. Výrok v intuicionistickej logike je teda pravdivý ak existuje dôkaz o jeho pravdivosti a nepravdivý ak existuje dôkaz ktorý vedie k sporu. V logike definujeme dôkaz ako z nejakých predpokladov ako konečnú postupnosť formúl, pri ktorých tvorbe môžeme v každom kroku spraviť jeden z nasledujúcich úkonov:

- Napísať postulát alebo axióm logiky
- Napísať jeden z predpokladov
- Napísať formulu, ktorú dostaneme aplikáciou dedukčného pravidla na niektorej formuly predchádzajúcej v postupnosti.

Všetky tieto kroky si postupne zavedieme aj s dedukčnými pravidlami. Pre lepšie pochopenie pojmu formuly a premennej uvádzame ich nasledujúcu definíciu:

Definícia 2.1 (Výroková premenná, formula). *Majme spočítateľnú množinu \mathcal{X} výrokových premenných. Množina premenných alebo formúl \mathcal{A} generujeme nasledovnou gramatikou:*

$$A, B ::= X \mid A \implies B \mid A \wedge B \mid A \vee B \mid \neg A \mid \top \mid \perp \tag{2.2}$$

Kde $X \in \mathcal{X}$ reprezentuje výrokovú premennú, a $A, B \in \mathcal{A}$ formulu.

Formulu ktorú tvorí len výroková premenná nazývame atomickou. V úvode tohto textu sme hovorili o postuláte alebo axióme zákone vylúčenia tretieho ktorý je formulou. Ďalším príkladom formuly generovanej uvedenou gramatikou je:

$$\neg A \wedge B \wedge C \implies A \vee B$$

V prípade že by sme chceli predchádzajúcu formulu ohodnotiť je precendencia negácie vyššia ako konjunkcie a disjunkcie a tie ju majú vyššiu ako implikácia. V prípade formuly obsahujúcej viac za sebou idúcich binárnych operátorov platí asociácia z pravej strany. V prípade že chceme uviesť jeden z predpokladov do dôkazu vyberáme z množiny ktorú nazývame kontextom.

Veta 2.1. *Kontextom(systém predpokladov) rozumieme zoznam formúl značených*

$$\Gamma = P_1, \dots, P_n \quad (2.3)$$

Dedukciou nazývame dvojicu pozostávajúcu z kontextu a formuly.

$$\Gamma \vdash A \quad (2.4)$$

Výraz $\Gamma \vdash A$ čítame ako premennú A je možné dokázať zo systému predpokladov Γ . Nad uvedenými predpokladmi a postulátmi potom pomocou dekučných pravidiel prirodzenej intucionistickej logiky potom odvádzame nové formuly a rozširujeme tak teóriu. Zaužívanou notáciou pre dedukčné pravidlá je nasledovná:

$$\frac{\Gamma_1 \vdash A_1 \quad \dots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A} \quad (2.5)$$

Hornú časť tvorí množinu dedukcií Γ_i ktoré nazývame prepokladom a dolnej Γ ktorú nazývame záverom. V prípade že čítame dedukčné pravidlo alebo dedukčný strom tvorený takýmito pravidlami zhora nadol hovoríme o dedukcii v opačnom smere o indukcii. Prirodzená intucionistická logika obsahuje nasledujúce pravidlá:

$$\frac{}{\Gamma, A, \Gamma' \vdash A} \text{ (ax)}$$

$$\frac{\Gamma \vdash A \implies B \quad \Gamma \vdash A}{\Gamma \vdash B} \text{ (}\implies\text{ }_E\text{)}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash B} \implies\text{ }_I$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A} (\wedge^l_E) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash B} (\wedge^r_E)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} (\wedge_I)$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} (\vee_E)$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} (\vee^r_I) \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} (\vee^l_I)$$

$$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \perp} (\neg_E)$$

$$\frac{\Gamma, A \vdash \perp \quad \Gamma \vdash A}{\Gamma \vdash \neg A} (\neg_I)$$

$$\frac{\Gamma \vdash \perp}{\Gamma \vdash A} (\perp_E)$$

Záver pravidiel s dolným indexom E majú v závere výrokové premenné. Pravidlá s dolným indexom I úvadajú neatomické formuly. Pravidlo \Rightarrow_E je známe ako *modus ponens*. Fragmentom logiky nazývame, systém ktorý dostaneme ak sa obmedzíme len na niektoré z dedukčných pravidiel. My uvedieme implikačný fragment ktorý je potom použitý v Curry-Howardovom izomorfizme.

Veta 2.2. *Implikačný fragmento intuicionistickej logiky dostaneme v prípade ak formuly budú tvorené gramatikou*

$$A, B ::= X | A \Rightarrow B \quad (2.6)$$

a pravidlami (ax) , (\Rightarrow_E) , (\Rightarrow_I)

Aj keď tento fragment tvorí len jedna logická spojka a výrokové premenné je možné pomocou nich odvodiť väčšinu logicky ekvivalentných formúl tvorených gramatikou uvedenou na začiatku.

2 λ -kalkulus

λ -kalkulus je výpočtový model ktorý sa v informatike využíva pre svoju jednoduchosť a výpočtovú schopnosť. Napriek tomu že formálne tento model tvoria len tri výrazy je jeho výpočtová schopnosť rovná *Turingovmu stroju*. To znamená že v ňom dokážeme naprogramovať všetky algoritmy ktoré dokáže spracovať klasický počítač. Pre tieto vlastnosti je využívaný aj ako základ pre všetky funkcionálne programovacie jazyky. V tejto sekcii si ho formálne uvedieme aj s príkladmi ktoré ukazujú ako je v ňom možné zakódovať jednoduché algoritmy.

Veta 2.3. *Množinu Λ tvorenú λ -výrazmi je potom generovaná nasledovnou gramatikou:*

$$t, u ::= x | tu | \lambda x. t \quad (2.7)$$

Kde prvý výraz x patrí do nekonečnej spočítateľnej množiny $\mathcal{X} = x, y, z, \dots$ premenných.

Jednotlivé výrazy majú nasledovný význam:

x - je premennou

tu - je aplikáciou výrazu t s argumentom u

$\lambda x. t$ - je abstrakciou t nad x

Premenná x sa vo výraze $\lambda x. t$ viaže na výraz t . O premennej potom x hovoríme že je viazaná inak je voľná.

$$VP(x) = x$$

$$VP(\lambda x. t) = VP(t) \setminus \{x\}$$

$$VP(tu) = VP(t) \cup VP(u)$$

Aplikácia λ -výrazov je implicitne aplikovaná zľava. Precedencia aplikácie je vyššia ako u abstrakcie.

$$\lambda x.tx = \lambda x.(tx)$$

Abstrakciu viacerých argumentov je možné prepísať do tvaru po sebe idúcich abstrakcií s jednotlivými argumentami.

$$\lambda xyz.t = \lambda x.\lambda y.\lambda z.t$$

Príkladmi λ -výrazov môžu byť:

$$\begin{aligned} tx \\ (\lambda y.\lambda x.ty) \\ (\lambda y.yx)(\lambda x.x) \end{aligned}$$

Veta 2.4. *O substitúcii hovoríme pri nahradení jednej premenej druhou.*

$$t[y/x] \tag{2.8}$$

Z doteraz uvedených výrazov nám výpočet čiastočne pripomínala len aplikácia na jednoduchý výraz. Ak za výraz dosadíme jednoduchú abstrakciu, aplikáciu tvorí nahradenie viazanej premennej vo výraze abstrakcie za aplikovanú premennú. Tento úkon nám zjednodušuje výraz na jednoduchú aplikáciu. Toto zjednodušovanie sa volá β -redukciou. Od komplikovanejších výrazov sa tak dostávame k jednoduchším predstavujúcimi výsledky výpočtu. β -redukcia predstavuje postupnosť v ktorej aplikujeme pravidlá aplikácii. Formálne je β -redukcia definovaná aplikovaným týchto pravidiel.

$$\begin{array}{c} \frac{}{(\lambda x.t)u \rightarrow_{\beta} t[u/x]} (\beta_s) \qquad \frac{t \rightarrow_{\beta} t'}{(\lambda x.t)u \rightarrow_{\beta} t[u/x]} (\beta_{\lambda}) \\[10pt] \frac{t \rightarrow_{\beta} t'}{tu \rightarrow_{\beta} t'u} (\beta_l) \qquad \frac{u \rightarrow_{\beta} u'}{tu \rightarrow_{\beta} tu'} (\beta_r) \end{array}$$

TU TREBA VYMYSLIET INY STROM

$$\frac{\frac{\frac{}{(\lambda y.y)x \rightarrow_{\beta} x} (\beta_s)}{(\lambda y.y)xz \rightarrow_{\beta} xz} (\beta_l)}{\lambda x.(\lambda y.y)xz \rightarrow_{\beta} \lambda x.xz} (\beta_{\alpha}) \tag{2.9}$$

Pre lepšiu predstavu uvádzame komplikovanejšie príklady ktorými je možné zakódovať prirodzené čísla a jednoduchú podmienku z programovania.

Veta 2.5. *Definujme rekúziu volania funkcie nasledovne*

$$\begin{aligned} f^0 x &= x \\ f^n x &= f(f^{n-1} x) \end{aligned}$$

Potom Churchove číslo c_n je λ -výraz

$$c_n = \lambda s.\lambda z.s^n(z)$$

Prirodzené čísla je potom možné definovať

$$0 = \lambda f x. x$$

$$1 = \lambda f x. f x$$

$$1 = \lambda f x. f(f x)$$

$$2 = \lambda f x. f(f(f x))$$

$$\begin{aligned} \text{nasledovnik}(n) &= (\lambda n f x. f(n f x))(\lambda f x. f^n x) \\ &\rightarrow_{\beta} \lambda f x. f((\lambda f x. f^n x) f x) \\ &\rightarrow_{\beta} \lambda f x. f((\lambda x. f^n x) x) \\ &\rightarrow_{\beta} \lambda f x. f(f^n x) \\ &= \lambda f x. f^{n+1} x \\ &= n + 1 \end{aligned}$$

Operáciu sčítania je potom možné vykonať pomocou nasledujúceho výrazu

$$f_+ = \lambda x. \lambda y. \lambda s. \lambda z. x s (y s z)$$

Pred vytvorením výrazu ktorý predstavuje podmienku si potrebujeme zakódovať booleovské hodnoty:

$$True = \lambda x y. x$$

$$False = \lambda x y. y$$

Podmienku potom predstavuje nasledujúci výraz, na ktorý potom aplikujeme λ -výraz predstavujúci logickú podmienku.

$$if = \lambda b x y. b x y$$

Jednotlivé vetvy výpočtu potom predstávajú ďalšie dva výrazy aplikované na celý výraz podmienky a výrazu. Po aplikácii výrazov t, u cez β -redukcie dostávame na konci výraz t v prípade pravdivého vyhodnotenia a u respektíve t v prípade nepravdivého.

$$\begin{aligned} if \text{ True } tu &= (\lambda b x y. b x y)(\lambda x y. x) tu \rightarrow_{\beta} (\lambda x y. (\lambda x y. x) x y) tu \\ &\rightarrow_{\beta} (\lambda y. (\lambda x y. x) t y) u \\ &\rightarrow_{\beta} (\lambda x y. x) t u \\ &\rightarrow_{\beta} (\lambda y. t) u \\ &\rightarrow_{\beta} t \end{aligned}$$

3 Typovo jednoduchý λ -calculus

Typový jednoduchý λ -kalkulus je rozšírením o jednoduché typy ktoré priradzujeme λ -výrazom. Výraz $t : T$ tak môžeme interpretovať spôsobom " t patrí množine T alebo z výpočtového "výsledkom výrazu t

je typ T'' . Z praktického hľadiska sa s typmi stretávame v staticky typových programovacích jazykoch kde nám zaručujú že typovo správny výsledok.

Veta 2.6. *Majme spočítateľnú množinu U obsahujúcu typové premenné. Jednoduché typy sú potom generované gramatikou*

$$A, B ::= U | (A \rightarrow B)$$

Teda okrem jednoduchých typov sú typmi aj funkcie medzi nimi. Precedencia zloženia funkcií typov je zľava $A \rightarrow (A \rightarrow B)$. Množinu tvoriacu premenné ktorým sú priradené typy nazývame kontextom.

Veta 2.7. *Kontextom je:*

$$x_1 : \tau_1, \dots, x_n : \tau_n$$

kde $\tau_1, \dots, \tau_n \in \Pi$ a $x_1, \dots, x_n \in \text{Obor kontextu}$ je množina obsahujúca

$$\text{domain}(\Gamma) = x_1, \dots, x_n$$

Koobor kontextu je množina obsahujúca

$$\text{range}(\Gamma) = \tau \in \Pi | (x : \tau) \in \Gamma$$

Príklady jednoduchých typov generované gramatikou.

- $\vdash \lambda x.x : \sigma \rightarrow \sigma$
- $\vdash \lambda x.\lambda y.x : \sigma \rightarrow \tau \rightarrow \sigma$
- $\vdash \lambda x.\lambda y.\lambda z.xz(yz) : (\sigma \rightarrow \tau \rightarrow \rho) \rightarrow (\rho \rightarrow \tau) \rightarrow \sigma \rightarrow \rho$

Výraz t je typu A ak v kontexte Γ je derivovateľná pomocou postupnosti nasledujúcich pravidiel:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} (ax)$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \lambda x^A.t : A \rightarrow B} (\rightarrow)$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} (E)$$

- (ax) : v kontexte x je typu A
- (\rightarrow) : ak je x typu A , t je typu B , potom funkcia $\lambda x.t$ ktorá asociuje x t je typu $A \rightarrow B$
- (E) : daná je funkcia t je typu $A \rightarrow B$ a argument u je typu A , výsledok aplikácia tu je typu B

Príklad odvodenia typu.

4 Curry-Howardov izomorfizmus

V úvode kapitoly sme popisovali konštruktivistický prístup ktorý hovoril o nutnosti skonštruovať objekt pre dôkaz existencie. Teda ak chceme dokázať nejaké tvrdenie potom ho musíme zostrojiť z množiny predpokladov a axióm. *Curry* s *Howard* si všimli že zostrojovanie dôkazu pripomína výpočet.

Pre dôkaz implikácie $a \text{ implies } b$ musíme ukázať na základe predpokladu a dôsledok b v konštruktivizme zostrojiť objekt b z objektu a a z výpočtového hľadiska zostrojiť funkciu s výstupom b na základe vstupu a .

Pri predstavení jednoduchých typov sme vraveli o možnosti interpretácie výrazu $t : T$ ako t patrí do nejakej množiny T . Z konštruktívneho pohľadu sme o logických tvrdeniach hovorili ako o objektoch ktoré majú svoje vymedzenie z logického hľadiska. V tomto zmysle z množinového hľadiska prepojenie medzi typovým λ -kalkulusom a logikou prepojenie znázornené v nasledujúcej tabuľke.

Logika	Typovo jednoduchý λ kalkulus
tvrdenia	typy
dôkaz tvrdenia T	λ -výraz t typu T
$T \implies P$	typ $T \rightarrow P$
$T \wedge P$	typ $T \times P$

Posledný príklad tvorí kosúčinový typ ktorý je usporiadanou dvojicou dvoch typov ktorý je aj priamo podporovaný Lean-om.

Nasledujúci odsek potrebujem preformulovať:

V sekcii o intuicionistickej logike sme zaviedli gramatiku implikačného fragmentu intuicionistickej logiky s tromi dedukčnými pravidlami (\implies_E), (\implies_I), (ax). V teórii jednoduchých typov sme zase ukončili pravidlami ktoré odvádžajú typ pomocou pravidiel (\xrightarrow{I}), (\xrightarrow{E}) a (ax). Ak dáme do rovnosti množinu propozičných premenných s množinou typových premenných.

Z formálneho hľadiska je:

Veta 2.8. *Curry-Howard izomorfizmus*

- Ak $\Gamma \vdash M : \varphi$ potom $|\Gamma| \vdash \varphi$.
- Ak $\Gamma \vdash \varphi$ potom existuje $M \in \Lambda_\Pi$ také že $\Delta \vdash M : \varphi$, kde $\Delta = (x_\varphi : \varphi) | \varphi \in \Gamma$

Kapitola 3

Lean dokazovací asistent

Lean je dokazovací asistent ktorý bol vytvorený ako otvorený softvérový projekt Leonardom de Mourom v Microsoft Research v roku 2013. Jazyk sa neustále vyvíja a momentálne sa nachádza vo štvrtej iterácii [4] zatiaľ čo komunitný projekt matematickej knižnice *mathlib* sa stále vyvíja v tretej verzii [3] vyvíjanej od roku 2017. Implementácia Lean-u je v jazyku C++ a jeho jadro má len 8000 riadkov. Prostredie je dostupné pre operačné systémy Linux, Windows a MacOS. Interaktívne prostredie pre dokazovanie je podporované pre editory *Emacs* a *Visual Studio Code*.

Lean podobne ako *Coq* je založené na kalkuluse konštrukcií ktorý je zovšeobecnením teórie jednoduchých typov a teórii závislostných typov.

V úvode tejto kapitoly popisujeme interaktívne prostredie a komunitný projekt *mathlib*. Opisujeme základnú syntax ktorá korešponduje s jednoduchou teóriou typov a príkazmi potrebné pre prácu s nimi. Syntax potom rozširujeme o možnosť tvorby definícií a prácu s nimi a priestorom mien pre vytváranie hierarchie z pohľadu organizácie kódu. Pokračujeme opisom spôsobu tvorby dôkazu priamym poskytnutím λ -výrazu a interaktívnou verziou v taktickom móde. Kapitola je uzatvorená opisom syntaxe popisujúcim induktívne a jednoduché štruktúry ktoré tvoria tie matematické a ich hierarchické rozširovanie a priradenie do tried.

1 Mathlib

Mathlib je komunitný projekt [5] ktorého cieľom je centralizovať matematickú teóriu implementovanú v Lean-e. Do projektu je možné jednoducho prispievať po udelení privilégii niektorým zo správcov repozitára a odobrením požiadavky na začlenenie kódu. Väčšina obsahu *mathlibu* obsahuje matematiku na vysokoškolskej úrovni. V dobre písanej práci je najvyššia hierarchia teórie nasledovná:

```
algebra/  
category_theory/  
data/  
geometry/  
measure_theory/  
probability_theory/  
algebraic_geometry/  
combinatorics/  
group_theory/
```

```

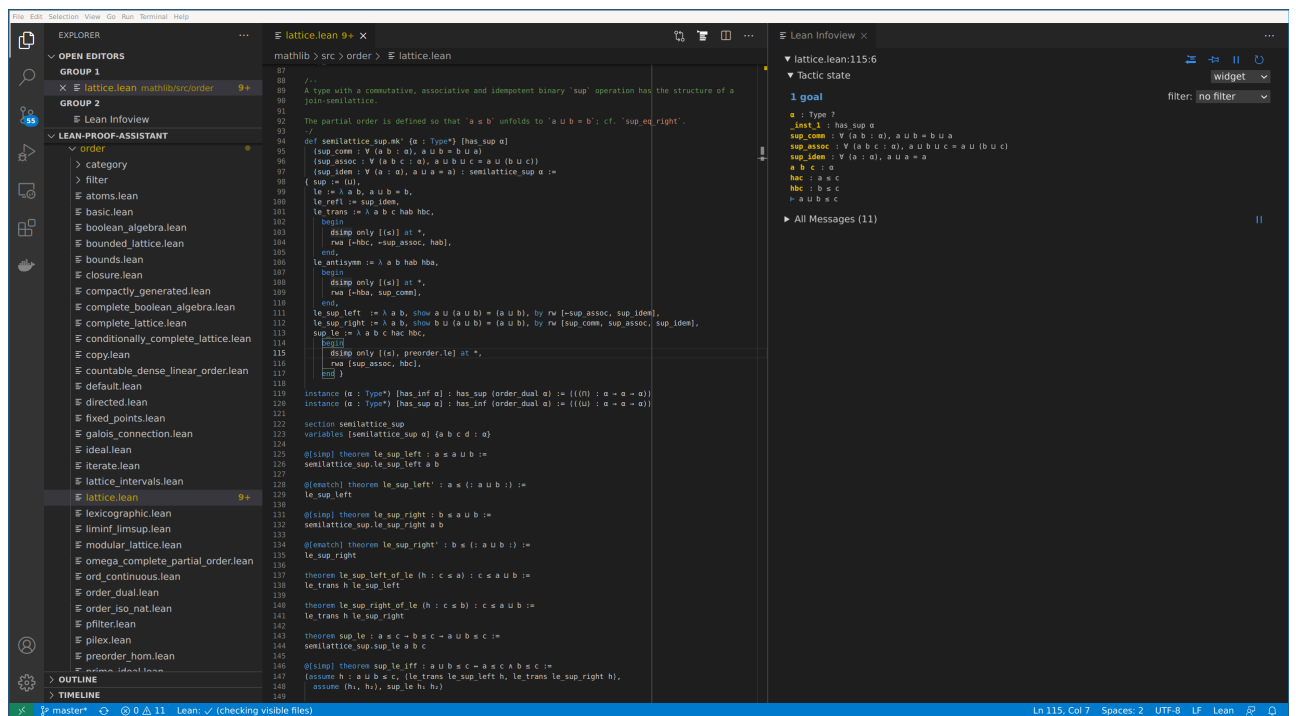
representation_theory/
algebraic_topology/
computability/
dynamics/
linear_algebra/
number_theory/
ring_theory/
analysis/
control/
field_theory/
logic/
order/
set_theory/
topology/

```

V kontraste s inými modernými dokazovacími asistentami má mathlib množstvo prispievateľov akademické vzdelanie v čistej matematike [6] čo ovplyvnilo aj jeho obsah.

2 Vývojové prostredie

V našom prípade sme pracovali vo vývojovom prostredí *Visual studio code* v kombinácii s jeho leanovským rozšírením ktoré je možné nainštalovať cez *marketplace*. Prostredie sa skladá z editora podporujúceho UTF-8 znaky a okno s interaktívnym výstupom reagujúce na polohu kurzora editora a kurzora počítačovej myši.



Obr. 3.1: Vývojové prostredie

V pravom okne *Lean infoview* je možné vidieť premenné s ich prislúchajúcimi typmi a v prípade dôkazu aj formulu ktorú je potrebné dokázať za znakom \vdash . Okrem toho poskytuje okno aj výstup zo

zabudovaných príkazov prostredia ako *print* alebo *reduce* ktoré rozoberieme neskôr. V prípade že sa nachádzame v taktickom móde okrem zavedenia nových predpokladov alebo transformácie existujúcich je možné vidieť aj zmenu cieľa a podcieľov napríklad v prípade že sme sa dostali k dôkazu vymenovaním prípadov.

3 λ -kalkulus

Pred predstavením techník dokazovania je nutné sa oboznámiť s prvkami funkcionálneho programovania v Leane. Výpočtový model jednoduchého λ -kalkulu je z programovacích paradigiem najbližšie práve funkcionálnemu spôsobu programovania. V nasledujúcej časti predstavujeme základy funkcionálneho programovania spolu s typmi a nástrojmi Lean-u na vývoj a menežment priestor mien.

3.1 Konštanty, aplikácie

Deklarácia konštanty zavádza do systému novú deklaráciu bez definície. Z tohto dôvodu sa ich pri rozvoji teórie snažíme vyhýbať. V nasledujúcich príkladoch budeme pracovať s prirodzenými a celými číslami ktorých štruktúry sú súčasťou kontextu bez nutnosti ich importovať.

```
constant m : nat
```

Hovorí o deklarovaní konštanty m ktorej typ je nat . Alternatívny zápis pre prirodzené číslo je pomocou sekvencie `|nat` alebo `|N` ktorú skonvertuje leanovské rozšírenie na znak \mathbb{N} . Vstavaný príkaz ktorý poskytuje typ výrazu zadaného na argumente je `#check`:

```
#check m
```

Mriežka na začiatku príkazu značí zabudovaný príkaz. V tomto prípade je to triviálne tak ako bola konštanta zadefinovaná s výstupom v informačnom okne:

```
m : ℕ
```

Pre zadefinovanie viacerých konštánt jedným príkazom a nie len v tomto prípade existuje plurálna verzia príkazu *constants*.

Definícia konštanty typu funkcie medzi prirodzenými číslami vyzerá nasledovne:

```
constant f : ℕ → ℕ
constant h : ℕ → ℕ → ℕ
```

Aplikácia funkcie sa notačne podobá aplikácii v λ -kalkule kde argument jednoducho pripíšeme za funkciu:

```
#constants m n : ℕ
```

```
#check f m
#check h m
#check h m n
```

Zatiaľ čo v prvom prípade dostaneme typ \mathbb{N} v druhom prípade $\mathbb{N} \rightarrow \mathbb{N}$ a v treťom vidíme aplikáciu na funkciu kde bude výsledným typom znova jednoduchý typ \mathbb{N} . Aplikácia je asociatívna z ľavej strany a preto je nasledujúci výraz potrebné uzátvorkovať napravo inak dostávame typový error pre funkciu g ktorá očakáva celé číslo a nie typ funkcie f .

```
constant f : ℕ → ℤ
constant g : ℤ → ℕ
constant a : ℕ
```

```
#check g (f a)
```

Číselné konštanty má v leane typ \mathbb{N} .

```
#check 5
#check (-5 : ℤ)
```

Pri deklarácii záporného čísla je tak nutné už explicitne uviesť typ. Nasledujúce príklady ilustrujú okrem funkcie $+$ aj implicitnú konverziu medzi typmi.

```
#constants (m : ℕ) (n : ℤ)
```

```
#check 1 + 2
#check m + 1
#check n + 1
#check n + m
#check m + n
```

V prípade prvého príkladu dostávame typ \mathbb{N} pre nespracovaný výraz pre ktorý by sme mohli očakávať výsledok 3. Pre druhý a tretí výraz dostávame typy definovaných konštánt. Pri treťom výraze je vhodné si uvedomiť už implicitnú konverziu prirodzeného čísla na celé. Konverzia je ešte viac zrejmá pri štvrtom výraze kde výsledným typom je \mathbb{Z} . Prekvapivo z piateho výrazu dostávame v informačnom okne chybový výstup. Za neschopnosťou dostať typ stojí nedefinovaná konverzia z celých do prirodzených čísel. Za vysvetlením stojí spôsob akým pracuje preťaženie infixového operátora $+$ podobným spôsobom ako pre triedy v jazyku *c++* alebo *python*. Štvrtý a piaty výraz je tak možné prepísať aj do tvaru:

```
#check n.add(m)
#check m.add(n)
```

Pretože znak $+$ je preťažením funkcie *add* nad štruktúrou množiny daných čísel.

3.2 Funkcie

V predchádzajúcej sekcii sme si ukázali explicitný typ ktorý bol funkciou prirodzenými číslami ktorý bol typu $\mathbb{N} \rightarrow \mathbb{N}$ alebo $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$. V Lean-e definujeme anonymnú funkciu alebo λ -výraz nasledovne.

```
#check λ x, x + x
```

Typ argumentu x je odvodený z výrazu ktorý na funkciu aplikujeme. Pre korektnú aplikáciu musí mať typ aj definovanú funkciu sčítania. Ak by sme chceli obmedziť argument len na konkrétny typ robíme to podobne ako pri definovaní konštanty. Potom aplikácia

```
#check (λ (x : ℕ), x + x) (m : ℤ)
```

je už typovou chybou pri ktorej nám informačné okno hlási

```
type mismatch at application
  (λ (x : ℕ), x + x) m
term
  m
has type
  ℤ
but is expected to have type
  ℕ
```

Uvedieme si zopár príkladov výpočtovo ekvivalentných funkcií obvodu obdĺžnika typu $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$.

```
#check λ (x : ℕ) (y : ℕ), (x + x) + (y + y)
#check λ (x y : ℕ), (x + x) + (y + y)
#check λ (x : ℕ), λ (y : ℕ), (x + x) + (y + y)
```

Všetky tri príklady sú ekvivalentné z pohľadu typov a predstavujú iný spôsob zápisu. Funkcia v ktorej neuvedieme typ argumentu je polymorfnou.

```
#check λ x y, x + x + y + y
```

V prípade že by sme chceli získať hodnotu z výrazu použijeme príkaz `#eval`, alebo `#reduce`. *Reduce* na rozdiel od *eval* pri vykonávaní používa jadro na získanie typu a je tak menej efektívnym.

```
#eval (λ (x y : ℕ), (x + x) + (y + y)) 2 3
```

Aplikácia nám dáva výsledok 10 ktorý je očakávanou hodnotou.

Pre pomenovanie alebo zadefinovanie takejto funkcie používame príkaz `def` ktorej tvar v najjednoduchšej podobe má tvar:

```
def meno_definie (argument_1 : typ) (argument_n : typ) :
  typ_navratovej_hodnoty
:=
  telo funkcie
```

V prípade definície štvorca aj s jeho výpočtom

```
def obvod_stvorca (x : ℕ) (y : ℕ) : ℕ := x + x + y + y

#eval obsah_stvorca 3 5
```

Kľúčové slovo `def` ktoré používame je funkčne bez funkčného roziedlu možné zameniť za *theorem*, *lemma*. V prípade že chceme uviesť iba príklad kde meno by pôsobilo nadbytočne používame slovo *example*.

3.3 Typy

Doteraz sme uvažovali len s typmi množiny prirodzených a celých čísel ktoré predstavujú na pozadí Leanu konkrétne definovanú štruktúru. Typovanie v Lean-e ale podporuje zavedenie nového abstraktného typu ktorý patrí univerzu(*universe*). Zavedenie týchto univerz je motivované problémom analogickým s Russelovým paradoxom a teda či množina všetkých množín obsahuje samú seba. V teórii typov sa jedná o Girardov paradox. Hierarchia týchto univerz je usporiadaná od 0 čo je univerzum tvoriace najjednoduchšie typy takže *Type* alebo aj *Type 0* je potom typom *Type 1* ktorý je typom *Type 2*.

Špeciálne postavenie v typoch majú tvrdenia označujeme *Prop* ktorého typom sú všetky dôkazy. Sú postavené v hierarchii oddelené na najnižšej úrovni a teda *Prop* je typu *Type*. Univerzum do ktorého patrí *Prop* označujeme *Sort* čo je len alias. Platí vzťah $\text{Sort } u + 1 = \text{Type } u$.

V tejto práci a ani vo väčšine práce s Leanom nie je potrebné využívať hierarchiu typov sofistikovanejším ako prezentovaným spôsobom.

```
universe u v
```

```
constant (a : Type u) (b : Type v)
```

Druhou možnosťou je využiť substitučnú syntax kde `*` znamená pre ľubovoľné univerzum a v prípade `_` necháme doplniť typ automaticky Lean-om.

```
constant f : Type _ → Type _
constant g : Type
```

```
#check f g
```

Kontrola typu aplikácie nám vypíše typ `Type u_1` čo je spôsob Lean-u označovať ešte neurčený typ bez konkrétneho univerza.

3.4 Premenné

V prípade že sa snažíme definovať viacero funkcií s rovnakými argumentami alebo používať objekt ktorého typ je vyjadrený komplikovanejším zápisom je vhodné si zápis zjednodušiť premennými. Z inak na pohľad komplikovaného zápisu

```
universes u v
```

```
def kompozicia (α : Type u) (β : Type v) (f : α → β) (g : β → γ) (x : α) :
  α
:=
  g (f x)
```

tak dostávame. Premenné vstupujú do predpokladov definícií len tam kde sú použité v jej tele.

```
universes u v z
```

```
variable α : Type u
variable β : Type v
variable γ : Type z
```

```
def kompozicia (x : α) (f : α → β) (g : β → γ) : α := g (f x)
```

Vstupné argumenty dokonca môžeme vynechať úplne, v niektorých prípadoch to ale môže byť kontra-produktívne z dôvodu orientácie pri rozsiahlych zápisoch v editore.

```
variables (f : α → β)
          (g : β → γ)
```

```
def kompozicia : α := g (f x)
```

Pre doplnenie dodávame že nie je nutné ani definovať návratový typ α respektíve β , explicitné určenie návratového typu je ale vhodné nie len pre čitateľa ale aj pre overenie správnosti výsledného typu.

3.5 Kontext, priestor mien

Pre ovládanie menného priestoru a kontextu existujú v Lean-e rôzne mechanizmy z ktorých si popíšeme len tie najdôležitejšie. Najjednoduchším je súbor, ktorého definície, konštanty a premenné vstupujú do kontextu pri deklarácii a na konci súboru zaniknú. Import obsahu iných súborov sa robí jednoduchým príkazom `import` ktorý musí byť deklarovaný na začiatku súboru. V súborovej hierachii sa potom od najvyššej úrovne v jednej z vyhľadávaných ciest vnoríme cez bodku.

```
import order.lattice
```

Sa snaží nájsť priečinok *order* so súborom *lattice.lean*. V aktuálnej verzii Lean-u v čase písania práce sa dá zistiť zoznam prehľadávaných ciest prepínačom *path* priamo programu *lean*.

Deklarácie premenných definované v argumentoch definície prepisujú vonkajší kontext. V prípade žeby sme chceli rozdeliť kontext súboru na menšie sekcie jednoduchým nástrojom je dvojica *section* *nazov_sekcie*, *end* *nazov_sekcie*.

```
section prirodzene_cisla
  variable  $\alpha$  :  $\mathbb{N}$ 
end prirodzene_cisla

section cele_cisla
  variable  $\alpha$  :  $\mathbb{Z}$ 
end cele_cisla
```

Používanejším je menný priestor *namespace* *nazov* ktorý po zedefinovaní ponúka možnosť opätovného zavedenia kontextu a vnorenie podobne ako rovnomenný mechanizmus v jazyku C++.

```
universe u

namespace skryty

  variable ( $\alpha$  : Type u)
  namespace priestor
    def identita (a :  $\alpha$ ) :  $\alpha$  := a
  end priestor
end skryty

#check skryty.priestor.identita

open skryty.priestor

#check identita
```

4 Dokazovanie

K dokazovaniu v Leane je možné pristupovať viacerými spôsobmi. Po poskytnutí dôkazu už vnútorne Lean-u nezáleží na spôsobe akým bolo tvrdenie dokázané. Prvým spôsobom je priamočiara konštrukcia typu pomocou funkcií. Druhým spôsobom je spätné dokazovanie ktoré využíva taktický mód za pomoci sady príkazov upravujúcich cieľ a s pomocou poloa automatizácie. V leanovskej komunite je preferovaný práve druhý spôsob hoci pre jednoducho dokázateľné tvrdenia stačí a je využívaná aj dopredná verzia.

4.1 Dopredné dokazovanie

Dôkaz zostavený len z λ -funkcií ktoré transformujú argumenty na výsledný typ je pri zložitejších dôkazoch ťažko čitateľný. Pre lepšiu čitateľnosť poskytuje Lean nástroje ktoré sa snažia konštruovať dôkazy tak aby boli čitateľné ako tie klasické.

```
variables p d : Prop

def plati_predpoklad : p  $\rightarrow$  d  $\rightarrow$  p :=  $\lambda$  hp : p,  $\lambda$  hd : d, hp
```

Predchádzajúci príklad je možné prepísať do čitateľnejšej verzie


```
def plati_predpoklad : p → d → p :=
  assume hp : p,
  assume hd : d,
  show p, from hp
```

Pre lepšiu predstavu toho čo sa deje na pozadí Lean poskytuje výstup:

```
pq: Prop
hp: p
hq: q
├ p
```

Funkcie sú asociatívne zprava $p \rightarrow (d \rightarrow p)$. Tvrdenie hovorí ak platí p , potom z d vyplýva p .

- Predpokladajme hp ,
- a predpokladajme hd ,
- potom vieme ukázať že platí p z predpokladu hp .

O čosi zložitejším príkladom je tvrdenie že ak platí p konjunkcia q tak potom platí q konjunkcia p .

```
def symetria_konjukcie : p ∧ q → q ∧ p :=
  assume hpq : p ∧ q,
  have hp : p := and.left hpq,
  have hq : q := and.right hpq,
  have hqp : q ∧ p := and.intro hq hp,
  show q ∧ p, from hqp
```

Konjunkciu v editore je možné napísať cez `|and`. Ak sa pozrieme na typ konjukcie a funkcií ktoré sme dostali cez príkaz `check`. Dostaneme výstup:

```
and : Prop → Prop → Prop
and.left : ?M_1 ∧ ?M_2 → ?M_1
and.right : ?M_1 ∧ ?M_2 → ?M_2
and.intro : ?M_1 → ?M_2 → ?M_1 ∧ ?M_2
```

Toto intuitívne korešponduje s tým čo od týchto funkcií očakávame. Tento dôkaz je rozšírený o príkaz `have` ktorý vytvára predpoklad z existujúcich. V poslednej fáze dôkazu je interaktívny výstup:

```
pq: Prop
hpq: p ∧ q
hp: p
hq: q
hqp: q ∧ p
├ q ∧ p
```

4.2 Spätné dokazovanie

V doprednom dokazovaní sme sa snažili transformovať predpoklady tak aby na konci bol výsledkom dôsledok. Ako evokuje názov v spätnom dokazovaní sa snažíme transformovať cieľ tak aby sme sa dostali k jednému z predpokladov. Pre tento účel existuje v Lean-e špeciálny spôsob dokazovania ktorý nazývame taktický mód. Na rozdiel od dopredného dokazovania je nutné ovládať väčšiu sadu príkazov ktoré priamo transformujú cieľ. Dokazovanie je tak užšie prepojené s interaktívnym prostredím a pripomína tak hru. Ďalšou výhodou tohto dokazovania je možnosť využitia umelej inteligencie pri vyhľadani dôkazu. Takýto prístup je obzvlášť užitočný napríklad v prípade že musíme dokázať identitu

ktorej dôkaz je prácný, a tvorí ho veľa za sebou nasledujúcich využití iných identít ako napríklad využitie komutativity alebo asociativity. Užšie prepojenie s prostredím a automatizácia vyhľadania dôkazu tohto módu ale ide na úkor priamej čitateľnosti. Pre úplnosť dopĺňame že v taktickom móde nie je nutné transformovať cieľ a je možné využiť aj konštrukcie dopredného dôkazu.

Pre dokazovanie v taktickom móde používame konštrukciu ktorá je ohraničená kľúčovými slovami *begin* a *end*.

```
def symetria_konjukcie : p ∧ q → q ∧ p :=
  begin
    intro h,
    cases h with p q,
    split,
    exact q,
    exact p
  end
```

Dôkaz začína zavedením predpokladu z implikácie k ostatným o ktorých tvrdíme že sú pravdivé príkazom *intro* a rozdelením konjukcie príkazom *cases* s argumentom zavedenej konjukcie a ich explicitným pomenovaním. Výstup v informačnom okne potom obsahuje.

```
pq: Prop
p: p
q: q
├ q ∧ p
```

Príkazom *split* potom rozdelíme cieľ na podciele kde sa snažíme dokázať ľavú a pravú stranu ktorú vidíme v informačnom okne ako:

```
2 goals
pq: Prop
h_left: p
h_right: q
├ q
pq: Prop
h_left: p
h_right: q
├ p
```

Dôkaz potom uzatvoríme príkazom *exact* ktorý ukáže na jeden z predpokladov s rovnakým typom ako cieľ.

5 Abstraktné štruktúry v Lean-e

V matematike a tak isto aj v programovaní sa často vyskytujú objekty nad ktorými je možné vykonávať operácie bez ohľadu na ich obsah pri splnení určitých podmienok. V matematike hovoríme o štruktúrach, ktoré majú striktnú definíciu v programovaní sa dá voľne hovoriť o triedach. Matematické štruktúry v Lean-e su navrhované práve pomocou mechanizmu tried, pričom definícií triedy zodpovedá štruktúra spĺňajúca z definície vyplývajúce podmienky. Potom inštanciou triedy môže byť znova štruktúra ktorá je definovaná na užšej množine znova spĺňajúcej podmienky, príkladom je vzťah

grupy a podgrupy alebo priestoru a podpriestoru. Druh hierarchie potom tvorí možnosť rozširovania tried pri ktorých spĺňa okrem podmienok pôvodnej štruktúry aj ďalšie ktoré ju obohacujú. Príkladom využitia môže byť pologrupa ktorá je množinou s asociatívnou operáciou a monoidom, čo je pologrupa s jednotkou.

5.1 Induktívne štruktúry

Induktívne štruktúry predstavujú notačne silný nástroj pre definíciu objektov medzi ktorými je jednoduchá asocioácia. Jednoduchým príkladom je zoznam kde asociáciou medzi prvkami tvorí vzťah usporiadania. Komplikovanejší klasický príklad tvoria stromové štruktúry tvoriace uzly s viacerými nasledovníkmi. V matematike sa s indukciou stretávame pri definícii prirodzených čísel cez Peanove axiomy. Dve z týchto axióm hovoria:

- 0 je číslo
- Ak a je číslo potom nasledovník a je tiež číslo.

Prvé pravidlo hovorí o základnom prípade, v prípade zoznamu alebo stromu prázdny zoznam respektíve strom. Implementácia prirodzených čísel pomocou indukčnej štruktúry by mohla vyzeráť nasledovne:

```
inductive prirodzene : Type
| nula      : prirodzene
| nasledovnik : prirodzene → prirodzene
```

Za kľúčovým slovom `inductive` nasleduje názov a sada konštruktorov s návratovými typmi. Už z názvu vyplýva súvislosť s dôkazom s matematickou indukciou kde platnosť nejakého tvrdenia $P(n)$ pre všetky prirodzené čísla sa dokazuje pre

- $P(0)$, t.j. základný prípad
- $\forall n \in \mathbb{N}, P(n) \implies P(n+1)$, indukčný krok

Pre tento typ dôkazu existuje v Leane v taktickom móde príkaz `induction`.

```
example : 5 * n = 4 * n + n :=
begin
  induction n with d hd,
  ...
end
```

Po príkaze `induction` sa informačnom nachádzajú dva ciele pre obe konštruktory prirodzených čísel `nat.zero` a `nat.succ` podobne ako v našej uvedenej štruktúre a základnému prípadu a tvrdeniam matematickej indukcie dokazujúcej platnosť P .

```
case nat.zero
| 5 * 0 = 4 * 0 + 0

case nat.succ
d: ℕ
hd: 5 * d = 4 * d + d
| 5 * d.succ = 4 * d.succ + d.succ
```

V príklade ktorý sme uviedli sú znamienka $+$ a $*$ ktoré používame v operáciach nad typom našej indukčnej štruktúry. Výsledkom oboch binárnych operácií je znova prirodzené číslo. Podľa predpokladov typ ktorý prislúcha ich definícii je *prirodzene* \rightarrow *prirodzene* \rightarrow *prirodzene*. Jednou z výhod indukčne definovaných štruktúr je jednoduchosť definovania rekurzívnych funkcií ktoré operujú nad týmito štruktúrami.

```
variables (a b: prirodzene)

open prirodzene

def sucet : prirodzene  $\rightarrow$  prirodzene  $\rightarrow$  prirodzene
| a      nula      := a
| a (nasledovnik b) := nasledovnik (sucet a b)
```

V tomto prípade znamená notácia $|$ prípad v ktorom sa snažíme nájsť zhodu s poskytnutým argumentom a návratový typ. V treťom prípade ide o rekurzívne pravidlo ktoré z argumentu b vytvorí jedo predchodcu. Celé pravidlo potom vráti nasledovníka zo súčtu prvého argumentu a predchodcu druhého. Rekúzia tak prenáša vlastnosť nasledovania z prvého na druhý argument až sa argumenty nezhodujú s prvým pravidlom a teda argument už nemá predchodcu. Pre úplnosť dodávame aj technický detail definovania znamienka $+$ ako volania funkcie súčtu:

```
local infix `+` := sucet
```

5.2 Jednoduché štruktúry

V prípade že by jeden z konštruktorov neobsahoval typ ktorý poskytuje funkciu medzi typmi samotnej štruktúry, ide o obyčajnú štruktúru. Ich využitie je najmä pri definovaní tých matematických a v lean-e majú špeciálnu syntax. V príklade uvádzame ekvivalentné spôsoby definovania štruktúr.

```
structure vector :=
  mk :: (x y: ℝ)

structure vector₂ :=
  (x : ℝ)
  (y : ℝ)
```

Vytváranie objektov danej štruktúry má explicitné priradzovanie prvkom podľa názvov v prípade jednoduchých štruktúr je vhodnejšie priradzovanie na základe poradia z definície štruktúry.

```
def v₂ : vector₂ :=
{
  x := 10.5,
  y := 13.7,
}

#check (<10.5, 13.7> : vector)
```

K jednotlivým prvkom štruktúry pristupujeme cez bodku:

```
#eval v.x
```

tak vracia hodnotu 10.5. Pre modelovanie hierarchie kde jedna štruktúra rozširuje druhú používame kľúčové slovo *extends*. Rozšírenie našej štruktúry o ďalšiu dimenziu vyzerá nasledovne:

```
structure vector₃ extends vector₂ :=
  (z : ℤ)
```

```
def V31 : vector3 :=
  { x := 1, y := 2, z := 3 }
```

V prípade že chceme vytvoriť objekt z rozšírenej štruktúry a máme k dispozícii jeho podkladový je často využívaná notácia.

```
def V32 : vector3 :=
{
  z := 10, ..v2
}
```

Závislostné typy predstavujú mechanizmus parametrizácie štruktúr a definícií na základe poskytnutého typu. V prípade nášho vektora by sme túto vlastnosť mohli využiť ak by sme chceli zúžiť množinu z ktorej pochádza náš vektora len na celé čísla.

```
structure vector { α : Type u } :=
  (x : α)
  (y : α)
  (z : α)

def v1 := (⟨10, 13, 3⟩ : vector)
def v2 := (⟨(-10 : ℤ), 13, 3⟩ : vector)

#check v1.z
#check v2.z
```

Pri prvej kontrole sme dostali typ \mathbb{N} . V druhom prípade je ale z -tová zložka vektora v_2 typu \mathbb{Z} . Nedostatkom uvedeného príkladu je že množina \mathbb{Z} obsahuje množinu \mathbb{N} . Príkladom ktorý lepšie ilustruje možnosti využitia sú matematické štruktúry, ktoré sú definované na množinách ktoré môžu obsahovať objekty ľubovoľného typu.

```
variables { T : Type* } ( A : šútruktra1 T ) ( B : šútruktra2 T )
```

Takáto definícia premenných A a B zase zaručuje že množina na ktorých sú štruktúry definované je rovnakého typu. V prípade že by sme výraz uzátvorkovali do obyčajných zátvoriek použili v definícii:

```
lemma ( T : Type* ) ( A : šútruktra1 T ) ( B : šútruktra2 T ) : Type :=
begin
  ...
end
```

Musíme pri použití alebo zavolaní poskytnúť aj prvý argument. Z tohto dôvodu sa tejto konštrukcii hovorí aj implicitný argument.

```
lemma { T : Type* } ( A : šútruktra1 T ) ( B : šútruktra2 T ) : Type :=
begin
  ...
end
```

Typy na ktorých sú definované argumenty závisia od implicitného.

5.3 Typové triedy

Typové triedy poskytujú mechanizmus pomocou ktorého zlučujeme rozdielne štruktúry s rovnakou vlastnosťou do tried. Príkladom môžu byť mať vlastnosť usporiadania, mať reprezentanta alebo byť grupou. Po zdefinovaní triedy potom štruktúru priradíme do tried inštancovaním. Inštancovanie

pre štruktúry potom býva odlišné v závislosti definície štruktúry. Vykonávanie dôkazov nad typovou triedou potom umožňuje zredukovať množstvo definícií. Nad našimi definíciami vektorov môžeme definovať dĺžku.

```
class norm (α : Type u) :=
  (length : α → ℝ)
```

Potom k definíciám štruktúr pridáme inštancie.

```
instance vector2_norma : norma vector2 :=
  { length := λ (v : vector2), sqrt(v.x * v.x + v.y * v.y) }

instance vecotr3_norma : norma vector3 :=
  { length := λ (v : vector3), sqrt(v.x * v.x + v.y * v.y + v.z * v.z) }
```

Triedu potom využívame v definíciách uvedením do hranatých zátvoriek aj s argumentom.

```
variables { T : Type* } ( a : šútruktra1 T ) ( b : šútruktra2 T ) [ norma T ]
```

Tento už komplikovanejší zápis hovorí o premenných a a b ktoré sú definované na rovnakom type ktorý má definovanú alebo je inštancovaný dĺžkou.

Kapitola 4

Teória čiastočného usporiadania

V tejto kapitole sa budeme snažiť ukázať možnosti Lean-u a využitie už existujúcich definícií v mathlibe pre dokázanie viet týkajúcich sa teórie čiastočného usporiadania. Pre tento účel je v Lean-e možnosť definovania vlastností usporiadania, ktoré následne možno aplikovať na abstraktnú množinu objektov. Výsledný typ je potom odvodený na základe závislostných typov. Usporiadanie je jednoducho intuitívne pochopiteľná vlastnosť bez matematických predispozícií. V každodennom živote porovnávame svoju výšku, čas, ktorý trval na vybehnutie do kopca alebo aj číselne neohodnotené, subjektívne merateľné objekty ako ktorý album od skupiny preferujem. Na otázky si potom vieme odpovedať "ja som vyšší", "zabehol si pomalšie alebo tieto albumy sú neporovnateľné". Teória usporiadania sa snaží tieto vlastnosti formálne definovať a rozvíjať ďalej otázkami ako,

- Aké je horné celej množiny objektov?
- Existuje ohraničenie horné alebo dolné pre ľubovoľnú podmnožinu objektov?
- Ako vyzerá zobrazenie zachovávajúce usporiadanie?

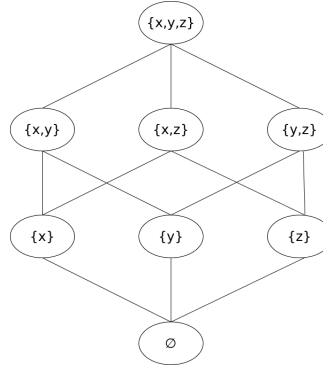
Pre stručnosť sa v rámci definícií obmedzíme len na definíciu relácie usporiadania, čiže podmnožinu karteziánskeho súčinu dvoch množín.

Definícia 4.1. *Majme množinu P , potom usporiadanie alebo čiastočné usporiadanie na množine P je binárna relácia \leq taká že, pre všetky $x, y, z \in P$*

- $x \leq x$ *reflexivita*
- $x \leq y$ a $y \leq x$ *implikuje* $x = y$ *antisymetria*
- $x \leq y$ a $y \leq z$ *implikuje* $x \leq z$ *tranzitivita*

Ideálnym nástrojom pre uvažovanie nad usporiadaním sú *Hasseho* diagramy. Ako príklad uvádzame diagram "kocky".

Na obrázku je usporiadanie všetkých podmnožín trojprvkovej množiny $\{a, b, c\}$ vzhľadom na kardinálitu podmnožín. Za porovnateľné považujeme len tie prvky, ktoré sú "pokryté" jednosmernou cestou cez orientované hrany grafu.

Obr. 4.1: Usporiadania $\mathcal{P}(\{a, b, c\})$

V Leane je usporiadanie definované ako rozšírenie triedy predusporiadania, ktorá je reláciou, ktorá nemá oproti čiastočnému usporiadaniu vlastnosť antisymetrie.

```
class has_le (α : Type u) := (le : α → α → Prop)
class has_lt (α : Type u) := (lt : α → α → Prop)

class preorder (α : Type u) extends has_le α, has_lt α :=
  (le_refl : ∀ a : α, a ≤ a)
  (le_trans : ∀ a b c : α, a ≤ b → b ≤ c → a ≤ c)
  (lt := λ a b, a ≤ b ∧ ¬ b ≤ a)
  (lt_iff_le_not_le : ∀ a b : α, a < b ↔ (a ≤ b ∧ ¬ b ≤ a) . order_laws_tac)
```

Čiastočné usporiadanie je potom rozšírením predusporiadania o vlastnosť antisymetrie.

```
class partial_order (α : Type u) extends preorder α :=
  (le_antisymm : ∀ a b : α, a ≤ b → b ≤ a → a = b)
```

1 Zväz

Zväz je čiastočne usporiadaná množina, pre ktorú navyše platí, že pre každé 2 prvky a, b vieme nájsť prvok c , ktorý je ich jedinečným najmenším horným, respektíve (*supremum*) najväčším dolným ohraničením (*infimum*). V prípade intervalu reálnych čísel je toto ohraničenie jednoducho predstaviteľné ako bod ohraničujúce množinu na číselnej osi. Ak ide o čiastočné usporiadanie, názov je pre tieto ohraničenia prvkov motivovaný zobrazením na grafe. *Spojenie* \sqcup, \vee pre supremum, respektíve *priesek* \sqcap, \wedge pre infimum. Popisnejším názvom pre zväz je preklad anglicky používaného názvu *lattice* "mriežka" tak isto motivovaná zobrazením takého usporiadania na grafe. Pri dokazovaní viet o zväzoch je často využívaná vlastnosť duality najmenšieho horného a duálne najväčšieho dolného ohraničenie pre druhú polovicu dôkazu.

```
class has_sup (α : Type u) := (sup : α → α → α)
class has_inf (α : Type u) := (inf : α → α → α)

infix ⊔ := has_sup.sup
infix ⊓ := has_inf.inf

class semilattice_sup (α : Type u) extends has_sup α, partial_order α :=
  (le_sup_left : ∀ a b : α, a ≤ a ⊔ b)
  (le_sup_right : ∀ a b : α, b ≤ a ⊔ b)
```



```

(sup_le : ∀ a b c : α, a ≤ c → b ≤ c → a ⊔ b ≤ c)

class semilattice_inf (α : Type u) extends has_inf α, partial_order α :=
  (inf_le_left : ∀ a b : α, a ⊓ b ≤ a)
  (inf_le_right : ∀ a b : α, a ⊓ b ≤ b)
  (le_inf : ∀ a b c : α, a ≤ b → a ≤ c → a ≤ b ⊓ c)

class lattice (α : Type u) extends semilattice_sup α, semilattice_inf α

```

2 Modulárne zväzy

V nasledujúcom úseku si ukážeme vetu týkajúcu sa špeciálneho typu zväzu s vlastnosťou modularity a ukážeme si formálny dôkaz a jej implementáciu v Leane, ktorú si podrobne rozoberieme.

O zväze L hovoríme, že je modulárny, v prípade, že má nasledujúcu vlastnosť.

$$(\forall x, y, z \in L) x \geq y \implies x \wedge (y \vee z) = (x \wedge y) \vee z$$

V Leane definovaný ako rozšírenie zväzu:

```

class modular_lattice(α : Type u) extends lattice α :=
  (modular_law : ∀ (x u v : α), (x ≤ u) → u ⊓ (v ⊔ x) = (u ⊓ v) ⊔ x)

```

V nasledujúcom úseku si ukážeme vetu o izomorfizme pre modulárne zväzy a podrobne si rozoberieme implementáciu jej dôkazu s obsahom prostredia v Leane.

Veta 4.1. Veta o izomorfizme pre modulárne zväzy *Nech L je modulárnym zväzom a $a, b \in L$. Potom*

$$\varphi_b : x \mapsto x \wedge b, x \in [a, a \vee b], \quad (4.1)$$

Je izomorfizmom medzi intervalmi $[a, a \vee b]$ a $[a \wedge b, b]$. Inverzným izomorfizmom je

$$\psi_a : y \mapsto y \vee a, y \in [a \wedge b, b]. \quad (4.2)$$

Dôkaz. Stačí ukázať, že $\varphi_b \psi_a(y) = y$ pre všetky $y \in [a \wedge b, b]$. Z duality potom vyplýva, že $\psi_a \varphi_b(x) = x$ pre všetky $x \in [a, a \vee b]$, Nech $y \in [a \wedge b, b]$, potom $\varphi_b \psi_a(y) = (y \vee b) \wedge a$ a ak platí nerovnosť $b \geq y$ tak potom z modularity

$$\varphi_b \psi_a(y) = (y \vee b) \wedge a = y \vee (b \wedge a) = y \quad (4.3)$$

pretože

$$y \geq a \wedge b.$$

□

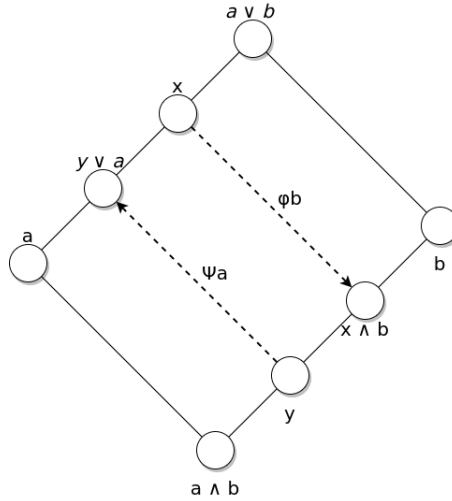
Horeuvedený dôkaz je znázornený na nasledujúcom obrázku.

k V prípade formálneho dôkazu sme sa mohli v časti dôkazu odkázať na dualitu. Pri návrhu dôkazu v Leane musíme ukázať dôkaz z oboch strán. Najskôr uvedieme hotový dôkaz vety:

```

theorem modular_lattice_isomorphism { α : Type u } [ modular_lattice α ] { a b x : α
  } :
  x ≤ a →
  x ≥ b →
  x ≥ a ⊓ b →

```



Obr. 4.2: Izomorfizmus modulárneho zväzu

```

x ≤ a ∨ b →
a ∧ (b ∨ x) = x ∧ (a ∧ x) ∨ b = x
:=
begin
  intros h1 h2 h3 h4,
  split,
  {
    rw modular_lattice.modular_law,
    exact sup_eq_right.mpr h3,
    exact h1
  },
  {
    rw inf_comm,
    rw ← modular_lattice.modular_law,
    exact inf_eq_left.mpr h4,
    exact h2
  }
end

```

Začínáme v taktickom móde ktorý je interaktívnou verziou dokazovania v Leane. Po zadaní prázdnej konštrukcie *begin* a *end* Interaktívne prostredie vyzerá nasledovne.

```

α: Type u
_inst_1: modular_lattice α
abx: α
⊢ x ≤ a → x ≥ b → x ≥ a ∧ b → x ≤ a ∨ b → a ∧ (b ∨ x) = x ∧ a ∧ x ∨ b = x

```

Prvým krokom dôkazu je presunutie predpokladov zo sledu implikácií do prostredia pre ďalšiu prácu s nimi s označením *h1, h2, h3, h4*.

```

α: Type u
_inst_1: modular_lattice α
abx: α
h1: x ≤ a
h2: x ≥ b
h3: x ≥ a ∧ b
h4: x ≤ a ∨ b
⊢ a ∧ (b ∨ x) = x ∧ a ∧ x ∨ b = x

```

Cieľ potom pozostáva z konjunkcie, kde v druhej časti máme výraz implicitne ozátvorkovaný zľava. Výraz rozdelíme do dvoch podcieľov príkazom *split*, a pre lepšiu čitateľnosť ozátvorkujeme množinovými

zátvorkami. Nachádzame sa v stave

```
begin
  intros h1 h2 h3 h4,
  split,
  {
  },
  {
  }
end
```

v ktorom nám lean ukazuje prostredie, kde musíme dokázať ľavú časť konjunkcie.

```
⊢ a ∧ (b ∨ x) = x
```

Na cieľ použijeme z definície modulárneho zväzu vlastnosť modularity

```
(modular_law: ∀ (x u v : α), (x ≤ u) → u ∧ (v ∨ x) = (u ∧ v) ∨ x)
```

a transformujeme prepíšeme cieľ cez príkaz

```
rw modular_lattice.modular_law,
```

na nasledujúci, kde má $u \wedge v$ vyššiu precedenciu

```
⊢ u ∧ v ∨ x = x
```

Nasledujúca transformácia vyžaduje znalosť už dokázaných definícií, ktoré boli dokázané pre podkladové štruktúry. Použijeme nasledujúcu definíciu, ktorá vychádza z kontextu *semilattice_sup*.

```
% @ [simp] theorem sup_eq_right : a ∨ b = b ↔ a ≤ b := / TODO NEZABUDNUT
% le_antisymm_iff.trans $ by simp [ le_refl ] / ODKOMENTOVAT
```

Zaujímavosťou je, že si Lean dokáže substitúovať výraz $u \wedge v$ za a z uvedeného výrazu. Pri použití vety dostávame ekvivalenciu, ktorá je definovaná ako štruktúra.

```
structure iff (a b : Prop) : Prop :=
  intro :: (mp : a → b)
          (mpr : b → a)
```

Z tejto štruktúry použijeme implikáciu smerujúca doľava nasledovne:

```
exact sup_eq_right.mpr h3,
```

Cieľ je teda transformovaný na:

```
⊢ x ≤ a
```

čo je už uvedený predpoklad $h1$. Týmto sme dokázali jeden z podcieľov. V tejto chvíli by sme sa v literatúre mohli odvolať na dualitu výrazov. V Leane musíme poskytnúť dôkaz aj o druhom cieľi.

Ideme dokázať

```
⊢ a ∧ x ∨ b = x
```

V tejto chvíli chceme znova použiť modularitu, leanu je, ale potrebné explicitne povedať, že chceme prepísať výraz nachádzajúci na pravej strane rovnosti pomocou symbolu ľavej šípky.

```
rw ← modular_lattice.modular_law,
```

Použijeme duálnu vetu duálnu k *sup_eq_right*.

```
@[simp] theorem inf_eq_left : a ∧ b = a ↔ a ≤ b
```

a využijeme opačné predpoklady k predchádzajúcim $h2, h4$.

```
{
  rw ← modular_lattice.modular_law,
  exact inf_eq_left.mpr h4,
  exact h2
}
```

Po dokázaní druhého cieľa sme dokázali celú vetu. \square

3 Podzváz

V rámci príspevku do knižnice *mathlib* sme implementovali štruktúru reprezentujúcu podzváz. Formálne je definovaný:

Veta 4.2. *Nech L je zväz. Množina $K \subseteq L$ je podzváz L ak pre všetky $a, b \in K$ platí, že, $a \wedge b \in K$, $a \vee b \in K$.*

```
import order.lattice

structure sublattice ( L : Type* ) [ lattice L ] :=
  ( carrier : set L )
  ( inf_mem' {a b : L} : a ∈ carrier → b ∈ carrier → a ⊓ b ∈ carrier )
  ( sup_mem' {a b : L} : a ∈ carrier → b ∈ carrier → a ⊔ b ∈ carrier )
```

Pre definíciu podzvazu je nutné zaviesť priestor mien pre zväzy ktorý sa v knižnici *mathlib* v súbore *"src/order/lattice.lean"*. Z tohto priestoru používame typovú triedu zväzu. V prípade že chceme vytvoriť typ *sublattice* je nutné poskytnúť typ ktorý má inšancovanú typovú triedu zväzu. Prvky štruktúry potom tvorí *carrier* v preklade nosič ktorého typ *set* je $L \rightarrow Prop$ a prvky *inf_mem* a *sup_mem* zaručujúce uzavretosť suprema a infima. Nasledujúce definície a inšancie sú definované v priestore mien *sublattice* Pre použitie tejto štruktúry sme štruktúru inštancovali pre triedu *set_like* ktorá poskytuje možnosť konverzie typu na jednoduchšie štruktúry.

```
instance : set_like (sublattice L) L :=
  ⟨sublattice.carrier, λ p q h, by cases p; cases q; congr'⟩
```

Definície označené značkou *simp* slúžia na zjednodušovanie výrazov pomocou príkazu *simp* taktického módu.

```
@[simp]
lemma mem_carrier {SL : sublattice L} {x : L} : x ∈ SL.carrier ↔ x ∈ SL := iff.rfl
```

V tomto prípade hovorí ide o ekvivalenciu tvrdení že prvok patriaci nosiču štruktúry je ekvivalentný tvrdeniu že patrí štruktúre.

Definície označené značkami *ext* potom označujú definície ktoré používa príkaz *exact*. Vo všeobecnosti ide o definície ktoré hovoria o tom kedy sú 2 typy rovnaké.

```
@[ext]
theorem ext {A B : sublattice L}
  (h : ∀ x, x ∈ A ↔ x ∈ B) : A = B := set_like.ext h
```

Nasledujúce definície sa týkajú vytvorenia nového objektu kopírovaním, vytvorením kópie cez pretypovanie a tvrdením a definíciou ktorá vráti typ tvrdenia o rovnosti kópie so štruktúrou podzvazu.

```

def copy (SL : sublattice L) (S : set L) (hs : S = SL) : sublattice L :=
{ carrier := S,
  inf_mem' := hs.symm ► SL.inf_mem',
  sup_mem' := hs.symm ► SL.sup_mem' }

variables { SL : sublattice L }

lemma coe_copy {B : sublattice L} {A : set L} (hs : A = SL) :
  (SL.copy A hs : set L) = A := rfl

lemma copy_eq {s : set L} (hs : s = SL) : SL.copy s hs = SL :=
  set_like.coe_injective hs

```

V prípade že by sme chceli prístup k definícii typu infima alebo superma našej štruktúry boli zavedené definície ktoré vracajú ich typ.

```

theorem inf_mem {a b : L} : a ∈ SL → b ∈ SL → a ⊓ b ∈ SL :=
  sublattice.inf_mem' SL

theorem sup_mem {a b : L} : a ∈ SL → b ∈ SL → a ⊔ b ∈ SL :=
  sublattice.sup_mem' SL

```

Literatúra

- [1] Samuel Mimram, Program = Proof, Independently published(July 3, 2020), ISBN-13: 979-8615591839
- [2] Morten Heine B. Sørensen, Pawel Urzyczyn, Lectures on the Curry-Howard Isomorphism, Elsevier Science (April 4, 2013), ISBN-13 : 978-0444545961
- [3] <https://github.com/leanprover/lean>
- [4] <https://github.com/leanprover/lean4>
- [5] <https://github.com/leanprover-community/mathlib>
- [6] <https://leanprover-community.github.io/papers/mathlib-paper.pdf>