FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Interaktivní dokazovací asistent pro výuku formálních metod
Student:	Bc. Martin Bedrna
Vedoucí:	doc. Dipl.-Ing. Dr. techn. Stefan Ratschan
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem práce je vytvořit program, pomocí kterého mohou studenti grafickým způsobem dokázat věty v predikátové logice prvního řádu. V každé dokazovací situaci má program studentům nabízet veškerá pravidla, která se v dané dokazovací situaci dají použít, a student si z nich vybere ta pravidla, která ho co nejlépe vedou k cíli a případně doplní další nutné informace. Základní dokazovací kalkul bude verze přirozené dedukce, kterou používá školitel v předmětu "Formální metody a specifikace".

1) Udělejte návrh grafického rozhraní tak, aby rozhraní co nejlépe pomohlo studentům vyřešit příklady.
2) Udělejte návrh celého softwarového balíku tak, aby se po skončení projektu dal snadno udržovat a rozšířit.
3) Implementujte Váš návrh.
4) Vypracujte několik příkladů, které zdokumentují použitelnost softwaru.

Průběžně Vaše výsledky dokumentujte a na konci z Vaší dokumentace vytvořte ucelenou diplomovou práci.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 20. dubna 2023 v Praze.

Diplomová práce

INTERAKTIVNÍ DOKAZOVACÍ ASISTENT PRO VÝUKU FORMÁLNÍCH METOD

Bc. Martin Bedrna

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: doc. Dipl.-Ing. Dr.techn. Stefan Ratschan
11. ledna 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat svému vedoucímu, panu doc. Dipl.-Ing. Dr. techn. Stefanu Ratschanovi, za velmi nápomocné vedení mé diplomové práce. Jeho konstruktivní kritika mi umožnila vylepšit důležité aspekty práce.
Dále chci poděkovat své rodině a svým blízkým, že mě ve studiu vždy podporovali. Jen díky nim mohl světlo světa spatřit tento projekt.
Zvláště děkuji své přítelkyni Tereze za provedenou korekturu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 11. ledna 2024

Abstrakt

Tato práce se věnuje implementaci dokazovacího asistenta. Odvozovací pravidla dokazovacího kalkulu používaná asistentem jsou verzí přirozené dedukce. V rámci této práce je problém nejdříve představen po teoretické stránce. Následně je provedena analýza a softwarový návrh. Nakonec je popsána implementace vytvořeného softwaru.

Klíčová slova dokazovací asistent, přirozená dedukce, nástroj pro podporu výuky, React aplikace

Abstract

This thesis focuses on the implementation of a proof assistant. The inference rules of the proof calculus used by the assistant are a version of natural deduction. In the context of this work, the problem is first introduced theoretically. Subsequently, an analysis and software design are carried out. Finally, the implementation of the created software is described.

Keywords proof assistant, natural deduction, educational support tool, React app

Úvod

Dokazovací systémy teorie důkazů přináší studentům softwarového inženýrství řadu překážek. Jednou z největších z nich je, že jsou ve své podstatě nepřehledné. Studenti, kteří se teprve s danou problematikou seznamují, se snadno v důkazu mohou ztratit a pociťovat frustraci.

Aplikace, vyvinutá v rámci této práce, si klade za cíl jim toto počínání usnadnit. Tento cíl naplňuje zobrazováním dokazovacího kalkulu jako sady bloků pravidel. Tyto bloky může uživatel aplikovat na dokazovaná tvrzení a dojít tak k řešení. Celý tento proces je navržen způsobem, který je pro studenta přehledný a snadný k pochopení.

V této práci je čtenář nejdříve seznámen s teorií dokazovacích systémů. Následně jsou zanalyzována řešení, které studenti mohou pro naplnění stejného cíle používat v současnosti. Na základě této analýzy a zadání práce je vypracován softwarový návrh a prototyp aplikace. Dále je čtenář seznámen s výzvami, které implementace aplikace přinesla, a jakým způsobem byly řešeny. V závěru práce je použití aplikace ukázáno na vybraných případech.

Kapitola 1
Cíl práce

Cílem této práce je vytvořit aplikaci, která studentům umožní se lépe zorientovat v dokazovacím kalkulu používaným v předmětu *Formální metody a specifikace*. Pro naplnění tohoto cíle je nutné:

- Zanalyzovat konkurenční aplikace, které řeší stejný problém. Na základě této analýzy dojít k poznatkům, které budou prospěšné při tvorbě aplikace.
- Navrhnout a vytvořit grafické rozhraní pro dokazování, které je přehledné a srozumitelné.
- Implementovat všechna pravidla kalkulu a vkládání lemmat.
- Implementovat aplikaci způsobem, který lze snadno rozšířit a udržovat.

Pro splnění těchto cílů je nutné nejdříve prozkoumat teoretický základ dokazovacích systémů. Následně analyzovat existující aplikace, které řeší podobný problém. Na základě těchto dvou kroků a přiloženého zadání udělat softwarový návrh aplikace. Tento návrh je třeba následně implementovat a implementaci otestovat.

Kapitola 2
Teorie

V této kapitole je čtenář seznámen s teorií dokazovacích systémů. Jsou představeny základní kategorie dokazovacích systémů a jejich vlastnosti. V poslední sekci je představen dokazovací systém, který používá implementovaná aplikace.

2.1 Logika

Z pohledu softwarového inženýrství je cílem logiky vyvinout jazyk, který nám umožní formálně argumentovat a rozhodovat o situacích. Dostatečně formálně správný jazyk lze následně použít při běhu softwaru, který umožní tento myšlenkový proces algoritmizovat.

2.1.1 Výroková logika

Základním kamenem formálního jazyka výrokové logiky jsou výroky. Výrok je věta, které lze přiřadit ohodnocení, zda je pravdivá, nebo nikoliv. Příkladem takovéto věty je: „Slunce vychází na východě.“ Pro tento výrok můžeme jednoznačně určit, že se jedná o pravdu. Stejně tak věta „Druhá odmocnina čísla 16 je 5.“ je výrok, protože lze jednoznačně říct, že se jedná o nepravdu. Podobně je výrokem i věta „Je nekonečný počet dvojic prvočísel p_1 a p_2, kde $p_2 = p_1 + 2$.“. I pro tuto větu jistě platí, že se jedná o pravdu, nebo nepravdu. Bohužel nejsme schopni potvrdit (prozatím), jestli je opravdu počet po sobě jdoucích lichých čísel, která jsou prvočísla, nekonečný, či nikoliv. Přesto se jedná o výrok, pro nějž existuje pravdivostní ohodnocení. Naopak věta „Půjdeš ven?“ výrokem není. Abychom mohli softwarově rozhodovat o logické validitě je potřeba takovéto výroky zakódovat do symbolického jazyka [1, s. 1–3].

2.1.1.1 Výroková logika jako formální jazyk

▶ **Definice 2.1** (Výrokové proměnné)**.** *Nechť V je množina proměnných, pak ji můžeme nazvat množinou výrokových proměnných.*

Výrokové proměnné jsou základem pro stavbu formulí. Díky spojení výrokových proměnných lze tvořit komplikovanější logické formule.

▶ **Definice 2.2** (Výroková formule)**.** *Výroková formule je posloupnost symbolů splňující dále uvedené podmínky:*

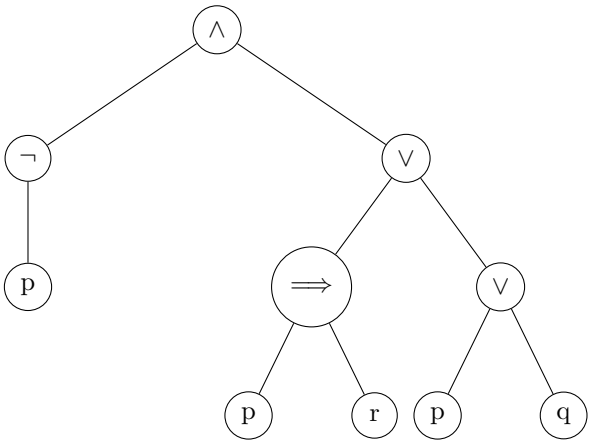
- *Jestliže v je výroková proměnná, pak je v výroková formule.*
- *Jesliže b je výroková formule, pak $\neg b$ je výroková formule.*

- *Jestliže b_1 a b_2 jsou výrokové formule, pak $b_1 \wedge b_2$, $b_1 \vee b_2$ a $b_1 \implies b_2$ jsou výrokové formule.*

Takováto definice dostačující, nicméně pro jednoznačnou interpretaci počítačem používáme závorky, které určují pořadí aplikování symbolů [2, s. 56–61].

Počítač by měl být schopen ověřit, že zadaná formule je správně formována. Toho může algoritmicky dosáhnout pomocí obrácení induktivního procesu tvorby logické formule. Pokud je formule správně zformována, je vždy možné určit, který logický symbol byl aplikován nejpozději. Pokud se jedná o logickou spojku, odstraněním tohoto symbolu se rozpadne formule na dvě podformule. V případě negace, se symbol pouze odstraní. Na těchto podformulích se opět aplikuje odstranění logického symbolu, dokud nezbude pouze atomická výroková proměnná. Tento způsob velmi přímočaře vede na vznik parsovacích stromů, kde se v uzlech nachází logické symboly a v listech atomické proměnné. Pokud po parsování formule existuje list, který není atomickou výrokovou proměnnou, formule není správně formována[1, s. 33–36].

▶ **Příklad 2.3.** Parsování formule $(\neg p) \wedge ((p \implies r) \vee (p \vee q))$.



2.1.2 Predikátová logika

Nadstavbou nad samotnou výrokovou logikou je predikátová logika. Výroková logika nemá nástroje pro zachycení jemnějších nuancí přirozeného jazyka. Větu „Každý student je mladší než nějaký instruktor.“ nelze přesně vyjádřit pouze v rámci výrokové logiky, jelikož bychom museli přehlédnout vnitřní stavbu této věty. Především zde chybí možnost vyjádřit vlastnosti a relace, které samotná věta představuje. Například můžeme zavést, že S(osoba) vyjadřuje skutečnost, že nějaká osoba je studentem. Dále lze přidat I(osoba) pro vytyčení faktu, že osoba je instruktorem. A naposledy M(první osoba, druhá osoba) označuje relaci, že první osoba je mladší než druhá osoba. Přidáním těchto symbolů jsme dosáhli větší granulity pro zachycení skutečného významu určitého výroku. Tyto symboly se nazývají predikáty. Dále je potřeba zavést symboly, které umožní se vyjadřovat obecně o daných predikátech. Abychom pro každý predikát nemuseli vypisovat konkrétní osoby, nahradíme je proměnnými. Tyto proměnné obvykle značíme x, y, z atd. Aby bylo možné vyjádřit množství osob, kterých se daný predikát týká, jsou zavedeny dva kvantifikátory. Symbol $\exists x$ označuje situaci, kdy existuje alespoň jedno x, pro které predikát platí. Symbol $\forall x$ určuje, že se predikát vztahuje na všechny proměnné x. Aby bylo možné s predikátovou logikou algoritmicky pracovat, je potřeba zavést formální jazyk [1, s. 93–107].

2.1.2.1 Predikátová logika jako formální jazyk

Základním objektem predikátové logiky je proměnná jako například x. Na proměnné se vážou tři různé množiny symbolů. $\mathcal{F}$ je množina funkčních symbolů, $\mathcal{C}$ konstantních symbolů a $\mathcal{P}$ je

množina predikátových symbolů. Vzhledem k tomu, že 0-aritní funkce jsou významově totožné s konstantami, lze jimi konstantní množinu nahradit. Spojením funkčních symbolů a proměnných vzniká term.

▶ **Definice 2.4** (Term)**.** *Term je definován následovně:*

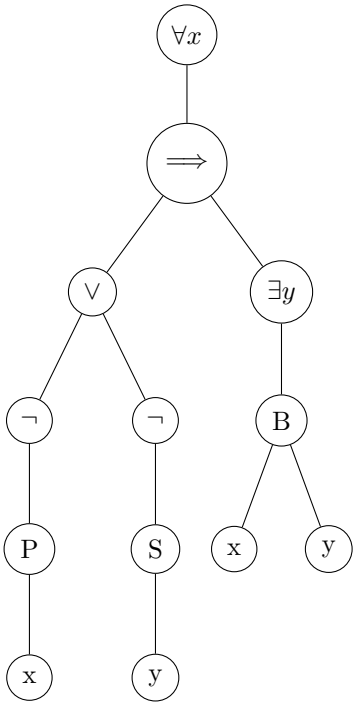
- *Každá proměnná je term.*
- *Pokud $c \in \mathcal{F}$ je 0-aritní funkce, pak c je term.*
- *Pokud $t_1, t_2, ..., t_n$ jsou termy a $f \in \mathcal{F}$ má aritu $n > 0$, pak $f(t_1, t_2, ..., t_n)$ je term.*
- *Nic jiného term není.*

S pomocí definice termu je možné přidat do jazyka predikátové logiky samotné predikáty a logické spojky. Sloučením těchto množin získáme vše potřebné pro definici logické formule v predikátové logice [1, s. 93–107].

▶ **Definice 2.5** (Formule predikátové logiky)**.** *Množina formulí je induktivně definována pomocí množin $\mathcal{F}$ a $\mathcal{P}$ a výše definované množiny termů:*

- *Jestliže $P \in \mathcal{P}$ je predikátový symbol s aritou $n > 0$ a pokud $t_1, t_2, ..., t_n$ jsou termy nad $\mathcal{F}$, pak $P(t_1, t_2, ..., t_n)$ je formule.*
- *Pokud ϕ je formule, tak $(\neg\phi)$ je formule.*
- *Pokud ϕ a ψ jsou formule, pak také $(\phi \wedge \psi)$, $(\phi \vee \psi)$ a $(\phi \implies \psi)$ jsou formule.*
- *Pokud ϕ je formule a x je proměnná, pak $(\exists\phi)$ a $(\forall\phi)$ jsou formule.*
- *Nic jiného formule není.*

▶ **Příklad 2.6.** Parsování formule $\forall x.(((\neg P(x)) \vee (\neg S(y))) \implies (\exists y.(B(x, y))))$, kde P, S, B jsou predikátové symboly.



2.1.2.2 Volné a vázané proměnné

Při tvorbě formulí v predikátové logice lze všechny proměnné významově vymezit pomocí kvantifikátorů $\exists$ a $\forall$. Nicméně někdy je potřeba proměnné nevázat k žádnému kvantifikátoru, aby proměnné byly platné za jakýchkoliv podmínek.

▶ **Příklad 2.7.** $\forall x.(x + y) = 10$

Příklad 2.7 obsahuje proměnné x a y. Proměnná x je vázaná na obecný kvantifikátor $\forall$, zatímco výskyt proměnné y není vázán na jakýkoliv kvantifikátor. Tato formule tedy říká, že pro všechna x platí, že $x + y = 10$, ať je y jakékoliv.

▶ **Definice 2.8.** *Nechť ϕ je formule v predikátové logice. Výskyt proměnné x ve formuli ϕ je volný právě tehdy, když se proměnná x nachází v listu parsovacího stromu formule ϕ, který nemá ve své cestě vzůru ke kořenu uzel $\forall x$ nebo $\exists x$. Pokud je tomu naopak je x proměnnou vázanou. Pro $\forall x\phi$, respektive $\exists x\phi$, řekneme, že ϕ je rozsah tohoto vázání, mínus podformule ψ ve tvaru $\forall\psi$ a $\exists\psi$.*

Tedy rozsah vázání kvantifikátoru proměnné je celý jeho podstrom, kromě podstromů, které opět obsahují kvantifikátory pro tu stejnou proměnnou. V příkladu 2.6 vidíme, že všechny výskyty proměnné x jsou vázány obecným kvantifikátorem v kořenu parsovacího stromu. Kdežto výskyt proměnné y je vázán pouze v pravém podstromu implikace na existenční kvantifikátor a v levém podstromu je její výskyt volný [1, s. 93–107].

2.1.2.3 Substituce

Jelikož proměnné pouze zastupují obecný obsah, lze je substituovat. K substituci je nutné přistupovat takovým způsobem, aby se význam formule jejím aplikováním nezměnil. Jelikož je substituovaná proměnná term, je možné ji opět substituovat pouze termem. Zároveň není možné nahrazovat vázané výskyty proměnných, jelikož takové výskyty nezastupují obecný obsah, ale *všechna*, nebo *některé x*.

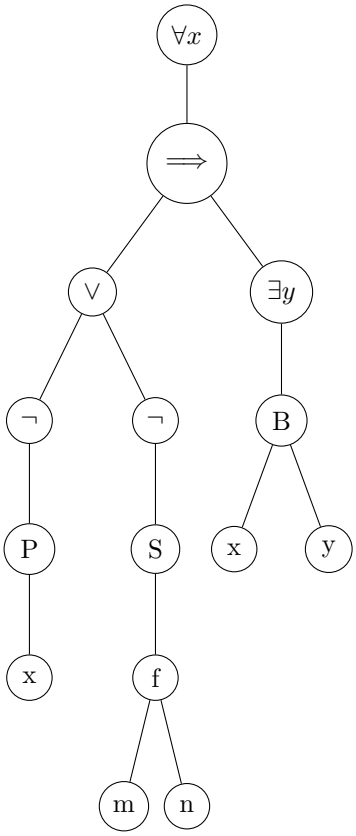
▶ **Definice 2.9.** *Pro proměnnou x, term t a formuli ϕ definujeme $\phi[t/x]$ jako formuli, která vznikne nahrazením volných výskytů proměnné x v formuli ϕ termem t.*

Význam formule po provedení substituce lze změnit ještě v případě, že substituce změní volný výskyt proměnné na výskyt vázaný. Tedy že místo, ve kterém je prováděna substituce, je v dosahu nějakého kvantifikátoru proměnné, kterou obsahuje substituující term. Taková operace není připuštěna.

▶ **Definice 2.10.** *Pro term t, proměnnou x a formuli ϕ, definujeme, že term t je volný pro x v ϕ, jestli žádný volný list x se neobjevuje v dosahu $\forall y$ nebo $\exists y$ pro jakoukoliv proměnnou y vyskytující se v t.*

Tedy za x lze dosazovat pouze term, který obsahuje jenom takové proměnné, jež nemají v cestě ke kořeni parsovacího stromu od volných výskytů x své kvantifikátory [1, s. 93–107].

▶ **Příklad 2.11.** Parsovací strom z příkladu 2.6 po aplikaci $\phi[f(m,n)/y]$



2.2 Teorie důkazů

Teorie důkazů (Proof theory) je odvětví matematické logiky, které se zabývá studiem struktury a povahy matematických důkazů. Cílem této disciplíny je analyzovat formální procesy, jakými se konstruují důkazy a jak jsou tyto důkazy organizovány. Teorie důkazů se věnuje otázkám, jakým způsobem jsou vyjádřeny matematické tvrzení ve formálním jazyce, jak lze důkazy systematizovat a jak jsou vzájemně propojeny různé metody důkazů. Zahrnuje také zkoumání vlastností dokazovacích systémů.

2.2.1 Historie

Historie teorie matematických důkazů sahá až do antického Řecka, nicméně za zakladatele moderní teorie důkazů je považován David Hilbert. Ten reagoval na nejnovější objevy své doby, které poukazovaly na špatně definované základy matematiky mnohými paradoxy. Chtěl lépe uchopit kořeny samotné matematiky vyvinutím nového systému pro matematické důkazy. Hilbert se tedy pokoušel vytvořit symbolický logický jazyk s pevně danou množinou pravidel pro manipulaci s jeho symboly. Logické a matematické věty by se poté pouze daly přeložit do takového jazyka, kde by pomocí definovaných pravidel bylo možné ověřit jejich pravdivost. Z tohoto snažení vzešlo dílo *Principia Mathematica (Whitehead, Russel, 1910-1913)*, které se snažilo tehdejší matematické vědění pomocí takového jazyka zadefinovat. Hilbert ovlivněn tímto dílem definoval Hilbertův program, který měl nalézt odpovědi na otázky teorie důkazů. U všech těchto otázek věřil, že skutečná odpověď je kladná.

Hlavní otázky byly následující:

- Je matematika kompletní neboli řešitelná? Tedy zda existuje pro každé pravdivé tvrzení důkaz.
- Je matematika konzistentní? Tedy jestli nenastane situace, kdy by bylo možné dokázat tvrzení a zároveň dokázat i jeho negaci.
- Je matematika rozhodnutelná? Tedy zda existuje algoritmus, který v konečném množství kroků pro určité tvrzení určí, zda následuje z počátečních axiomů, či nikoliv.

Druhým směrem než Hilbert, se při tvoření dokazovacích systémů vydal Gerhard Gentzen. Ten se také podílel na zodpovídání otázek Hilbertova programu. Přišel s důkazem, že pokud lze dokázat kontradikci v klasické aritmetice, lze ji dokázat i v intuicionistické aritmetice. Ve své disertační práci o logické dedukci představil dva nové dokazovací systémy - přirozenou dedukci a sekvenční kalkulus. Tyto systémy jsou v základu značně odlišné od Hilbertových axiomatických systémů. Oba tyto systémy představuji v následující sekci [3, s. 1–10].

2.2.2 Dokazovací systémy

Dokazovací systémy představují klíčový nástroj v oblasti logiky. Slouží k formálnímu ověřování platnosti logických tvrzení. V těchto systémech se systematicky manipuluje s logickými formulemi a pravidly odvozování, což umožňuje konstrukci důkazů a zajišťuje správnost logických závěrů. Hlavní funkcí dokazovacích systému je možnost vyvozovat z platných tvrzení nové logické důsledky na základě pevně daných formálních pravidel.

Nejdříve definice důkazu:

▶ **Definice 2.12.** *Nechť S je doména symbolů a Q je množina formulí nad S, pak konečná sekvence $\bar{p}$ formulí je důkaz v teorii Q právě tehdy, když pro všechny $1 \leq i \leq$ délka($\bar{p}$) platí:*

- *$\bar{p}_i$ je axiom*
- *pro nějaké $j_1, ..., j_m$, kde $m < i$, $\bar{p}$ může být odvozeno z $\bar{p}_{j_1}, ..., \bar{p}_{j_m}$ aplikací odvozovacího pravidla.*

Potom $Q \vdash p$ právě tehdy, když existuje důkaz $\bar{p}$ v Q takový, že p je poslední formule $\bar{p}$ [3, s. 1–10].

Rozlišujeme dva základní typy dokazovacích systémů.

2.2.2.1 Hilbertovy dokazovací systémy

Prvním typem jsou Hilbertovi dokazovací systémy. Jejich základem je počáteční množina logických zákonů. Tyto zákony jsou nazývány axiomy. Pro vytváření nových logických znalostí jsou použitá odvozovací pravidla. Základním znakem takovýchto systémů je, že obsahují velké množství axiomů a velmi malé množství odvozovacích pravidel. Typickým odvozovacím pravidlem je *modus ponens*. Tedy pravidlo které říká, že pokud je známo A a zároveň je známo $A \implies B$, pak je možno usoudit B. Takovýto dokazovací systém představím na několika známých kalkulech pro výrokovou logiku [3, s. 18–24].

Základním kalkulem je minimální logika. Místo zavedení substitučního pravidla jsou axiomy představeny ve formě schématu $\phi \implies (\phi \wedge \phi)$, kde ϕ zastupuje jakoukoliv formuli. Axiomy minimální logiky jsou následující:

- PL1. $\phi \implies (\phi \wedge \phi)$
- PL2. $(\phi \wedge \psi) \implies (\psi \wedge \phi)$
- PL3. $(\phi \implies \psi) \implies ((\phi \wedge \chi) \implies (\psi \wedge \chi))$

- PL4. $((\phi \implies \psi) \wedge (\psi \implies \chi)) \implies (\phi \implies \chi)$
- PL5. $\psi \implies (\phi \implies \psi)$
- PL6. $\phi \wedge (\phi \implies \psi)) \implies \psi$
- PL7. $\phi \implies (\phi \vee \psi)$
- PL8. $(\phi \vee \psi) \implies (\psi \vee \phi)$
- PL9. $((\phi \implies \chi) \wedge (\psi \implies \chi)) \implies ((\phi \vee \psi) \implies \chi)$
- PL10. $((\phi \implies \psi) \wedge (\phi \implies \neg\psi)) \implies \neg\phi$

Intuicionistická logika používá stejnou množinu axiomů, navíc přidává axiom:

- PL11. $\neg\phi \implies (\phi \implies \psi)$

A nakonec klasická logika doplňuje do stejné množiny svůj vlastní axiom:

- PL12. $\neg\neg\phi \implies \phi$

Všechny tyto logické kalkuly používají jediné odvozovací pravidlo a to *modus ponens* [3, s. 18–24].

▶ **Definice 2.13.** *Derivace v minimální, intuicionistické a klasické logice je konečná sekvence formulí $A_1, ..., A_n$, kde každá z nich je axiom, nebo formule odvozená z předchozích formulí pravidlem modus ponens. Formuli A_n nazýváme konečnou formulí [3, s. 20].*

▶ **Definice 2.14.** *B je teorém v minimální, intuicionistické a klasické logice, pokud v nich existuje derivace s kroky $A_1, ..., A_n$, kde B je konečnou formulí. Zároveň řekneme, že B je dokazatelné v dané logice. Tuto skutečnost označíme $A_1, ..., A_n \vdash B$ [3, s. 20].*

Aplikaci takovéhoto dokazovacího systému Hilbertova typu lze ukázat na příkladu dokazování $p_1 \implies (p_2 \vee p_1)$.

▶ **Příklad 2.15.** Důkaz $p_1 \implies (p_2 \vee p_1)$, zdroj: [3, s. 20].

1. $\vdash p_1 \implies (p_1 \vee p_2)$ (axiom PL7)
2. $\vdash (p_1 \implies (p_1 \vee p_2)) \implies$ $(((p_1 \vee p_2) \implies (p_2 \vee p_1)) \implies (p_1 \implies (p_1 \vee p_2)))$ (axiom PL5)
3. $\vdash ((p_1 \vee p_2) \implies (p_2 \vee p_1)) \implies (p_1 \implies (p_1 \vee p_2))$ (MP 1,2)
4. $\vdash (((p_1 \vee p_2) \implies (p_2 \vee p_1)) \implies (p_1 \implies (p_1 \vee p_2))) \implies$ $(((p_1 \vee p_2) \implies (p_2 \vee p_1) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1))) \implies$ $((p_1 \implies (p_1 \vee p_2)) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1))))$ (axiom PL3)
5. $\vdash ((p_1 \vee p_2) \implies (p_2 \vee p_1) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1))) \implies$ $((p_1 \implies (p_1 \vee p_2)) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1)))$ (MP 3,4)
6. $\vdash ((p_1 \vee p_2) \implies (p_2 \vee p_1))$ $\implies (((p_1 \vee p_2) \implies (p_2 \vee p_1)) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1)))$ (axiom PL1)
7. $\vdash (p_1 \vee p_2) \implies (p_2 \vee p_1)$ (axiom PL8)
8. $\vdash ((p_1 \vee p_2) \implies (p_2 \vee p_1)) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1))$ (MP 6,7)
9. $\vdash (p_1 \implies (p_1 \vee p_2)) \wedge ((p_1 \vee p_2) \implies (p_2 \vee p_1))$ (MP 5,8)
10. $\vdash ((p_1 \implies (p_1 \vee p_2)) \wedge ((p_1 \vee p_2)$ $\implies (p_2 \vee p_1))) \implies (p_1 \implies (p_2 \vee p_1))$ (axiom PL4)
11. $\vdash p_1 \implies (p_2 \vee p_1)$ (MP 9,10)

Je očividné, že důkaz v Hilbertovském systému je těžko čitelný. Tento důvod vyplývá z fakta, že takovéto důkazy jsou vytvářeny odspoda nahoru. Tedy nejdříve je potřeba pomocí axiomů vystavět formuli, na kterou může být aplikováno odvozovací pravidlo. Tato činnost je zdlouhavá a identifikovat které axiomy použít s jakými formulemi je netriviální operace.

Pro úplnost uvedu rozšíření daných kalkulů v predikátové logice. Pro všechny zmíněné kalkuly jsou přidány dva axiomy a dvě odvozovací pravidla:

- QL1. $\forall x.\phi(x) \implies \phi(t)$
- QL2. $\phi(t) \implies \exists x.\phi(x)$
- QR1. Pokud $\phi \implies \psi(a)$ je derivovatelné, a se neobjevuje v ϕ a x není vázaná proměnná v $\psi(a)$, pak $A \implies \forall x.\psi(x)$ je derivovatelné
- QR2. Pokud $\psi(a) \implies \phi$ je derivovatelné, a se neobjevuje v ϕ a x není vázaná proměnná v $\psi(a)$, pak $\exists x.\psi(x) \implies \phi$ je derivovatelné

, kde $phi(a)$ je formule, t je term a x je vázaná proměnná neobjevující se v $phi(a)$ [3, s. 45].

2.2.2.2 Gentzenovy dokazovací systémy

Druhým typem dokazovacích systémů jsou Gentzenovy. Ty se vyznačují velkým počtem odvozovacích pravidel a malým (obvykle žádným) množstvím axiomům. Samotné dokazování tedy probíhá shora dolů. Na počátku stojí komplikovaná formule, kterou postupným aplikováním pravidel zjednodušujeme. Navíc tato pravidla obvykle kopírují vzory, které matematici používají, takže dokazování je přímočařejší. Například při dokazování výroku ve formě „pokud P pak Q“ je v matematice obvyklé předpokládat P a na základě tohoto předpokladu dokázat Q. Není tedy potřeba mít dokázáno P, aby bylo možné dokázat $P \implies Q$. Axiomatické systémy umožňují tohoto výsledku dosáhnout pouze nepřímo přes deduktivní teorém. Pro demonstraci dokazovacích systémů založených na odvozovacích pravidlech uvádím systém vytvořený Gerhardem Gentzenem nazvaný přirozená dedukce [1, s. 5–27].

Přirozená dedukce je systém, který nemá žádné axiomy a který má pro každý logický operátor dvě verze odvozovacích pravidel. Introdukční pravidlo vytvoří ze vstupních argumentů formuli, která obsahuje daný operátor. Eliminační verze tohoto pravidla naopak operátor odstraňuje. Pravidla mají vstupní argumenty (nahoře) a výstup (dole) oddělené vodorovnou čarou [1, s. 5–27].

Pravidla pro konjunkci obsahují jedno introdukční pravidlo a dvě eliminační:

$$\frac{\phi \quad \psi}{\phi \wedge \psi}\wedge_i, \quad \frac{\phi \wedge \psi}{\phi}\wedge_e, \quad \frac{\phi \wedge \psi}{\psi}\wedge_e$$

Pravidla pro implikaci obsahují jedno eliminační pravidlo, které je de facto modus ponens. Introdukční pravidlo přináší do odvozovacího procesu vrstvu komplexnosti navíc. Před jeho aplikací je nutné nejdříve dokázat, že z předpokladu dané implikace, lze dojít ke kýženému závěru. Teprve poté lze toto pravidlo použít. Graficky je to vyjádřeno následovně:

$$\frac{\begin{array}{c}[\phi]\\ .\\ .\\ \psi\end{array}}{\phi \implies \psi} \implies_i, \quad \frac{\phi \implies \psi \quad \phi}{\psi} \implies_e$$

Pravidla pro dvojitou negaci říkají, že je ekvivalentní s absencí negace.

$$\frac{\phi}{\neg\neg\phi}\wedge_i, \quad \frac{\neg\neg\phi}{\phi}\wedge_e$$

Pravidla pro disjunkci obsahují dvě přímočará introdukční pravidla a jedno eliminační pravidlo, které podobně jako u implikace, vyžaduje samostatný důkaz před jeho aplikováním. Kromě

vstupního argumentu v podobě dané disjunkce, má toto pravidlo dva další vstupní argumenty, které říkají, že je nejdříve potřeba dokázat výstup pravidla pro oba členy disjunkce. Graficky vyjádřeno následovně:

$$\frac{\phi}{\phi\vee\psi}\vee_i,\ \frac{\psi}{\phi\vee\psi}\vee_i,\ \frac{\begin{array}{ccc}\phi\vee\psi & \begin{array}{c}[\phi]\\ \cdot\\ \cdot\\ \chi\end{array} & \begin{array}{c}[\psi]\\ \cdot\\ \cdot\\ \chi\end{array}\end{array}}{\chi}\vee_e$$

Pravidla pro negaci zavádějí práci s existencí kontradikce. Ta nám umožňuje dokončovat důkazy přes existenci negace. Zároveň je nutné ji pomocí odvozovacích pravidel zadefinovat:

$$\frac{\bot}{\phi}\bot_e,\ \frac{\phi\quad\neg\phi}{\bot}\neg_e$$

Pro samotné dokazování je zavedeno pravidlo, které nám umožní vytvořit předpoklad a následně dojít ke kontradikci:

$$\frac{\begin{array}{c}[\phi]\\ \cdot\\ \cdot\\ \bot\end{array}}{\neg\phi}\neg_i$$

Zbývá zavést pravidla pro práci v predikátové logice, tedy pravidla ohledně kvantifikátorů. Nejdříve obecný kvantifikátor:

$$\frac{\forall x.\phi}{\phi[t/x]}\forall_e\ ,\ \frac{\begin{array}{c}[x_0]\\ \cdot\\ \cdot\\ \phi[x_0/x]\end{array}}{\forall x.\phi}\forall_i$$

Eliminační pravidlo říká, že je možné odstranit kvantifikátor u proměnné x substitucí x za term t. Zároveň toto pravidlo kvůli definici substituce obsahuje vedlejší podmínku, že t je volné pro x v ϕ. Introdukční pravidlo opět zavádí potřebu důkazu před jeho aplikací. Říká, že pokud je pro novou proměnnou x_0 možné dokázat $\phi[x_0/x]$, pak lze usoudit $\forall x.\phi$. Významově toto pravidlo uvádí, že pokud je možné něco dokázat pro náhodnou proměnnou, lze to samé dokázat pro všechny.

Naposledy uvedu pravidla pro existenční kvantifikátor:

$$\frac{\phi[t/x]}{\exists x.\phi}\exists_i\ ,\ \frac{\exists x.\phi\qquad\begin{array}{c}[x_0.\phi[x_0/x]]\\ \cdot\\ \cdot\\ \chi\end{array}}{\chi}\exists_e$$

Introdukční pravidlo je přímočaré. Pokud je splněna postranní podmínka, že t je volné pro x v ϕ, pak lze ten výraz omezit pouze pro nějaké x. Pro eliminační pravidlo je nejdříve nutné dokázat, že můžeme x nahradit jakoukoliv proměnnou. Tedy, že zavedeme novou proměnnou x_0, která se neobjevuje ve zbytku formule – především v χ. Pokud po substituci x touto proměnnou úspěšně dokážeme χ, pak to znamená, že je možné χ dokázat pro jakoukoliv proměnnou, takže

lze odstranit existenční kvantifikátor [1, s. 5–27] [1, s. 107–117] [3, s. 65–88].

Jako poslední věc pro úplné představení přirozené dedukce uvedu dva příklady. Nejdříve dokáži $p_1 \vdash (p_2 \vee p_1)$, aby bylo vidět porovnání oproti příkladu 2.15. Následně ukážu dokazování v přirozené dedukci pro složitější důkaz. Důkazy v přirozené dedukci budu zapisovat ve formě $\phi_1, \phi_2, ...\phi_n \vdash \psi$, kde $\phi_1, \phi_2, ...\phi_n$ je množina formulí zvané premisy a ψ je jejich důsledek .

▶ **Příklad 2.16.** Důkaz $p_1 \vdash (p_2 \vee p_1)$:

1. p_1 (premisa)
2. $p_2 \vee p_1$ $(\vee_i 1)$

▶ **Příklad 2.17.** Důkaz $\forall x.(P(x) \implies Q(x)), \forall x.P(x) \vdash \forall x.Q(x)$, zdroj: [1].

1. $\forall x.(P(x) \implies Q(x))$ (premisa)
2. $\forall x.P(x)$ (premisa)
3. $x_0 \;\; P(x_0) \implies Q(x_0)$ $(\forall_e 1)$
4. $P(x_0)$ $(\forall_e 2)$
5. $Q(x_0)$ $(\implies_e 3, 4)$
6. $\forall x.Q(x)$ $(\forall_i 3 - 5)$

2.2.3 Korektnost dokazovacích systémů

Před samotným zavedením dokazovacího systému, kterým se zabývá tato práce, je potřeba ověřit, že takovéto dokazovací systémy jsou korektní. Tedy že aplikováním odvozovacích pravidel se nezmění pravdivost, kterou si s sebou formule nesou.

2.2.3.1 Sémantika

Sémantika v logice se zabývá samotným významem formulí, které jsou dokazovány. Toho cílí nahlédnutím do pravdivostních hodnot atomických formulí a jak je ovlivňuje aplikace logických operátorů.

▶ **Definice 2.18.** *Ohodnocení ve výrokové logice [1, s. 36–46].*

- *Množina pravdivostních hodnot obsahuje dva prvky T a F, kde T reprezentuje pravda a F je nepravda.*
- *Ohodnocení formule ϕ je přiřazení každého výrokového atomu v ϕ k pravdivostní hodnotě.*

Definovat ohodnocení v predikátové logice je složitější, jelikož je nutné brát v potaz, že odlišné proměnné se pro stejný predikát vyhodnocují různě. Zároveň v ohodnocení hrají roli kvantifikátory. Je nutné modelovat, jak spolu interagují termy v jednotlivých predikátech.

▶ **Definice 2.19.** *Nechť $\mathcal{F}$ je množina funčních symbolů a $\mathcal{P}$ množina predikátových symbolů, kde každý z těchto symbolů má fixní počet argumentů. Pak $\mathcal{M}$ je pár $(\mathcal{F}, \mathcal{P})$ skládající se z následující množiny dat (zdroj: [1, s. 122–127]):*

- *Neprázdná množina A, doména konkrétních hodnot.*
- *Pro každý 0-aritní funkční symbol $f \in \mathcal{F}$, konkrétní element $f^{\mathcal{M}}$ z A.*
- *Pro každý n-aritní funkční symbol $f \in \mathcal{F}$, kde $n > 0$, konkrétní funkci $f^{\mathcal{M}}: A^n \to A$, kde A^n je množina ntic nad A.*
- *Pro každý predikátový symbol $P \in \mathcal{P}$ s aritou > 0, podmnožina $P^{\mathcal{M}} \subseteq A^n$ ntic nad A.*

Pravdivostní vyhodnocení následně probíhá na základě tohoto $\mathcal{M}$ a následujících pravidel (zdroj: [4]):

- Atomická formule s predikátovým symbolem $P(t_1, t_2, ..., t_n)$ je T, pokud ntice $(o_1, o_2, ..., o_n)$ splňuje vlastnost predikátu P.
- Pokud známe pravdivostní ohodnocení formulí ϕ a ψ, pak známe i ohodnocení $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \implies \psi$, které je stejné jako ve výrokové logice.
- $\forall x.\phi$ je T, pouze pokud platí, že pro každé $a \in A$ po substituci za x dostaneme pravdivou formuli.
- $\exists x.\phi$ je T, pouze pokud platí, že pro nějaké $a \in A$ po substituci za x dostaneme pravdivou formuli.

▶ **Definice 2.20.** *Pokud pro všechna ohodnocení, ve kterých se $\phi_1, \phi_2, ..., \phi_n$ vyhodnotí jako T, se vyhodnotí jako T i ψ pak řekneme, že $\phi_1, \phi_2, ..., \phi_n \vDash \psi$ platí, a $\vDash$ nazveme relací sémantického důsledku [1, s. 36–46].*

Důkazy podle dokazovacích systémů jdou po syntaktickém významu výroků. Je tedy nutné uvést vztah mezi těmito dvěma pohledy. Platí dvě doplňující se tvrzení:

▶ **Tvrzení 2.21.** *O korektnosti odvozování v predikátové logice [2, s. 74–75].*
Pokud $\phi \vdash p$, pak $\phi \vDash p$.

▶ **Tvrzení 2.22.** *Rozšířená Gödelova věta o úplnosti predikátové logiky [5].*
Pokud $\phi \vDash p$, pak $\phi \vdash p$.

▶ **Důsledek 2.23.** *$\phi \vdash p$ právě tehdy, když $\phi \vDash p$.*

Důsledek těchto tvrzení nám říká, že sémantické a syntaktické dokazování jsou stejně validní. Při používání odvozovacích systémů se neztrácí význam formulí, i když se k němu nepřihlíží. Oba způsoby přistupují k řešení odlišným způsobem, čehož lze využít. Často může být snadnější nalézt ohodnocení formulí takové, ve kterém se předpoklady vyhodnotí na T a důsledek na F, než se neúspěšně snažit nalézt důkaz pomocí dokazovacích systémů.

2.3 Dokazovací systém aplikace

Dokazovací systém aplikace patří do odvozovacích systémů Gentzenova typu.

Proces dokazování je charakterizován změnou dokazované situace. Ta se skládá z vědomostní báze a výroku, který má být dokázán. Dokazovací situaci mění aplikace odvozovacích pravidel, která ji zjednodušuje. Cílem tohoto procesu je dosáhnout triviální situace, kdy se dokazovaný výrok nachází ve vědomostní bázi.

Při volbě, které odvozovací pravidlo v dané situaci aplikovat, je vždy třeba se řídit syntaktickou skladbou formule. Vždy lze aplikovat pouze pravidlo, které koresponduje s nejvnějším symbolem formule. Navíc pro formule ve vědomostní bázi a dokazovaný výrok jsou k dispozici odlišná pravidla. Použitím odvozovacích pravidel dochází k zjednodušení dokazované situace. Tedy že buď je zjednodušen výrok k dokázání, nebo je rozšířena vědomostní báze.

Při samotném dokazování se nejdříve snažíme dokázat daný výrok pomocí nalezení situace, kde se nachází ve vědomostní bázi. Pokud se nám to nepodaří, zkusíme předpokládat negaci výroku a nalézt kontradikci. V takovémto případě je výrok také dokázán. Pokud i tento přístup selže, můžeme využít předpokladu, že predikátová logika je konzistentní. Pokud se nám tedy podaří dokázat negaci výroku nebo předpokládat výrok a dojít ke kontradikci, znamená to, že pro daný výrok důkaz neexistuje a nemá smysl ho hledat[2, s. 76–84].

Nyní představím používaná odvozovací pravidla:

Pravidla pro konjunkci $A \wedge B$:

Pravidlo pro dokazování výroku	Pravidlo pro rozšiřování vědomostní báze
Odděleně dokažte A i B	Přidejte do vědomostní báze A i B

Při aplikování pravidla pro dokazování výroku dojde k rozštěpení dokazované situace na dvě odlišné. Obě tyto podsituace obsahují zjednodušený původní výrok. Zároveň ze sémantického pohledu tato operace dává smysl, jelikož konjunkce je pravdivá, pouze pokud jsou oba její členy pravdivé. Tudíž je nutné dokázat oba. U pravidla pro generování vědomostí zase je podobně potřeba předpokládat platnost obou členů, aby byla předpokládaná konjunkce validní.

Pravidla pro disjunkci $A \vee B$:

Pravidlo pro dokazování výroku
Přidejte $\neg A$ do vědomostní báze a dokažte B (nebo naopak)
Pravidlo pro rozšiřování vědomostní báze
Přidejte do vědomostní báze nejdříve A a dokončete důkaz pro tento případ,
pak vytvořte případ, kdy je do vědomostní přidáno B, a dokončete důkaz pro něj

Při aplikování pravidla pro dokazování výroku je přidána negace jednoho z členů disjunkce do vědomostní báze a druhý je dokazován. Negace ve znalostní bázi zajistí, že nositel pravdivostního ohodnocení disjunkce zůstane jako dokazovaný výrok. Při rozšiřování znalostní báze dochází opět k rozštěpení dokazovaných situací. Vzhledem k tomu, že po odstranění disjunkce není jasné, na kterém členu její platnost závisela, musíme dokázat výrok pro oba členy samostatně.

Pravidla pro implikaci $A \implies B$:

Pravidlo pro dokazování výroku
Předpokládejte A a dokažte B
Pravidlo pro rozšiřování vědomostní báze
Pokud se už A nachází ve vědomostní bázi, můžete usoudit i B

Obě tyto pravidla jsou základem dokazovacích systémů. Jak bylo uvedeno v teorii o Gentzenových dokazovacích systémech, předpokládání A při dokazování $A \implies B$ bylo jednou z hlavních motivací pro vytvoření tohoto typu systémů. Druhé pravidlo je instancí pravidla *modus ponens*.

Pravidla pro existenční kvantifikátor $\exists x.A$:

Pravidlo pro dokazování výroku
Vyberte si jakýkoliv term t a dokažte $A[x \leftarrow t]$
Pravidlo pro rozšiřování vědomostní báze
Vyberte si novou konstantu a a přidejte $A[x \leftarrow a]$ do znalostní báze

Smysl těchto pravidel kopíruje argumentaci, která byla představena již v rámci systému přirozené dedukce. Při dokazování výroku díky existenčnímu kvantifikátoru stačí, abychom nalezli libovolný term, pro který bude výrok dokázaný. Naopak při rozšiřování znalostní báze můžeme pouze předpokládat náhodnou konstantu, abychom neovlivnili výsledek důkazu. Toho docílíme pomocí zavedení konstanty, která se v důkazu do té doby neobjevila.

Pravidla pro obecný kvantifikátor $\forall x.A$:
Pravidlo pro dokazování výroku
Vyberte si novou konstantu a a dokažte $A[x \leftarrow a]$
Pravidlo pro rozšiřování vědomostní báze
Vyberte si jakýkoliv term t a přidejte $A[x \leftarrow t]$ do vědomostní báze

Pravidla pro obecný kvantifikátor jsou obrácenou verzí pravidel pro existenční kvantifikátor. Neboli chceme-li dokázat výrok obsahující obecný kvantifikátor musíme ho dokázat pro náhodnou proměnnou. Naopak při rozšiřování znalostí si lze zvolit jakýkoliv term, jelikož předpokládáme pravdivost tvrzení pro všechny možné hodnoty proměnné.

Pravidla pro negaci $\neg A$:
Pravidlo pro dokazování výroku
Předpokládejte A a pokuste se najít spor

Pravidlo pro negaci je použitelné pouze pro dokazování výroku. Po jeho aplikaci se snažíme nalézt konfiguraci znalostní báze vedoucí na důkaz sporem, kdy se ve znalostní bázi objeví formule ve tvaru A i ve tvaru $\neg A$.

Pravidlo pro ekvivalenci stejných výrazů:
$A \Leftrightarrow \neg\neg A$

Pravidlo pro ekvivalenci nám říká, že vždy je možné nahradit ekvivalentní výrazy. Tedy že vždy můžeme změnit A na $\neg\neg A$ anebo naopak. Tato operace se zejména hodí, když chceme dokázat A sporem. Nejdříve A změníme na $\neg\neg A$, poté aplikujeme pravidlo pro negaci a přidáme do vědomostní báze $\neg A$. Pokud následně dojdeme ke sporu, tak jsme dokázali A [6].

Navíc systém poskytuje možnost rozšiřování znalostní báze pomocí lemmat. Můžeme narazit na situaci, kdy pro dokončení důkazu nepomůže aplikovat žádné z odvozovacích pravidel. Pokud by se ale ve znalostní bázi nacházela určitá formule, důkaz by dokončit šel. Takovou formuli poté označíme jako lemma a pokusíme se ji dokázat pomocí současné znalostní báze. Pokud důkaz dokončíme, můžeme toto lemma přidat do znalostní báze důkazu. V rámci dokazování lemmatu lze vytvářet další vnitřní lemmata, která jsou potřeba k dokázání vnějšího lemmatu. Typickou situací pro potřebu lemmatu je chybějící předpoklad pro aplikaci *modus ponens* nebo možnost dokončení důkazu sporem pomocí dokázání negace formule, která se už ve vědomostní bázi nachází [6].

Kapitola 3
Analýza

V této kapitole se zabývám existujícími aplikacemi, které umožňují pracovat s dokazovacími systémy, a snažím se na jejich základě zanalyzovat ideální vlastnosti tvořené aplikace.

3.1 Existující aplikace

Existuje mnoho aplikací pro interaktivní použití teorie důkazů. Velká část z nich je vytvořená univerzitami, kde každé řešení naplňuje specifické požadavky dané univerzity. Na té nejvyšší úrovni lze existující aplikace rozdělit do třech různých kategorií.

První z nich jsou aplikace s velkou dokazovací silou, které umožňují dokazování pomocí různých dokazovacích systémů nad libovolnými objekty. Často používají pro dokazování svůj vlastní jazyk a není pro ně důležitá vizualizace. Jejich primárním cílem je usnadnit dokazování uživatelům, kteří jsou s teorií dobře srozuměni. Používání takovýchto aplikací vyžaduje podrobné studium dokumentace a úsilí věnované do učení jejich fungování. Množství úsilí lze přirovnat k učení nového programovacího jazyka.

Druhou kategorií jsou aplikace, které se zaměřují na porozumění určitému dokazovacímu systému pomocí interaktivity a vizualizace. Jejich cílovou skupinou jsou studenti. Mají za účel studentům usnadnit pochopení dokazování v daném dokazovacím systému bez toho, aby své uživatele zaplavily přílišným množstvím možných funkčností. Pro tyto aplikace je důležitá srozumitelná vizualizace dokazovacího procesu za cenu volnosti při jejich používání.

Třetí kategorií jsou aplikace, které ukazují vlastnosti dokazovacích systémů s velkou mírou abstrakce. Jejich cílem je ukázat, že lze dokazovací systémy využívat bez jejich podrobné znalosti při použití značné míry abstrakce jejich gamifikací. Demonstrují, že axiomy a odvozovací pravidla dokazovacích systémů se dají převést na zdánlivě nesouvisející objekty a následně je možné s nimi pracovat pomocí herních pravidel. Uživatel si při používání těchto aplikací nemusí uvědomovat, že ve skutečnosti pracuje na dokazování určitého tvrzení.

V následující části představím příklady aplikací pro všechny tyto kategorie.

3.1.1 Komplexní dokazovací aplikace

3.1.1.1 Isabelle

Isabelle je generický dokazovací asistent, který nabízí nástroje pro dokazování matematických formulí ve formálním jazyce. Původně byl vyvinut ve spolupráci *University of Cambridge* a *Technische Universität Münche* [7].

Isabelle využívá svůj jazyk *HOL*, který spojuje prvky funkcionálního programování s logikou. Moduly se v něm nazývají teoriemi, ve kterých probíhá uzavřené dokazování. Do teorie lze

importovat rodičovské teorie, které poskytnou jejich definice i v dané teorii (viz 3.1) [8].

Výpis kódu 3.1 Struktura teorie v Isabelle, zdroj: [8]

```
theory T
imports $T_1,...,T_n$
begin
    definitions, theorems, proofs
end
```

Isabelle má tři základní datové typy - *bool*, *nat (přirozené číslo)* a *int*. Tyto datové typy lze spojovat do listů a množin, jak je ve funkcionálním programování obvyklé. Isabelle také umožňuje definovat vlastní funkce. Na základě těchto funkcí jsou pak dokazovány lemmata a teorémy. Isabelle nabízí množství způsobů, kterými lze definovaná lemmata automaticky dokazovat. Jedná se například o důkaz indukcí, ale na výběr jsou i pokročilejší dokazovací algoritmy (*blast*, *sledgehammer*) [8].

Výpis kódu 3.2 Ukázka důkazu v Isabelle - dvě obrácení listu vyprodukují původní list, autor: Makarius

```
theory Seq
  imports Main
begin

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

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

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

lemma conc_empty: "conc xs Empty = xs"
  by (induct xs) simp_all

lemma conc_assoc: "conc (conc xs ys) zs = conc xs (conc ys zs)"
  by (induct xs) simp_all

lemma rev_conc: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
  by (induct xs) (simp_all add: conc_empty conc_assoc)

lemma reverse_reverse: "reverse (reverse xs) = xs"
  by (induct xs) (simp_all add: rev_conc)

end
```

V příkladu 3.2 vidět, že důkaz začíná definováním datového typu sekvence a definování funkcí conc a reverse. Pomocí těchto funkcí jsou definována lemmata, na které je aplikována indukce. Pro dokázání lemmat rev_conc a reverse_reverse jsou použita dříve dokázaná lemmata.

Isabelle obsahuje velké množství dalších nástrojů, které usnadňují dokazování. Má obsáhlou knihovnu předpřipravených důkazů, umožňuje rozšiřovat HOL o spustitelný kód a mnohé další. Samotná Isabelle je distribuována společně s IDE Isabelle/jEdit.

3.1.1.2 Coq

Coq je systém pro managment formálních důkazů. Implementuje jazyk *Gallina* založený na Kalkulu induktivních konstrukcí, který kombinuje logiku vyšších řádů s funkcionálním programováním. Poskytuje interaktivní dokazovací metody, rozhodovací algoritmy a jazyk taktik, kterým může uživatel definovat své vlastní dokazovací metody. Coq byl vyvinut a je udržován francouzským výzkumným ústavem INRIA [9].

Základem systému Coq je Coq kernel. Jedná se o jazyk, který obsahuje pouze základní funkčnosti, na které jsou všechny pro uživatele pohodlné abstrakce (notace, implicitní argumenty...) překládány. Tento přístup je zvolen, aby bylo nutné věřit pouze malé komponentě - kernelu, a vše ostatní je vůči ní ověřováno [10].

Jelikož jazyk je vystaven na kontrole typování, vše v Coqu je definováno typově. Existují zde dva typy objektů - tvrzení(3.3) a matematické struktury (3.4) [11].

Výpis kódu 3.3 Tvrzení v Coq, zdroj: [11]

```
forAll A B : Prop, A /\ B -> B \/ B
forAll x y : Z, x * y = 0 -> x = 0 \/ y = 0
```

Výpis kódu 3.4 Type v Coq, zdroj: [11]

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.

Inductive list (A:Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.
```

Podobně lze typově definovat funkce, viz 3.5.

Výpis kódu 3.5 Funkce sort v Coq, zdroj: [11]

```
sort : forAll (l : list nat),
{l' : list nat | sorted l' /\ same_elements l l'}
```

Při samotném dokazování uživatel nejdříve představí typy matematických struktur a natypuje dokazované vlastnosti. Následně nadefinuje teorém. Příkazem *Proof* započne dokazování tohoto teorému. Výběrem vhodných taktik pak mění dokazovanou situaci. Příklady těchto taktik může být *intro n*, která vytvoří term n, *induction s*, která započne dokazování *s* indukcí, nebo *simpl*, která simplifikuje dokazovanou situaci. Důkaz je uzavřen pomocí příkazu *Qed*. Překlad do Coq kernel jazyka takto vytvořeného důkazu si uživatel může zobrazit pomocí *Print* [11].

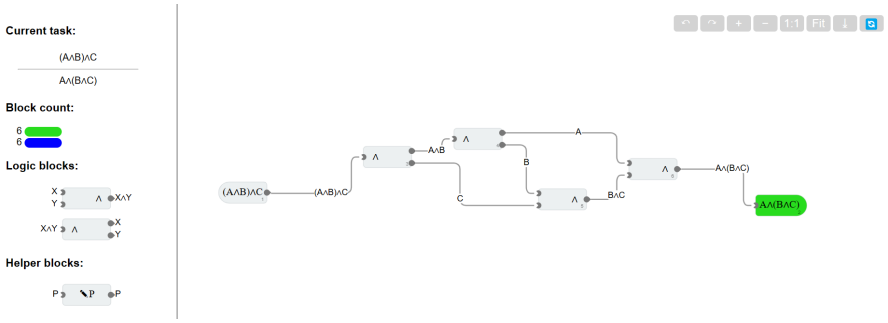
Celková funkcionalita systému Coq je mnohonásobně obsáhlejší a vyžaduje studium rozsáhlé dokumentace. Poskytuje například možnost vytváření knihoven důkazů, paralelního dokazování, extrakce certifikovaných programů do jiných jazyků (např. OCaml), automatické řešiče a mnohé další.

Coq je distribuován samostatně, ale také v rámci specializovaného CoqIDE, nebo různých pluginů pro jiná uživatelské rozhraní. Navíc existuje jsCoq umožňující používání systému Coq přímo v prohlížeči bez nutnosti instalace. Oproti výše uvedenému Isabelle má rozsáhlejší komunitu, která přispívá k integraci Coq do mnohých ekosystémů.

3.1.2 Výukové dokazovací aplikace

3.1.2.1 The Incredible Proof Machine

The Incredible Proof Machine je interaktivní dokazovací webová aplikace vytvořená za účelem zpřístupnění dokazování středoškolským žákům. Pracuje nad logikou prvního řádu. Byla vyvinuta pod vedením Joachima Breitnere z Karlsruher Institut für Technologie [12].



Obrázek 3.1 Důkaz asociativity konjunkce pomocí The Incredible Proof Machine, zdroj: [14]

Aplikace zobrazuje dokazovací proces pomocí tří hlavních typů bloků, jež uživatel propojuje. Každý blok může obsahovat vstupní a výstupní porty, se kterými interagují spojovací čáry. Bloky předpokladů obsahují pouze výstupní porty, bloky závěrů mají pouze vstupní porty. Logické bloky mají vstupní i výstupní porty a tvrzení, která jsou do nich vkládána na vstupních portech jako předpoklady, jsou transformovány dle definice bloku na výstupních portech jako závěry. Konektory mezi porty jsou označeny popiskem, který určuje, jakou formuli daný konektor nese, viz 3.1 [13].

Bloky lze libovolně posouvat po canvasu a tvořit tak názorný průběh důkazu. Mezi další funkce se řadí vytváření vlastních bloků z dříve dokázaných skutečností, které umožňují zkracovat důkazy, nebo pomocné bloky, které umožňují vyjadřovat přechodné výsledky. Navíc si aplikace pomatuje svůj stav, tudíž se lze k dříve rozdělaným důkazům vracet.

The Incredible Proof Machine ve své základní podobě obsahuje osm lekcí, ve kterých jsou připravené předpoklady a dokazovaná tvrzení. Logické bloky jsou implementovány jako pravidla přirozené dedukce, jejichž množina se postupně průchodem lekcemi rozrůstá. Na konci každé lekce je možné vytvořit vlastní důkaz s pravidly představenými v rámci dané lekce. Logika aplikování logických bloků je definována konfiguračním souborem, nikoliv přímo programována, tudíž dokazování není omezeno pouze na přirozenou dedukci. Tuto skutečnost aplikace demonstruje lekcí, která používá Hilbertovský systém dokazování.

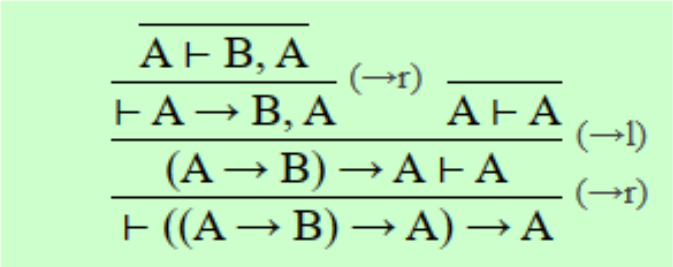
3.1.2.2 Logitext

Logitext je výukový dokazovací asistent využívající sekvenční kalkul. Byl vytvořen jako projekt Edward Z. Yanga na Massachusetts Institute of Technology.

Aplikace poskytuje webové rozhraní, jež je implementováno pomocí funkcionálních jazyků Ur/Web a Haskell. Toto rozhraní je integrováno s Coqem, který kontroluje faktickou správnost dokazovaných teorémů [15].

Samotné dokazování poté probíhá pomocí aplikování odvozovacích pravidel sekvenčního kalkulu na nejvnější symbol. Takto dochází zpětné dedukci, kdy na začátku se nachází celý *sekvent* ve tvaru $\Gamma, A_1, A_2, ..., A_n \vdash B$. Cílem dokazování je všechny větve důkazu převést pomocí odvozovacích pravidel do stavu, kde se na obou stranách $\vdash$ nachází stejná atomická klauzule a je tedy možné aplikovat axiom $\Gamma, A \vdash A$ [16].

Aplikace reaguje na kliknutí na logický symbol pro aplikaci pravidla, případně na vybrání atomické klauzule pro dokončení důkazu. Jsou aplikována rozdílná odvozovací pravidla podle toho, na které straně symbolu $\vdash$ se vybraný logický symbol nachází. Po aplikování pravidla je změna sekventu zobrazena jako vodorovná čára nad původním sekventem, nad kterou se nachází sekvent nový, případně sekventy dva, jestliže aplikované odvozovací pravidlo způsobuje rozvětvení důkazu. Při vybrání stejných atomických klauzulí na obou stranách sekventu se nad sekventem zobrazí vodorovná čára. Důkaz je dokončen v momentě, kdy jsou takto ukončeny všechny větve. Ukázka takto dokončeného důkazu pomocí aplikace Logitext je v 3.2 [16].



Obrázek 3.2 Důkaz pomocí Logitext. Kroky byly aplikovány zdola nahoru. Zdroj: [15]

3.1.3 Gamifikované dokazovací aplikace

3.1.3.1 Domino On Acid

Domino On Acid je aplikace pro vizualizaci Přirozené dedukce na hře domino. Byla vytvořena Matthiasem S. Benkmannem.

Tato aplikace je inspirována diagramatickým odvozováním, při kterém je úplně přehlížená sémantická strana důkazů. Namísto toho je kladen důraz na syntaktický přístup, který vede k abstrakci původního problému [17].

Domino On Acid ve svém jádru používá výrokovou logiku s jazykem, který obsahuje pouze výrokové proměnné, implikaci a $\bot$. Pro tento jazyk zavádí tři odvozovací pravidla přirozené dedukce:

- $\frac{X_1}{X_0 \implies X_1}[X_0]$ introdukce implikace
- $\frac{X_0 \; X_0 \implies X_1}{X_1}$ modus ponens
- $\frac{\bot}{X_0}[X_0 \implies \bot]$ reductio ad absurdum

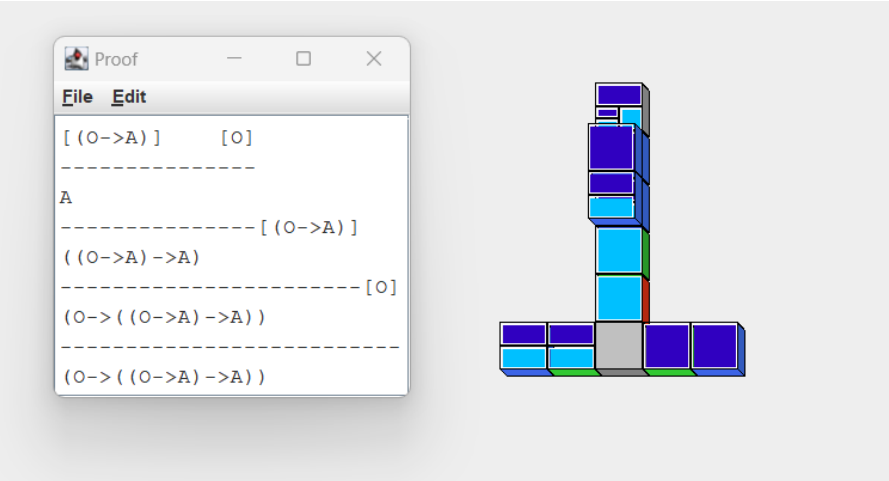
Nad horizontální čarou se nachází premisa, pod ní závěr a vedle čáry předpoklad [17].
Abstrakce spočívá v převodu těchto formulí na barvy dlaždic domina (zdroj [17]):

- $X_0 = \bot$ je označeno červenou barvou s bílým a černým okrajem.
- $X_0 = A, X_1 = B \ldots$ je označeno tak, že každé výrokové proměnné je přiřazena vlastní barva.
- $X_0 = (A \implies B)$ je označeno vícebarevně dle barev výrokových proměnných v implikaci. Toto barevné dělení nejdříve probíhá horizontálně. Pokud implikace obsahuje dceřiné implikace, tak jsou tyto horizontální bloky děleny střídavě vertikálně a horizontálně podle počtu dceřiných implikací.

Odvozovací pravidla jsou mapována na tvar takovýchto dlaždic. Dlaždice mají na jedné straně zelené bloky premis, na druhé straně červené bloky závěrů. Na vrchu dlaždic se nacházejí modré bloky předpokladů, které je potřeba *splnit*. Obrázek 3.4 je ukázkou konkrétních dlaždic [17].

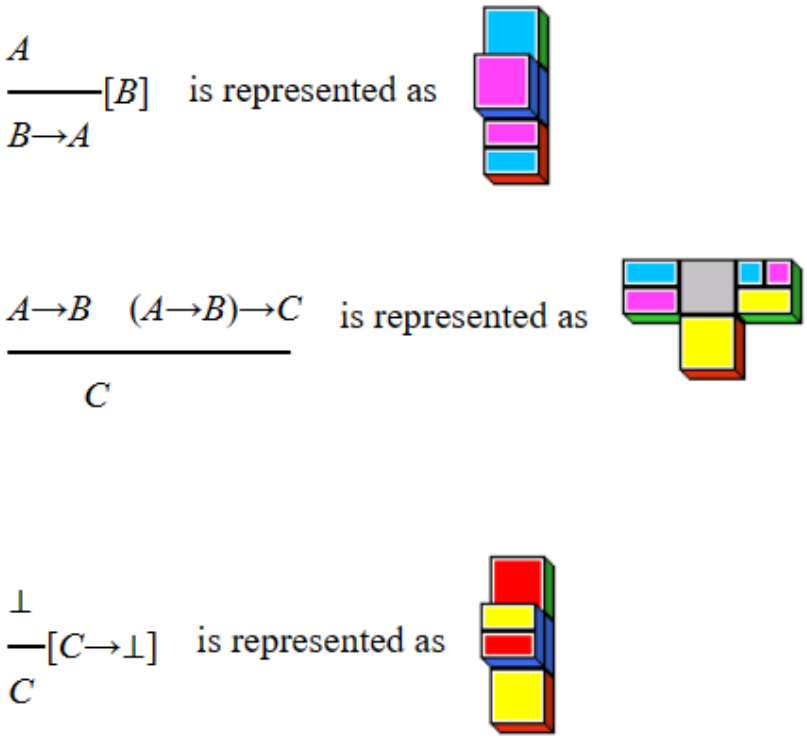
Hraní hry probíhá pokládáním dominových dlaždic, kdy lze k sobě položit pouze zelené a červené bloky, které mají stejný barevný vzor. Vzniká tak strom, který má ve svém kořeni počáteční dlaždici a alespoň jeden list s červeným blokem. List je uzavřen, jakmile se v něm vyskytuje zelený blok se stejným barevným vzorem, jako má některý modrý blok v jeho cestě ke kořeni. Hra je ukončena, jakmile jsou takto uzavřeny všechny listy[17].

Lze nahlédnout, že díky zavedenému mapování se jedná o validní důkaz v Přirozené dedukci. Navíc aplikace umožňuje zobrazit textový přepis standardní důkazové notace průběhu hry. Tuto skutečnost lze nahlédnout v komparaci těchto dvou reprezentací v 3.3.

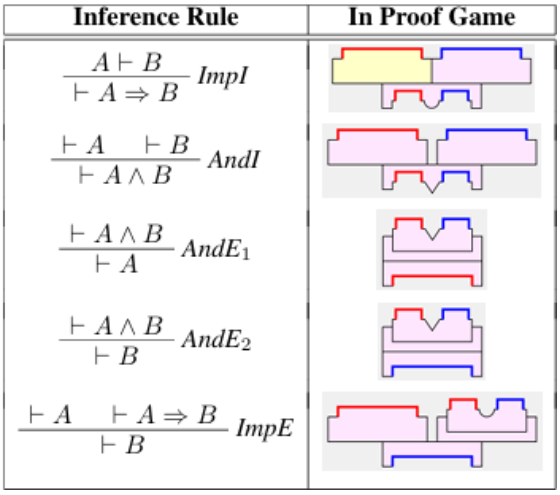


Obrázek 3.3 Důkaz jako strom domina a jeho textový přepis, zdroj: [18]

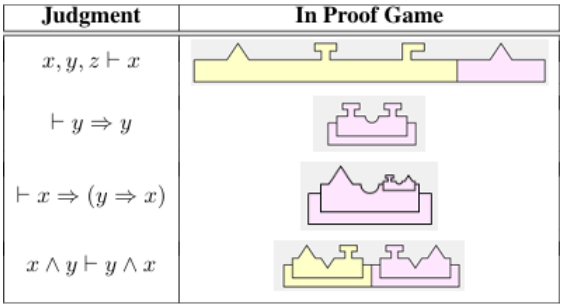
Hra je implementována v jazyce Java. Její JAR je volně přístupné ke stažení.



Obrázek 3.4 Dlaždice v Domino On Acid, zdroj: [17]



Obrázek 3.5 Odvozovací pravidla a jejich volné porty, zdroj: [19]



Obrázek 3.6 Tvary konektorů dle kódovaných proměnných, zdroj: [19]

3.1.3.2 Polymorphic Blocks

Polymorphic Blocks je rozhraní, které vizualizuje dokazování přirozenou dedukcí pomocí stavění bloků. Byla vyvinuta na University of California, San Diego.

Polymorphic Blocks volí velmi podobný přístup jako Domino On Acid, které má usnadnit učení dokazovacích systémů pomocí gamifikace. Avšak liší se herními mechanikami, které představuje. Namísto hry domina a přiřazování barevných vzorů, používá bloky různých tvarů s různě tvarovanými konektory, které do sebe zapadají jako puzzle.

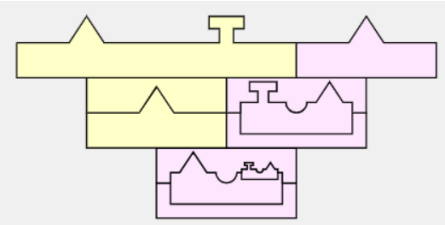
Jazyk možných logických operátorů omezuje pouze na konjunkci a implikaci, tedy omezuje i množství odvozovacích pravidel. Jejich výpis je zobrazen na obrázku 3.5. Barevné plochy v tomto obrázku představují volné porty, kam lze umístit blok s výrokovými proměnnými. Různé výrokové proměnné jsou zobrazeny různými tvary konektorů, viz 3.6. V momentě, kdy je spojen blok odvozovacího pravidla s takto tvarovanými konektory, dojde k přesunu těchto konektorů i na ostatní barevné plochy bloku odvozovacího pravidla. Tvar konektoru, který se dotýká červené plochy, se zkopíruje na druhou červenou plochu, tvar konektoru, který zapadl do modré plochy, se kopíruje na modrou plochu atd. Navíc má každý blok dvě barvy, jedna reprezentující dokazované tvrzení, druhá premisy neboli kontext. Takto jsou stavěny bloky na sebe, pokud jsou použita pravidla *AndI* nebo *ImpE*, vznikají dva bloky místo jednoho. Blok je možné *uzavřít*, pokud má na svém vrchu stejný konektor v obou barvách. Uzavření vyjadřuje aplikaci pravidla předpokladu (3.7). Toto uzavření je reprezentováno přikrytím konektorů. Důkaz je dokončen v momentě, kdy jsou uzavřeny všechny svrchní bloky [19].

Uživatele provádí postupně úrovněmi, ve kterých představuje nové mechaniky – pravidla. To

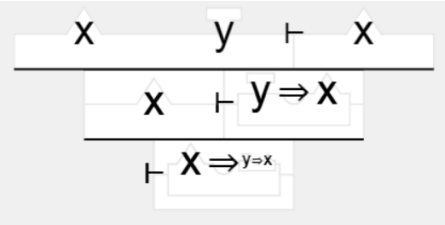
umožňuje uživateli snadno hru pochopit bez toho, aniž by musel mít jakékoliv teoretické základy. Aplikace nicméně umožňuje zobrazit logiku, která se pod bloky skrývá, a tím nahlédnout do struktury důkazu. Tato funkce je vidět v rozdílu mezi obrázky 3.8 a 3.9.

$$\dfrac{\dfrac{\dfrac{}{y \vdash y}\,\textit{Assume} \quad \dfrac{}{y \vdash y}\,\textit{Assume}}{y \vdash y \wedge y}\,\textit{AndI}}{\vdash y \Rightarrow (y \wedge y)}\,\textit{ImpI}$$

Obrázek 3.7 Zakončení důkazu pravidlem předpokladu, zdroj: [19]



Obrázek 3.8 Vystavěné bloky, zdroj: [20]



Obrázek 3.9 Vystavěné bloky se zobrazenou logikou, zdroj: [20]

3.2 Závěr analýzy

V rámci analýzy jsem představil příklady aplikací, které přistupují k problému dokazovacích systémů různými způsoby. Vzhledem k požadovanému zadání je nejbližší kategorií pro aplikaci vyvíjenou v rámci této práce kategorie aplikací, které se přímo zaměřují na výuku.

Komplexní dokazovací systémy jako Isabelle nebo Coq poskytují potřebnou logiku pro dokazovací systém se znalostní bází. Nicméně jejich syntax s vlastním jazykem a nepřeberné množství funkcí by si vynutily alespoň několik lekcí strávených jejich učením, než by bylo možné přistoupit k samotnému dokazovacímu systému.

Naopak aplikace, které se zaměřují na gamifikaci, pro studijní potřeby využívají příliš velké abstrakce. Jsou vystavěny tak, že je uživatel může používat bez toho, aniž by cokoliv věděl o logických důkazech. Bez toho aniž by si uživatel přečetl dokumentaci těchto aplikací, nemá naději pochopit, jak důkaz v rámci dané hry probíhá. Navíc dokazovací systémy nelze snadno gamifikovat, což obě zkoumané aplikace vyřešili omezením jazyka na menší počet logických symbolů a používáním pouze výrokové logiky, což zadání vylučuje.

Největší míra inspirace pro aplikace bude tedy vycházet z výukových dokazovacích systémů, kde uživatel jasně vidí průběh důkazu. Obě analyzované aplikace jsou názorné a snadné k použití. Z mého pohledu je o něco přístupnější The Incredible Proof Machine, jelikož volné porty na

odvozovacích pravidlech jasně uživateli napovídají, co očekávají na svém vstupu. V Logitextu musí uživatel chvíli experimentovat, než pochopí, jakým způsobem by mělo probíhat dokazování. Také větší důraz Proof Machine na vizualizaci průběhu obvykle uživateli jasně napoví, jaký je další krok. Naopak Logitext vyniká v integraci se systémem Coq, který zaručuje jeho správnost. Navíc důkazní proces je zobrazen ve standardní notaci, tudíž uživatel po jeho dokončení vidí kroky, které k dokončení vedly.

Na základě této analýzy budu do mé aplikace přidávat vizuální systém podobný The Incredible Proof Machine. Propojování bloků pomocí konektorů je velmi vizuálně názorné. Navíc se často používá i v jiných aplikacích, například při kreslení UML diagramů, tudíž je na to cílová skupina aplikace, studenti, zvyklá. Z aplikace Logitext se inspiruji popisem důkazu po jeho dokončení. V Proof Machine je pro uživatele obtížné sepsat kroky vedoucí k dokončení důkazu, kdežto v Logitextu může uživatel pouze opsat obsah obrazovky.

Kapitola 4
Softwarový návrh

V této kapitole jsou nejdříve popsány požadavky kladené na vyvíjený systém a následně je na jejich základě popsán postup pro výběr použitého řešení.

4.1 Systémové požadavky

Systémové požadavky rozděluji na funkční a nefunkční požadavky dle jejich povahy.

4.1.1 Funkční požadavky

Funkční požadavky specifikují funkčnost systému. Obvykle vztahují ke konkrétní funkci systému, zatímco nefunkční požadavky jsou naplňovány skupinou funkcí. Korektní funkční požadavek je měřitelný a implementovatelný. Tedy po dokončení implementace je jasné, zda byl splněn, či nikoliv [21].

4.1.1.1 FP1: Vložení logické formule

Uživatel musí být schopen do systému zadat vlastní logickou formuli k dokázání. Tato formule musí svou formou vyhovovat dokazovací metodě. To znamená především, že pro formuli a její podformule bude možné určit vnější symbol. Zároveň formule bude složená z konstant a termů spojených povolenými logickými symboly. Pokud formule nebude vyhovovat formátu, aplikace ji musí odmítnout.

4.1.1.2 FP2: Dokazování logické formule pomocí pravidel

Uživatel musí mít dostupná dokazovací pravidla s jejichž pomocí je možné formuli dokázat. Pravidlo lze aplikovat pouze vždy na aktuálně nejvnější logický symbol formule. Jakmile je pravidlo aplikováno na formuli k dokázání, provede se transformace formule dle definice pravidla, případně je přidán nový fakt do knihovny znalostí. Pokud dochází k aplikaci pravidla na fakt z knihovny znalostí, je vytvořen nový fakt, případně dojde k rozvětvení důkazu.

4.1.1.3 FP2a: Aplikace pravidla konjunkce

Aplikování pravidla konjunkce musí probíhat v souladu s jeho definicí. Aplikace pravidla na známý fakt způsobí vytvoření dvou nových faktů. Pokud je pravidlo použito pro dokázání formule, důkaz se rozdělí do dvou větví.

4.1.1.4 FP2b: Aplikace pravidla disjunkce

Aplikování pravidla disjunkce musí probíhat v souladu s jeho definicí. Při použití na známý fakt dojde k rozdělení důkazu do dvou větví, kdy je jednou předpokládáno A a podruhé B. Při aplikaci na dokazovanou formuli si uživatel může vybrat, zda chce předpokládat negaci A a dokazovat B, nebo naopak.

4.1.1.5 FP2c: Aplikace pravidla implikace

Aplikování pravidla implikace musí probíhat v souladu s jeho definicí. Při použití na známý fakt dojde k přidání nového faktu, pouze pokud už se v databázi faktů nachází předpoklad. Při aplikaci na dokazovanou formuli dojde k přidání předpokladu mezi fakta.

4.1.1.6 FP2d: Aplikace pravidla negace

Aplikování pravidla negace musí probíhat v souladu s jeho definicí. Pravidlo lze aplikovat pouze pokud se uživatel snaží dokázat formuli, jejíž vnější symbol je negace. Po aplikaci se přidá formule bez této negace mezi známá fakta a uživatel pro dokončení důkazu musí najít kontradikci mezi fakty.

4.1.1.7 FP2e: Aplikace existenčního pravidla

Aplikování existenčního pravidla musí probíhat v souladu s jeho definicí. Při aplikování na známý fakt musí uživatel zadat novou konstantu, kterou se vázaná proměnná nahradí. Aplikace musí zkontrolovat, že zadaná konstanta je validní. Při použití na dokazovanou formuli si uživatel může vybrat libovolný term.

4.1.1.8 FP2f: Aplikace obecného pravidla

Aplikování obecného pravidla musí probíhat v souladu s jeho definicí. Při aplikování na známý fakt si uživatel může vybrat libovolný term. Při použití na dokazovanou formuli musí uživatel zadat novou konstantu, kterou se vázaná proměnná nahradí. Aplikace musí zkontrolovat, že zadaná konstanta je validní.

4.1.1.9 FP2g: Aplikace pravidla ekvivalence

Aplikování pravidla ekvivalence musí probíhat v souladu s jeho definicí. Aplikováním tohoto pravidla může uživatel přidávat či odebírat dvojitou negaci logické formule.

4.1.1.10 FP3: Zadání lemma

Uživatel musí být schopen v rámci důkazu formule přidat nové lemma. V momentě přidání lemmatu se důkaz přepne a uživatel nejdříve dokazuje toto lemma. V případě úspěšného dokončení důkazu je lemma přidáno do knihovny znalostí.

4.1.1.11 FP4: Krok zpět

Aplikace si musí pamatovat historii svého stavu, aby uživatel v případě potřeby mohl zvrátit svoje předchozí akce.

4.1.1.12 FP5: Shrnutí důkazu

Aplikace si musí pamatovat historii svého stavu, aby si uživatel po úspěšném dokončení dokazování mohl prohlédnout jednotlivé kroky. Nahlédnout do historie důkazu bude mít uživatel umožněno dvěma způsoby. Pomocí stručného textového logu a pomocí vizualizace.

4.1.1.13 FP6: Vizualizace historie důkazu

V rámci vizualizace historie důkazu bude mít uživatel možnost nahlédnout do detailu aplikování pravidla. Při aplikování pravidla vždy dochází ke změně stavu a uživatel by měl být schopen tuto změnu stavu blíže prozkoumat. Zároveň by měl být uživatel schopen si libovolně skrývat detaily dokazování lemmatu.

4.1.2 Nefunkční požadavky

Nefunkční požadavky kladou omezení a předpoklady na fungování systému. Funkční požadavky určují, jakou činnost má systém vykonávat. Oproti tomu nefunkční požadavky říkají, jakým způsobem toho má docílit [22].

4.1.2.1 NP1: Responzivní uživatelské rozhraní

Uživatelské rozhraní by mělo být jednoduché na použití. Doba odezvy by měla být do jedné sekundy.

4.1.2.2 NP2: Bezpečnost

Aplikace by měla být chráněná vůči běžným typům útoků jako SQL injection, či cross-site scripting.

4.1.2.3 NP3: Udržovatelnost

Aplikace by měla obsahovat detailní dokumentaci, která po dokončení aplikace bude umožňovat snadnou údržbu systému.

4.1.2.4 NP4: Rozšiřitelnost

Aplikace by měla být navržena takovým způsobem, aby ji bylo možné snadno rozšířit o novou funkcionalitu.

4.2 Případy užití

V této sekci popíšu jednotlivé případy užití aplikace uživatelem. Případ užití je textový popis interakce uživatele a systému. Zahrnuje jeho krátký popis, předpoklady, účastníky, běžný scénář, alternativní scénáře, případně výjimky a stav systému po jeho dokončení [23].

4.2.1 PU1: Vložení logické formule

Tento případ užití začíná v momentě, kdy uživatel chce vložit do aplikace logickou formuli k dokázání. Případ užití končí, jakmile je logické formule do systému vložena.

- **Předpoklady**: Žádné
- **Účastníci**: Systém, uživatel

- **Běžný scénář**: Uživatel zadá do textového pole logickou formuli, kterou by chtěl dokázat. Systém následně zkontroluje, zda daná formule má správný formát. Pokud ano, systém si formuli zapamatuje.
- **Alternativní scénář**: Uživatel si během zadávání logické formule není jistý, jak má vypadat správný formát formule. Klikne na tlačítko Pomoc. Systém mu nabídne nápovědu s potřebnými informacemi. Uživatel si může tuto nápovědu prohlédnout a následně pokračovat ve vkládání logické formule.
- **Výjimka**: Po odeslání logické formule uživatelem systému, systém zjistí, že formule není ve správném formátu. Systém zpraví uživatele, že formule neodpovídá požadovanému formátu. Uživatel následně může zadanou logickou formuli opravit.
- **Stav systému po dokončení**: V systému je vložena logická formule připravená k dokazování.

4.2.2 PU2: Dokázání logické formule

Tento případ užití začíná v momentě, kdy je v systému vložená logická formule a uživatel ji chce dokázat. Uživatel dokazuje logickou formuli pomocí aplikací pravidel na vnější symbol formule. Aplikací pravidel se původní formule rozpadá na podformule, na které lze opět používat pravidla. Aplikací určitých pravidel se rozšiřuje knihovna faktů, na jejímž základě lze formuli dokázat. Případ užití končí ve chvíli, kdy se mezi dvěma formulemi v knihovně faktů objeví kontradikce nebo když se mezi fakty objeví formule, kterou uživatel dokazuje.

- **Předpoklady**: Logická formule uložená v systému
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel aplikuje pravidlo na formuli. Systém následně zkontroluje, zda se dané pravidlo dá na danou formuli aplikovat. Pokud ano, systém transformuje stav aplikace dle definice daného pravidla. Tedy především upraví dokazovanou formuli a rozšíří seznam faktů. Následně systém zkontroluje, jestli transformací stavu došlo k dokončení důkazu. Pokud ano, zpraví uživatele, že důkaz byl dokončen. Pokud ne, uživatel může pokračovat v aplikaci pravidel.
- **Alternativní scénář**: Uživatel si není jistý, jakým způsobem dokazování funguje. Klikne na tlačítko Pomoc. Systém mu nabídne nápovědu s potřebnými informacemi. Uživatel si může tuto nápovědu prohlédnout a následně pokračovat v dokazování.
- **Výjimka**: Pokud systém zjistí, že vybrané pravidlo nelze dle definice aplikovat na danou formuli, vyhodí chybu a neprovede žádnou transformaci. Uživatel následně může zkusit jinou akci.
- **Stav systému po dokončení**: Důkaz logické formule byl dokončen.

4.2.3 PU3: Aplikace pravidla konjunkce

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli pravidlo konjunkce. Chování systému se liší v závislosti na tom, zda je pravidlo aplikováno na formuli, kterou chce uživatel dokázat, či zda je použito na formuli z knihovny faktů a slouží k jejímu rozšíření. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: Vybraná logická formule má jako svůj vnější logický symbol konjunkci.

- **Účastníci**: Systém, uživatel
- **Běžný scénář - formule k dokázání**: Uživatel použije pravidlo konjunkce na formuli k dokázání ve tvaru $A \wedge B$. Systém následně rozdělí důkaz na dvě větve. V jedné větvi je dokazováno A a v druhé větvi je dokazováno B.
- **Běžný scénář - knihovna faktů**: Uživatel použije pravidlo konjunkce na některou formuli z knihovny faktů ve tvaru $A \wedge B$. Systém do knihovny faktů přidá A i B.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.4 PU4: Aplikace pravidla disjunkce

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli pravidlo disjunkce. Chování systému se liší v závislosti na tom, zda je pravidlo aplikováno na formuli, kterou chce uživatel dokázat, či zda je použito na formuli z knihovny faktů a slouží k jejímu rozšíření. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: Vybraná logická formule má jako svůj vnější logický symbol disjunkce.
- **Účastníci**: Systém, uživatel
- **Běžný scénář - formule k dokázání**: Uživatel použije pravidlo disjunkce na formuli k dokázání ve tvaru $A \vee B$. Systém se následně uživatele zeptá, zda chce předpokládat $\neg A$ a dokazovat B, nebo naopak. Uživatel si vybere jednu z těchto možností a systém na základě toho rozšíří knihovnu faktů a upraví dokazovanou formuli.
- **Běžný scénář - knihovna faktů**: Uživatel použije pravidlo disjunkce na některou formuli z knihovny faktů ve tvaru $A \vee B$. Systém následně rozdělí důkaz na dva případy. V jednom je předpokládáno A a v druhém B. Zároveň si systém uloží, že pro dokončení důkazu je potřeba dokázat oba případy a uživateli zobrazí dokazování jednoho z případů.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.5 PU5: Aplikace pravidla implikace

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli pravidlo implikace. Chování systému se liší v závislosti na tom, zda je pravidlo aplikováno na formuli, kterou chce uživatel dokázat, či zda je použito na formuli z knihovny faktů a slouží k jejímu rozšíření. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: Vybraná logická formule má jako svůj vnější logický symbol implikaci.
- **Účastníci**: Systém, uživatel
- **Běžný scénář - formule k dokázání**: Uživatel použije pravidlo implikace na formuli k dokázání ve tvaru $A \implies B$. Systém přidá A do knihovny faktů a formuli k dokázání zjednoduší na B.
- **Běžný scénář - knihovna faktů**: Uživatel použije pravidlo implikace na některou formuli z knihovny faktů ve tvaru $A \implies B$. Systém zkontroluje, zda se předpoklad A nachází v knihovně faktů a případně přidá B mezi známá fakta.
- **Výjimka - knihovna faktů**: Pokud systém vyhodnotí podmínku, že předpoklad A nachází v knihovně faktů, zpraví uživatele o této skutečnosti a neprovede žádnou transformaci stavu.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.6 PU6: Aplikace pravidla negace

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli pravidlo negace. Pravidlo negace lze aplikovat pouze na formuli, kterou chce uživatel dokázat. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: Dokazovaná formule má jako svůj vnější logický symbol negaci.
- **Účastníci**: Systém, uživatel
- **Běžný scénář - formule k dokázání**: Uživatel použije pravidlo negace na formuli k dokázání ve tvaru $\neg A$. Systém přidá A do knihovny faktů. Systém si uloží příznak, že od tohoto momentu probíhá hledání kontradikce mezi fakty.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.7 PU7: Aplikace existenčního pravidla

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli existenční pravidlo. Chování systému se liší v závislosti na tom, zda je pravidlo aplikováno na formuli, kterou chce uživatel dokázat, či zda je použito na formuli z knihovny faktů a slouží k rozšíření této knihovny. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: Vybraná logická formule má jako svůj vnější logický symbol existenční symbol.
- **Účastníci**: Systém, uživatel
- **Běžný scénář - formule k dokázání**: Uživatel použije existenční pravidlo na formuli k dokázání ve tvaru $\exists x.A$. Systém nabídne uživateli textové pole k zadání termu t, kterým bude x nahrazeno. Po zadání termu uživatelem systém zkontroluje, zda se jedná o validní term. Pokud ano, systém provede náhradu $A[x \leftarrow t]$.
- **Běžný scénář - knihovna faktů**: Uživatel použije existenční pravidlo na některou formuli z knihovny faktů ve tvaru $\exists x.A$. Systém nabídne uživateli textové pole k zadání konstanty a, kterou bude x nahrazeno. Po zadání konstanty uživatelem systém zkontroluje, zda se jedná o validní konstantu. Pokud ano, systém přidá do knihovny faktů $A[x \leftarrow a]$.
- **Výjimka - formule k dokázání**: Pokud systém vyhodnotí, že uživatelem zadaný term je nevalidní, zobrazí chybovou hlášku a neprovede žádnou transformaci stavu.
- **Výjimka - knihovna faktů**: Pokud systém vyhodnotí, že uživatelem zadaná konstanta je nevalidní, zobrazí chybovou hlášku a neprovede žádnou transformaci stavu.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.8 PU8: Aplikace obecného pravidla

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli obecné pravidlo. Chování systému se liší v závislosti na tom, zda je pravidlo aplikováno na formuli, kterou chce uživatel dokázat, či zda je použito na formuli z knihovny faktů a slouží k rozšíření této knihovny. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: Vybraná logická formule má jako svůj vnější logický symbol obecný symbol.
- **Účastníci**: Systém, uživatel

- **Běžný scénář - formule k dokázání**: Uživatel použije obecné pravidlo na formuli k dokázání ve tvaru $\forall x.A$. Systém nabídne uživateli textové pole k zadání konstanty a, kterou bude x nahrazeno. Po zadání konstanty uživatelem systém zkontroluje, zda se jedná o validní konstantu. Pokud ano, systém provede náhradu $A[x \leftarrow a]$.
- **Běžný scénář - knihovna faktů**: Uživatel použije existenční pravidlo na některou formuli z knihovny faktů ve tvaru $\forall x.A$. Systém nabídne uživateli textové pole k zadání termu t, kterým bude x nahrazeno. Po zadání termu uživatelem systém zkontroluje, zda se jedná o validní term. Pokud ano, systém přidá do knihovny faktů $A[x \leftarrow t]$.
- **Výjimka - formule k dokázání**: Pokud systém vyhodnotí, že uživatelem zadaná konstanta je nevalidní, zobrazí chybovou hlášku a neprovede žádnou transformaci stavu.
- **Výjimka - knihovna faktů**: Pokud systém vyhodnotí, že uživatelem zadaný term je nevalidní, zobrazí chybovou hlášku a neprovede žádnou transformaci stavu.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.9 PU9: Aplikace pravidla ekvivalence

Tento případ užití začíná v momentě, kdy uživatel chce aplikovat na formuli pravidlo ekvivalence. Pravidlo ekvivalence slouží k převedení formule na její jinou podobu s logicky stejným významem. Případ užití končí v momentě, kdy je provedena transformace stavu aplikace.

- **Předpoklady**: V systému je uložena formule.
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel použije pravidlo ekvivalence na vybranou formuli. Systém se poté zeptá, zda chce uživatel před vybranou formuli přidat dvojtou negaci, nebo zda chce dvojitou negaci odebrat. Uživatel vybere jím preferovanou možnost a systém provede transformaci.
- **Výjimka**: Pokud si uživatel, že chce odebrat dvojitou negaci, přestože vybraná formule nemá dvojitou negaci jako svůj vnější symbol, stav aplikace se nezmění.
- **Stav systému po dokončení**: Stav aplikace je změněn dle definice pravidla.

4.2.10 PU10: Přidání lemmatu

Tento případ užití začíná v momentě, kdy uživatel během dokazování narazí na situaci, kdy potřebuje pro dokončení důkazu rozšířit svoji knihovnu faktů o novou formuli. Toho může docílit přidáním lemmatu a jeho následným dokázáním. Případ užití končí ve chvíli, kdy je lemma dokázáno a systém ho přidá mezi známá fakta.

- **Předpoklady**: Logická formule uložená v systému
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel použije tlačítko Přidat lemma. Systém uživateli nabídne textové pole, do kterého uživatel může vložit lemma, které by rád dokázal. Systém zkontroluje, že se jedná o validní formuli. Následně uživatel může pokračovat s dokazováním lemmatu s pomocí již dříve známých faktů. Dokazování probíhá pomocí aplikování pravidel uživatelem. Systém po každé aplikaci pravidla transformuje svůj stav. Pokud se mezi známými fakty objeví kontradikce nebo pokud je některý známý fakt stejný jako lemma k dokázání, systém označí lemma jako dokázané. Následně vrátí stav aplikace před přidání dokázaného lemmatu a přidá ho mezi známá fakta.

- **Alternativní scénář**: Uživatel během dokazování lemmatu zjistí, že pro dokázání daného lemmatu je potřeba dokázat další lemma. Systém umožní uložit současný stav důkazu lemmatu a započíst další důkaz lemmatu hlouběji. Po dokončení důkazu se stav systému vrátí před tento bod, dokázané sublemma se přidá do znalostní báze a uživatel může pokračovat v dokazování původního lemmatu.
- **Stav systému po dokončení**: Knihovna faktů je rozšířena o dokázané lemma.

4.2.11 PU11: Krok zpět

Tento případ užití začíná v momentě, kdy uživatel usoudí, že jeho současný postup nepřináší kýžený výsledek. Aby nemusel začínat nový důkaz, tak se může vrátit pouze o krok zpět. Případ užití končí poté, co se stav aplikace navrátí před předchozí transformaci.

- **Předpoklady**: Systém již provedl nejméně jednu změnu stavu.
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel použije tlačítko Krok zpět. Systém se podívá do paměti, kde má uložené změny stavů, a revertuje poslední změnu stavu.
- **Stav systému po dokončení**: Stav aplikace je navrácen do bodu před poslední transformací.

4.2.12 PU12: Zobrazení průběhu důkazu - stručně

Tento případ užití začíná ve chvíli, kdy uživatel dokončil úspěšně důkaz a chce si zpětně zobrazit své kroky. Pokud mu stačí krátký přepis zobrazující kroky, které uživatel během dokazování provedl, může si prohlédnout textový přepis. Tento případ užití končí poté, co uživatel dokončí prohlížení důkazu.

- **Předpoklady**: Formule byla dokázána.
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel si rozklikne textový přepis historie důkazu. Systém mu zobrazí jednotlivé kroky, které uživatel podnikl, v textové podobě.
- **Stav systému po dokončení**: Stav aplikace se nezmění.

4.2.13 PU13: Zobrazení průběhu důkazu - podrobně

Tento případ užití začíná ve chvíli, kdy uživatel dokončil úspěšně důkaz a chce si zpětně zobrazit své kroky. Pokud potřebuje si prohlídnout průběh důkazu ve větším detailu, použije vizuální zobrazení historie dokazování. V ní si může prohlídnout detaily jednotlivých kroků v podobě přechodu mezi stavy systému. Tento případ užití končí poté, co uživatel dokončí prohlížení důkazu.

- **Předpoklady**: Formule byla dokázána.
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel si rozklikne vizuální historii důkazu. Systém mu zobrazí jednotlivé kroky dokazování. V krocích, kde bylo aplikováno pravidlo, si uživatel může zobrazit stav systému před a po aplikaci transformace.
- **Stav systému po dokončení**: Stav aplikace se nezmění.

4.2.14 PU14: Zobrazení průběhu důkazu - skrytí důkazu lemmatu

Tento případ užití začíná ve chvíli, kdy uživatel chce při zobrazovaní historie důkazu skrýt kroky, které zobrazují dokazování lemmatu. To uživateli umožní se soustředit na pouze hlavní osu důkazu a odstínit vedlejší větve. Tento případ užití končí v momentě, kdy uživatel dokončí prohlížení důkazu, nebo kdy nechá kroky lemmatu opět zobrazit.

- **Předpoklady**: Formule byla dokázána s pomocí lemmatu.
- **Účastníci**: Systém, uživatel
- **Běžný scénář**: Uživatel si rozklikne vizuální historii důkazu. Systém mu zobrazí jednotlivé kroky dokazování. V kroku, kde byla zadána lemma, uživatel klikne na tlačítko Skrýt. Systém skryje všechny následující kroky, které se pojí s dokazováním daného lemmatu. Uživatel následně může kliknutím na stejné tlačítko důkaz lemmatu opět zobrazit.
- **Stav systému po dokončení**: Aplikace skryje kroky vedoucí k dokázání lemmatu.

4.3 Pokrytí funkčních požadavků případy užití

V tabulkách 4.1 a 4.2 je zobrazeno pokrytí funkčních požadavku případy užití.

Tabulka 4.1 Pokrytí funkčních požadavků případy užití - 1.

	FP1	FP2	FP2a	FP2b	FP2c	FP2d	FP2e	FP2f	FP2g
PU1	X								
PU2		X							
PU3		X	X						
PU4		X		X					
PU5		X			X				
PU6		X				X			
PU7		X					X		
PU8		X						X	
PU9		X							X

Tabulka 4.2 Pokrytí funkčních požadavků případy užití - 2.

	FP3	FP4	FP5	FP6
PU10	X			
PU11		X		
PU12			X	
PU13				X
PU14				X

4.4 Výběr technologií

V této části popíšu technologické možnosti, které jsem zvažoval pro implementaci aplikace. V rámci analýzy jsem určil, že chci vytvořit responsivní webovou aplikaci. Tato aplikace má pouze

klientskou část, tudíž jsem hledal programovací jazyk a framework, jež budou nejlépe odpovídat požadavkům aplikace.

4.4.1 HTML

Protože se jedná o webovou aplikaci, v jejím centru musí být HTML, které nemá jinou alternativu.

Hypertext Markup language je základní jazyk pro tvorbu webových stránek, který je úzce spojen se vznikem internetu.

Jeho kořeny sahají do CERNu do roku 1990, kde byl vytvořen pro snadnější sémantický popis vědeckých prací. Vzhledem k rapidnímu rozvoji s internetem spojených technologií byla již roku 1994 vytvořena organizace W3C, jejímž úkolem bylo tyto technologie centrálně spravovat [24].

V následujících letech probíhaly snahy o přepracovaní jazyka HTML do nové podoby XML zvaného XHTML. To způsobilo velkou nevoli kvůli zpětné nekompatibilitě a přerušení inovací původního HTML. Tato situace vyústila až v založení konkurenční organizace WHATWG, která si dala za cíl inovovat HTML bez narušení zpětné kompatibility a s robustní specifikací.

V roce 2011 se obě organizace dohodly na vývoji HTML5. Tento vývoj opět provázely spory mezi organizacemi, kdy W3C chtěla vydat hotovou verzi HTML5, zatímco WHATWG chtěla průběžně HTML5 podporovat a vydávat opravy a vylepšení. Přesto v roce 2019 dosáhli dohody, že dále budou společně vyvíjet pouze jednu verzi HTML [25].

Jazyk HTML je postaven na stromové struktuře, která obsahuje HTML prvky a texty. Každý HTML prvek je ohraničen počátečním a koncovým tagem. Tagy různých prvků se nesmí překrývat, prvky mohou být do sebe vnořené, nebo stát vedle sebe. Prvky mohou obsahovat uvnitř tagu atributy, jež určují způsob fungování těchto prvků. Příkladem takového kódu je ukázka 4.1.

Webové prohlížeče překládají HTML do DOM stromu. Ten slouží pro reprezentaci objektů v systému klienta, s pomocí něhož mohou skriptovací jazyky s těmito prvky interagovat [24].

Výpis kódu 4.1 Příklad HTML

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Sample page</title>
    </head>
    <body>
        <h1>Sample page</h1>
        <p>This is a <a href="demo.html">simple</a> sample.</p>
        <!-- this is a comment -->
    </body>
</html>
```

4.4.2 CSS

Podobně dominantní postavení jako HTML má i CSS, které je s ním úzce spojeno. V rámci HTML5 byla odstraněna většina funkcionalit, které umožňovaly měnit zobrazování. Toto rozhodnutí bylo učiněno, aby se dostálo cíle, že HTML je nezávislé na platformě (tedy i dostupné i na nevizuálních). Jedinou zobrazovací funkcionalitou, která přímo v HTML zůstala, je atribut style, jehož používání není příliš preferováno. Hlavním způsobem pro stylování webových aplikací tedy zůstalo CSS [25].

Jelikož cílem této práce je vytvořit vizuální software, který pomůže studentům s vizuálním zobrazováním důkazního procesu, bylo třeba do mého technologického stacku přidat i CSS.

Podobně jako HTML i CSS bylo vyvinuto v CERNu roku 1994. Stylizování webových dokumentů nebylo do té doby nijak standardizováno. HTML specifikace předpokládala, že uživatel, či jeho prohlížeč si sám určí, jakým způsobem chce dokument zobrazit. Prohlížeče se ale ubíraly směrem, který stále více omezoval uživatele v aplikování stylu na své dokumenty. Odpovědí na tento problém se stalo CSS neboli Cascading Style Sheets, které se dají aplikovat na stromově orientované jazyky (HTML, XML). Ty, jak název napovídá, kaskádově vytváří styl dokumentu podle přání autora, čtenáře, ale i podle možností zobrazovacího zařízení. Od té doby je v rámci organizace W3C dále rozvíjeno [26].

CSS je vydáváno v takzvaných Levels. Každý Level je nadmnožinou předchozího Levelu, který rozšiřuje a případně opravuje. První 2 Levely byly monolitické a zahrnovaly celou implementaci CSS. Level 3 znamenal přechod na modulární implementaci, kdy se jednotlivé funkční celky oddělily. V současné době má tedy každý modul svůj vlastní Level, nezávislý na ostatních [27].

Kritickým modulem je Selectors, který zajišťuje vyhledávání prvků ve stromu dokumentu, na které jsou aplikovány styly. Druhy selektorů jsou vypsány v následujícím listu [26].

- **Type selector**: Selektor typu prvku. Příkladem takového selektoru je: *h1* pro výběr h1.
- **Attribute selector**: Selektor dle hodnoty atributu. Například: *[title=val]* pro výběr h1 s atributem title=val.
- **Class selector**: Selektor dle přidělené třídy. To jest: *.myClass* pro výběr h1 s atributem class=myClass.
- **Id selector**: Selektor vybírají podle id prvku. Tedy například: *#myId* pro výběr h1 s atributem id=myId.
- **Pseudoclass selector**: Selektor, který vybírá prvek na základě okolností, které nejsou přímo vyjádřeny ve stromové reprezentaci dokumentu, nebo dle informací, které nejdou vyjádřit kombinací předchozích jednoduchých selektorů. Pseudoclass selektory jsou dvojího typu. Příkladem dynamického typu (informace leží vně stromu) je: *a:hover* pro výběr ¡a¿ elementu, po vyvolání události typu hover. Příkladem strukturálního typu (element nelze určit kombinací jednoduchých selektorů) je: *tr:nth-child(even)* pro výběr všech sudých potomků prvku tr.

4.4.3 JavaScript

JavaScript je skriptovací jazyk, který doplňuje webové aplikace založené na HTML a CSS o nastavitelné chování.

JavaScript byl vytvořen Brendanem Eichem z Netscape Communications v roce 1995. Byl vytvořen specificky pro jejich webový prohlížeč Netscape Navigator. Později byl specifikován pomocí nově vytvořeného ECMAScript. Největšího rozkvětu se dočkal až po zavedení výkonných enginů do webových prohlížečů, kterým nevadilo, že JavaScript není kompilovaný jazyk [28].

Specifikace určená jazykem ECMAScript označuje za obecně použitelný skriptovací jazyk, takový programovací jazyk, který je používaný k manipulaci, customizaci a automatizaci částí existujícího systému. V takových systémech, užitečná funkcionalita je již dostupná přes uživatelské rozhraní, a skriptovací jazyk je mechanismus jejího odhalení kontrole programu [29].

Základní jednotkou JavaScript programu je value, který určuje běh programu. Value se dělí na dva typy – primitivní a objekt. Mezi primitivní typy patří například character nebo boolean. Objekty jsou kolekce klíčů a hodnot. Speciálními objekty jsou pole a funkce. JavaScript je slabě typovaný jazyk (viz 4.2), tudíž POJO lze vytvořit bez předchozího definování, či není potřeba specifikovat konverze mezi typy [30].

Výpis kódu 4.2 Ukázka slabého typování JavaScriptu

```
function greet(name) {
  return "Hello, " + name;
}
```

4.4.4 TypeScript

TypeScript je nadmnožinou JavaScriptu, která přidává do tohoto jazyka statické typování. To z něj činí, narozdíl od JavaScriptu, silně typovaný jazyk.

TypeScript byl představen roku v roce 2012 Microsoftem. Motivací pro vytvoření TypeSriptu byla narůstající komplexita JavaScriptových aplikací, kde runtime chyby bylo v rozsáhlém kódu těžké identifikovat. TypeScript měl přinést statické typování a určité prvky objektově-orientovaného programování. Ukázkou anotací typů budiž 4.3.

Kompilátor TypeScriptu převede před samotným spuštěním kód na jeho JavaScriptovou verzi. Díky tomu je TypeScriptový kód kompatilní s JavaScriptem [31].

Výpis kódu 4.3 Ukázka statického typování TypeScriptu.

```
function greet(name: string): string {
  return "Hello, " + name;
}
```

4.4.5 Výběr skriptovacího jazyka

Výhodou TypeScriptu je, že jeho silné typování umožňuje snazší údržbu kódu. Obecně je lépe čitelný a s dobrým IDE umožňuje odhalovat chyby spojené s typováním dříve, než vůbec nastanou. I přes výhody jazyka TypeScript jsem si vybral pro implementaci JavaScript, a to právě pro jeho slabé typování. Aplikace, kterou má tato práce za cíl vytvořit, nedosahuje příliš velkých rozměrů. Díky tomu je snadno chybám v typech předcházet a mít přehled o celé implementaci. Naopak slabé typování umožňuje vynechat definici typů neprimitivních objektů a využívat slabé typování pro flexibilní obsah prvků jednotlivých objektů. Obě tyto věci přispívají k menší délce kódu a pohodlnějšímu vývoji. Navíc, využívání JavaScriptu je bezpečnější cesta z hlediska využívání knihoven. Používání JavaScriptových knihoven v TypeScriptovém prostředí může vést k problémům s kompilací kvůli odlišnému typování. Nicméně kompaktnost aplikace umožňuje toto rozhodnutí v budoucnu snadno změnit a přejít k silnému typování v podobě TypeScriptu.

4.4.6 Angular

Angular je open-source framework pro tvorbu webových aplikací napsaný v TypeScriptu od společnosti Google.

Nynější Angular vzešel z AngularuJS. AngularJS, dříve Angular 1.0, vznikl v roce 2012, aby při implementaci skryl přímou práci JavaScriptu s DOMem. Pro svou práci používal JavaScript. AngularJS přinesl mnoho konceptů, které jsou dnes neodmyslitelnou součástí i konkurenčních frameworků [32].

AngularJS byl vytvořen pro snazší tvorbu dynamických webových CRUD aplikací. Samotné HTML nebylo pro takový účel zamýšleno. Vytvářet responzivní webové aplikace není přímočaré a vyžaduje velké množství boilerplate kódu. AngularJS přinesl velké množství abstrakce, které schovává přímou práci s DOMem. Abstrakci také používá ke skrývání callbacků, což opět kód zpřehledňuje [33].

První důležitou funkčností byl data binding. Běžně systémy pro vykreslení pohledu spojovaly template, který určoval, jakým způsobem má pohled fungovat, s modelem, který naplňoval

template daty. Vykreslený pohled už nijak dál nereflektoval změny v modelu. Pokud uživatel udělal změny v pohledu, již nebyly propsány zpět do modelu. Udržovat model a pohled synchronizovaně byla odpovědnost uživatele. Data binding představený AngularemJS tento problém odstranil změnou konceptu modelu. Po zkompilování templatu je vytvořen pohled, který je naplněn modelem. Pokud dojde k aktualizaci dat v pohledu, změna je ihned propagována zpět do modelu. Pohled je tedy pouze instancí modelu a přestává nezávisle existovat [34].

Druhou novinkou bylo zavedení návrhové vzoru dependency injection. Vkládání závislostí, namísto jejich tvoření, vytváří lépe udržovatelný a testovatelný kód [35].

Poslední novinkou bylo zavedení direktiv. Direktivy umožňují seskupovat standartní HTML elementy a doplňovat je o nastavitelné chování. Když kompilátor narazí na direktivu pomocí jejího selektoru identifikuje, jak má být transformována a o jaké chování být rozšířena. Hlavní selektory jsou dva: jméno elementu a hodnota atributu [36].

Přes všechny tyto funkcionality se postupně začaly objevovat problémy AngularuJS. AngularJS výkonnostně ztrácel při obsluhování velkého množství prvků na DOMu. Google tedy přistoupil k redesignu frameworku. Spojil síly s Microsoftem, aby společně přenesli framework do TypeScriptu a začali využívat dekorátorů. Angular 2.0, později pouze Angular, využíval všechny výše zmíněné funkcionality, přidával svá vlastní vylepšení, odstraňoval výkonnostní problémy a umožnil používat Typescript pro tvorbu aplikací [32].

Největší změnou bylo členění kódu do komponent. Každá komponenta je samostatně stojící jednotka obsluhující určitou funkcionalitu aplikace. Každá komponenta má vlastní template, který určuje, jakým způsobem bude vykreslena, kód, který určuje její chování, a případně CSS, který na ní aplikuje styly [32].

4.4.7 React

React je open-source knihovna pro tvorbu uživatelského rozhraní, kterou vytvořila a udržuje společnost Meta.

React byl poprvé využit v rámci Facebook Ads v roce 2012. Jeho syntax byl inspirován XHP, který byl rozšířením PHP. V roce 2013 došlo k oddělení Reactu od implementace Facebooku a jeho open-sourcování [37, s. 1–20].

Základní myšlenkou Reactu je, že jakékoliv uživatelské rozhraní lze dělit inkrementálně do komponent. Pokud je toto dělení dostatečně precizní, určité komponenty se začnou opakovat, a tudíž je možné jejich implementaci sdílet. Tímto inkrementálním dělením je možné vytvořit strom komponent, kde jeho kořen obsahuje komponentu celé aplikace, listy obsahují základní prvky jako jsou tlačítko, zatímco uzly obsahují své potomky pospojované do logických celků [38].

Každá komponenta je JavaScript funkce, která implementuje chován této komponenty. Navíc ale obsahuje část podobnou HTML, která označuje, jak má být komponenta vykreslována. Toto syntaktické rozšíření JavaScriptu se nazývá JSX. Původně bylo vyvinuto přímo jako součástí Reactu, ale nyní je možné ho využívat v jakémkoliv projektu nezávisle pod open-source licencí. JSX je preprocesory přeložen na vnořené JavaScript funkce, tudíž prohlížeče již konzumují pouze čistý JavaScript [37, s. 43–63].

Informace jsou mezi komponenty sdíleny vždy pouze směrem od rodičovských komponent k jejich potomkům. Ty je jim předkládají pomocí props. Pokud dojde ke změně prop, na již vykreslené komponentě, tak dojde k jejímu překreslení [38].

Velkou součástí moderního Reactu jsou React Hooks. Hooky jsou JavaScript funkce, které usnadňují práci vývojářům. Nejzákladnějším hookem je useState. Tent umožňuje komponentě uchovávat stav a odstraňuje nutnost využívat JavaScript class a stavové proměnné. Druhým významným Hookem je useEffect. Ten se zavolá po dokončení určité funkce. Umožňuje tedy provádět různé postranní efekty, jež předpokládají, že daná funkce už byla dokončena. Kromě built-in hooks si může uživatel definovat i libovolné vlastní [39].

4.4.8 Vue.js

VueJS je JavaScript open-source framework pro tvorbu uživatelských rozhraní.

VueJS byl vytvořen Evanem You, jenž pracoval v Googlu na AngularJS projektech. AngluarJS byl již v té době robustní framework, který se zaměřoval především na CRUD aplikace. Když You chtěl vytvořit pouze prototyp uživatelského rozhraní, znamenalo to, že musí napsat velké množství kódu navíc. Aby napravil tuto situaci, extrahoval části AngularuJS, které umožňovaly rychlý vývoj prototypů uživatelského rozhraní. Na základě toho vznikl VueJS. Postupem času se dále vyvíjel do dnešní podoby flexibilního frameworku, jež umožňuje tvořit komplexní webové aplikace [40].

Kód VueJS je členěn do komponent. To zvyšuje čitelnost, udržovatelnost a možnost znovupoužití kódu. Komponenta je zapsána v jediném souboru, který je rozdělen na tři části. Část ohraničená tagem script obsahuje logiku aplikace popsanou obvykle JavaScriptem. Tag template ohraničuje template aplikace zapsaný pomocí HTML. Poslední částí je style, kde je pomocí CSS aplikován styl komponenty. VueJS používá dvě různé API pro překlad, starší Options API a novější Composition API. Composition API umožňuje vynechávat určité deklarace, jako je například ohraničení pomocí script tagu, ale logické dělení zůstává stejné [41].

Základním rysem VueJS je reaktivita. VueJS pro sledování stavu objektů používá reaktivní JavaScript objekty. Pokud je takový objekt čten nebo měněn, VueJS tento proces naruší. Při tomto narušení se v případě čtení projde množina závislostí tohoto objektu a v případě změny se updatují objekty závislé na měněném objektu. Tato implementace umožňuje tvorbu dvoucestné vazby mezi pohledem a modelem. které jsou synchronizovány [42].

VueJS se inspiroval u Reactu a pro vykreslení stránky používá virtuální DOM. HTML template je převeden do JavaScript objektů, které obsahují veškeré informace ve formě atributů a následnické prvky jsou uloženy jako atribut typu pole. Vzniká tak v paměti reprezentace DOMu. Při renderování je tento virtuální strom traversován a dochází k jeho převedení na skutečný DOM. Pro urychlení vytváření virtuálního DOMu, především při aktualizaci jeho závislostí, používá kompilátor mnohé optimalizace. Například statické prvky nejsou vždy znovu převáděny na jejich virtuální reprezentaci. Místo toho je jejich reprezentace po prvním vytvoření virtuálního stromu uložena a při tvoření nových je zavolána z paměti[43].

4.4.9 Výběr knihovny/frameworku

Při výběru řešení pro implementaci aplikace jsem uvažoval nad výše uvedenými frameworky a knihovnou, jakožto oblíbenými řešeními skrz softwarovou komunitou. Měl jsem zkušenosti s používáním Angularu, který byl díky tomu mou první volbou. Během prozkoumávání těchto možných řešení jsem ale došel k závěru, že pro účel této aplikace není ideální. Kvůli své robustnosti obsahuje velké množství funkcí, které bych ve své čistě FE aplikaci zaměřené na uživatelské rozhraní nevyužil. Zároveň jsem chtěl, z již výše zmíněných důvodů, používat JavaScript, což je sice Angularem podporováno, ale nekoresponduje to s jeho myšlenkou.

React a VueJS mají velmi podobné motivace a funkcionality. VueJS nabízí obsáhlejší funkcionalitu, která vychází z jeho frameworkové podstaty, zatímco React, jakožto knihovna, pro tvoření webových aplikací spoléhá na externí knihovny. Při rozhodování mezi Reactem a VueJS jsem se zaměřil na konkrétní věci, které jsem chtěl ve své aplikaci implementovat. Především se jednalo o kreslení šipek pro propojování pravidel s bloky logických formulí. Pro VueJS jsem našel akorát řešení využívající Canvas, která mi přišla pro můj případ užití komplikovaná. Flexibilnější React s větší podporou komunity naproti tomu nabízel hned dvě knihovny, se kterými lze jednoduše dosáhnout kýženého výsledku. Díky této skutečnosti jsem se rozhodl postavit svoje řešení na knihovně React.

4.5 Lo-Fi prototyp

Jelikož zadání práce klade důraz na uživatelského rozhraní aplikace, v rámci softwarového návrhu jsem vytvořil papírový model uživatelského rozhraní.

Papírový model neboli wireframe, je vizuální prezentace webové aplikace, která zahrnuje pouze základní strukturu a prvky. Wireframe se obvykle zaměřuje pouze na základní prvky, jako jsou bloky obsahu, textová pole, tlačítka, obrázky a další komponenty, bez detailních grafických prvků, barev nebo stylů. To umožňuje vývojáři lépe rozplánovat strukturu aplikace a zamyslet se nad tokem informací, bez nutnosti řešit vizuální styly [44].

Díky wireframe lze v rané fázi vývoje určit se zadavatelem, které prvky mají mít jinou strukturu, či co je potřeba dělat jiným způsobem. Pokud jsou takové změny potřeba, jejich cena je nesrovnatelně nižší, než kdyby už proběhla plnohodnotná implementace [44].

Při tvoření wireframu jsem identifikoval tři základní obrazovky aplikace.

Na první z nich uživatel musí vložit do systému formuli k dokázání. Tato obrazovka musí obsahovat textové pole a tlačítko, kterým se vložení formule potvrdí. Navíc by tato obrazovka měla umožňovat uživateli vkládat speciální logické symboly, které se nevyskytují na běžné klávesnici. Do této obrazovky tedy byla přidána i klávesnice pro takováto tlačítka.

Na druhé obrazovce probíhá samotné dokazování. Na obrazovce se mají vyskytovat bloky známých faktů a samotná formule k dokázání. Navíc obrazovka musí obsahovat bloky pravidel, která se dříve zmíněnými bloky spojují pomocí konektorů. Dále je potřeba vizuálně odlišit, že pravidla pro ten samý logický symbol se chovají jinak, pokud jsou použita na známý fakt a pokud jsou použita na dokazovanou formuli. Obrazovka také musí uživatele seznámit s definicí pravidel. Pro tuto funkci jsem si vybral zobrazení pomocí tooltipu. Nakonec se na obrazovce dle funkčních požadavků musí nacházet tlačítko pro přidání lemmatu.

Třetí obrazovka se uživateli objeví po dokončení dokazování. Měl by na ní mít možnost nahlédnout na kroky, které vedly k dokončení důkazu. Tento náhled musí být přehledný.

Vytvořený wireframe je dostupný v příloze: A.

Použití papírového modelu se ukázalo jako velmi výhodné, protože na základě zpětné vazby od zadavatele byly následně přidány funkční požadavky FP2g, FP4 a FP6. Také bylo určeno, že uživatel nemá mít možnost vybrat pravidlo, které se neshoduje s vnějším symbolem vstupního bloku.

Kapitola 5
Implementace

V této kapitole popisuji, jakým způsobem je aplikace implementovaná na základě znalostí z předchozích kapitol. Dále uvádím, na jaké problémy jsem při implementaci narazil a jakým způsobem byly řešeny.

5.1 Vkládání logické formule

První funkční požadavek uvádí, že systém musí umožňovat uživateli vložit libovolnou logickou formuli predikátové logiky. V kapitole Teorie bylo definováno, co se za logickou formuli považuje. Zároveň bylo předloženo, jak lze pomocí parsovacích stromů ověřit, že formule má validní syntax. Sestavování parsovacího stromu vede na implementaci rekurzivního algoritmu. Ten hledá nejvnější symbol. Jakmile ho nalezne, vytvoří nový uzel. Tento uzel obsahuje informace o druhu nalezeného symbolu, jak vypadá část formule, jejíž nejvnější symbol byl právě nalezen, a odkazy na synovské uzly doplněné rekurzí. Pokud se jedná o kvantifikátor, je uloženo, ke které proměnné se váže. Rekurzivní volání pro vytvoření synovských uzlů je zavoláno na formuli bez daného symbolu. Pokud se jednalo o binární symbol, zavolá dvě rekurzivní volání pro levou a pravou část. Ukončovací podmínka tohoto algoritmu je, že vstupem rekurzivního volání je term. Term dle obvyklé notace značí malá písmena, zatímco predikátový symbol velká písmena. Z definice parsovacího stromu predikátu vyplývá, že jeho podstrom bude končit termem. Pro implementaci je výhodné spojit celý podstrom, který obsahuje predikáty, funkce a proměnné do jednoho listu a manipulovat s nimi společně. Pokud všechny rekurze algoritmu jsou ukončeny takovýmto listem, jedná se o validní formuli a je vrácen její parsovací strom. Pseudokód algoritmu je rozepsán v 5.1.

Problém zadání logické formule do systému s sebou přináší celou řadu výzev a problémů.

První z nich představuje výzvu z hlediska uživatelského pohodlí. Logické operátory predikátové logiky se nenachází v běžné znakové sadě klávesnice, tudíž je pro uživatele obtížné takovou formuli správně napsat. Tento problém jsem řešil hned dvěma způsoby. Prvním z nich je přidání tlačítek na obrazovku, na které uživatel může kliknout a vložit tak speciální symbol do vstupního pole. Tyto tlačítka jsou vidět na návrhu A.1. Druhým způsobem je použití náhradních znaků pro reprezentaci logických symbolů. Uživatel může například napsat symbol &, který se před sestavováním parsovacího stromu přeloží na $\wedge$. Podobné mapování existuje pro všechny používané logické symboly. Díky tomu může uživatel rychle napsat formuli, kterou chce dokázat, pouze pomocí své klávesnice.

Další problém je obvyklá forma, kterou se formule zapisují. Aby bylo možné zpracovávat vstup softwarově, je nutné definovat vlastnosti validní formule. Uzavřené hranaté závorky značí nedělitelnou část formule – podformuli. Vnitřek dvojice závorek je v parsovacím stromu reprezentován jedním uzlem a všechny hranaté závorky obsažené ve vrchní dvojici závorek jsou uzly v jeho podstromu. Kulaté závorky značí argumenty funkce. Velká písmena predikáty a malá písmena

Výpis kódu 5.1 Pseudokód pro tvorbu parsovacího listu logické formule.

```
function buildParseTree(formula):
    // Odebrani vnejsich zavorek z aktualni casti formule
    pokud prvni char formule je '[' a posledni symbol formule je ']':
        formula = odstranit vnejsi zavorky z formule

    inicializovat pocet zavorek

    pro kazdy char v formuli:
        pokud char je '[':
            zvysit pocet zavorek
        jinak pokud char je ']':
            snizit pocet zavorek
        jinak pokud char je unarni operator a pocet zavorek je 0:
            nastavit flag, ze byl nalezen unarni operator
        jinak pokud char je binarni operator a pocet zavorek je 0:
            vytvorit novy uzel s binarnim operatorem a dvema syny
            syn1: buildParseTree(formula.substring(0, operator))
            syn2: buildParseTree(formula.substring(operator, konec))
        jinak pokud char je kvantifikator a pocet zavorek je 0
          a neni nastaven flag unarniho operatoru:
            najit promennou
            vytvorit novy uzel s kvantifikatorem, promennou
              a jednim synem
            syn1: buildParseTree(formula.substring(operator, konec))
    pokud pocet zavorek je 0:
        pokud char na zacatku formule je unarni operator:
            vytvorit novy uzel s unarnim operatorem a jednim synem
            syn1: buildParseTree(formula.substring(operator, konec))
        jinak:
            // List je platny term pouze tehdy, pokud obsahuje
                pouze symboly predikatu, funkci a promennych
            pokud formule obsahuje pouze platne symboly:
                vytvorit novy platny list
            jinak:
                list je neplatny
    jinak:
        zkoumana formule je neplatna
```

proměnné a názvy funkcí. Kromě logických symbolů, jejich náhradních znaků, tečky a čárky nejsou žádné další znaky povoleny. Uživatel je s očekávaným formátem srozuměn v nápovědě.

Problémem jsou také zvyklosti pro zápis formulí. Ideálně by každý uzel a list parsovacího stromu byl oddělen závorkami od zbytku formule, aby bylo pro systém snadné identifikovat vnější symbol. Obvykle se ale takto oddělují pouze části formule dělené binárními operátory. Unární operátory a termy závorkami ohraničené nejsou. U kvantifikátorů zase platí, že se vztahují pouze k části formule, která se nachází napravo od nich a zároveň ve stejných závorkách. Přesto lze pomocí určení přednosti jednotlivých typů operátorů vždy určit, jaká část formule je právě vnější. V případě že se v rámci jedné dvojce závorek objeví binární i unární operátor, má přednost binární symbol. Pokud pouze unární symbol, vytvoří se uzel pro unární operátor. Jestli se uvnitř dvojce závorek objeví kvantifikátor, vytvoří se uzel, pokud je v rámci dané části formule nejlevější logický symbol. Zároveň pokud závorky obsahují více operátorů stejného druhu, prioritně se řeší nejlevější operátor. A pokud zkoumaná část formule neobsahuje žádný operátor, jedná se o list. Formule $\neg p \vee p$ se tedy nejdříve rozpadne na dvě podformule pomocí binárního operátoru $\vee$ a až v synovské rekurzi se vytvoří uzel pro $\neg$ a následně termy p (postup tvoření parsovacího stromu lze nahlédnout v pseudokódu 5.1). Další zvyklostí je vynechávat nejvnější závorky formule. Pro jednodušší implementaci rekurze jsou při přípravě vstupu pro stavbu parsovacího stromu přidány. Formule $\neg p \vee p$ je přeložena na $[\neg p \vee p]$. Podobně jsou při preprocessingu vstupu odděleny za sebou jdoucí kvantifikátory tečkou, aby bylo možné snadno určit, ke které proměnné se vážou. Tedy formule $\neg\exists x \exists y.T(x, y)$ je přeložena na $[\neg\exists x.\exists y.T(x, y)]$.

Posledním problémem, který bylo potřeba v rámci vkládání formulí vyřešit, byla naplnění požadavku na snadnou rozšiřitelnost aplikace. Používaná množina logických symbolů se může v budoucnu změnit. Symboly jsem tedy nadefinoval v rámci samostatného souboru. Každý symbol obsahuje čtyři atributy – svoje id, vlastní symbol, typ symbolu a náhradní znak. Ukázka pro konjunkci je v 5.2. Pro přidání nového operátoru stačí zadefinovat tyto čtyři vlastnosti a veškerá další logika se upraví sama.

Výpis kódu 5.2 Definice konjunkce jako speciálního symbolu.

```
    id: "conjunction",
    symbol: "∧",
    type: "binary",
    replacement: "&"
```

5.2 Informace o stavu aplikace během procesu dokazování

Během procesu dokazování si komponenta dokazovací obrazovky udržuje mnoho informací o stavu systému prostřednictvím state hooks.

Hlavní stavovou informací je *proofState*. Tento hook si udržuje důležité informace o všech aktivních větvích důkazu. Každá větev obsahuje následující informace:

- **formulaToProve**: aktuálně dokazovaná formule
- **facts**: pole známých faktů
- **wasNegationUsed**: informace o tom, zda ve větvi bylo použito pravidlo negace
- **lemma**: informace, zda v dané větvi probíhá důkaz lemmatu
- **disjunctionAssumedFact**: informace, zda daná větev obsahuje fakt vzniklý jako jeden z případů pravidla disjunkce

Objekt důkazního stavu obsahuje i další informace. Například historii důkazu, použité identifikátory, či informaci, zda je důkaz hotový.

Další podstatná množina state hooks udržuje informace o proměnných, které řídí životní cyklus konektoru. Především zda má být viditelný, kde je jeho počátek a konec nebo jaký má aktuálně vybraný blok vnější symbol.

Třetí velkou množinou jsou informace o stavu modálů. Ke každému modálu existuje state hook, který říká, jestli je modál otevřený. Některé modály mají navíc další hooks, které potřebují ke svému správnému fungování

State hooks v každé z těchto množin jsou sdružené do jednoho objektu a jsou předávány funkcím společně. Jelikož funkce obvykle používají více objektů ze stejné množiny, společné předávání zpřehledňuje předpisy těchto funkcí.

5.3 Vizualizace aplikace odvozovacích pravidel

Při návrhu vizualizace aplikace odvozovacích pravidel jsem nejvíce vycházel ze způsobu, jakým tento problém řeší The Incredible Proof Machine. Zobrazení důkazu pomocí bloků propojených konektory je pro uživatele nejintuitivnější. I cílová skupina uživatelů, studenti softwarového inženýrství, jsou s takovýmto uživatelským rozhraním dobře srozuměni z nákresů softwarových diagramů.

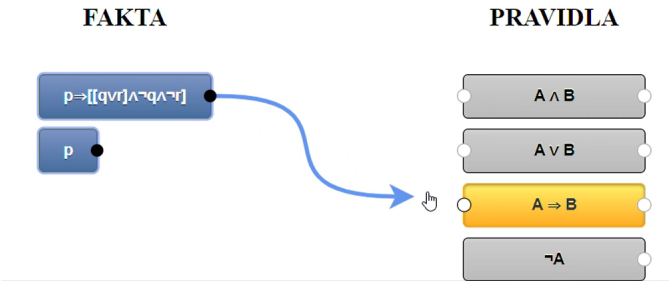
Specifikum dokazovacího systému této práce je existence znalostní báze. Pro většinu logických symbolů díky tomu existují dvě rozdílná odvozovací pravidla v závislosti na tom, zda jsou aplikována na známý fakt, nebo na dokazovanou formuli. Tuto skutečnost je nutno reflektovat v uživatelském prostředí, aniž by se stalo nepřehledným. V návrhu uživatelského prostředí A.2 jsem nejdříve rozdělil obrazovku na dvě části, kde v jedné se zobrazují fakta a v druhé je aktuální formule k dokázání. To umožňuje uživateli vizuálně jasně od sebe odlišit tyto dva rozdílné druhy bloků.

Dále bylo potřeba vyřešit, jakým způsobem zobrazovat propojování bloků. Bylo potřeba, aby použité řešení umožňovalo aktivně vykreslovat spojení mezi vybraným blokem a kurzorem myši. Pro vykreslování spojení jsem si vybral knihovnu React-Xarrows, která umožňuje vykreslovat komponenty šipek Xarrow mezi různými elementy DOMu. Tyto šipky lze pomocí props graficky upravovat. Navíc knihovna poskytuje komponentu Xwrapper a hook useXarrow. V momentě změny pozice šipky pomocí useXarrow jsou poté selektivně přerendrovány pouze synovské komponenty Xwrapperu. Díky tomu je aktivní vykreslování levnější na výpočetní výkon. Jelikož knihovna umožňuje šipkami spojovat elementy DOMu, není možné přímo nastavit, aby šipka následovala aktivně kurzor. Lze toho ale docílit použitím další knihovny a to React-Draggable. Ta pomocí komponenty Draggable umožní přesouvat všechny její synovské komponenty. Tento přesun probíhá pomocí CSS transformací. Pro vytvoření efektu šipky následující kurzor je pak nutné uvnitř komponenty Xwrapper vytvořit neviditelný element obalený komponentou Draggable. Tento neviditelný element překrývá jiný v uživatelském rozhraní viditelný objekt. Jakmile uživatel na takový objekt klikne a zahájí táhnutí, zobrazí se šipka, jež má počátek ve spodním viditelném objektu a konec ve vrchním neviditelném objektu. Použití hooku useXarrow poté zajistí překreslení šipky během přesouvání neviditelného elementu. Vzniká tak vizuální efekt, že šipka aktivně následuje samotný kurzor. JSX tohoto řešení lze nahlédnout v 5.3, kde vrchní komponenta Xwrapper obsahuje prostor pro vykreslování spojení a samotné spojení Xarrow. Dále lze vidět rendrovací funkce předané synovským komponentám FactArea a ProveArea, ve kterých se nacházejí Draggable komponenty. Tyto komponenty určují, zda lze šipku vidět a jakou má pozici.

V další fázi navrhování jsem upustil od zobrazování odvozovacích pravidel pro fakta a formuli k dokázání odděleně. Jelikož většinou pro ten stejný symbol existují oba druhy pravidel, je možné obě spojit do jednoho bloku. To sníží vizuální zatížení uživatele a poskytuje větší prostor knihovně faktů a dokazované formuli. Navíc jsem tyto pravidla přesunul do středu, aby je uživatel nemusel pokaždé nejdříve přetahovat na své místo.

Výpis kódu 5.3 Zjednodušený JSX dokazovací stránky.

```
<Xwrapper>
    <div className="mainArea">
        <FactArea toggleArrow={startDrawingArrow}
            arrowPosition={arrowPosition}
            handleDragEnd={stopDrawingArrow}/>
        <RuleArea/>
        <ProveArea toggleArrow={startDrawingArrow}
           arrowPosition={arrowPosition}
            handleDragEnd={stopDrawingArrow}/>
    </div>
    {isArrowVisible && <Xarrow start={startElement} end={endElement}/>}
</Xwrapper>
```



Obrázek 5.1 Aktivní pravidlo implikace s aktivním slotem na straně faktů. Ostatní pravidla a sloty jsou neaktivní.

Pro snazší navigaci v důkazu je uživatel při vytváření konektoru vizuálně informován, které pravidlo může v současné chvíli použít. Každý blok pravidla má jeden, nebo dva aktivní sloty pro připojení konektoru. V momentě, kdy je započteno tvoření nového konektoru, blok pravidla obdrží informaci, ze kterého bloku toto spojení vychází. Pokud tento blok má jiný nejvnější symbol, než je symbol daného pravidla, dojde ke změně CSS třídy pravidla na její disabled verzi. Zároveň pokud spojení vychází z bloku faktu, dojde ke změně CSS třídy konektoru, který se nachází na straně formule k dokázání. Stejně to funguje v opačném případě: 5.1.

Spojení je úspěšně vytvořeno v případě, že při dokončení táhnutí šipky, se se kurzor nachází nad validním pravidlem. V případě, že se tak nestalo, šipka je zneviditelněna a hooky sledující stav spojení jsou vráceny na svoji výchozí hodnotu.

5.4 Logika aplikace odvozovacího pravidla

Funkční požadavky FP2a až FP2g popisují nutnost transformace stavu důkazu po aplikaci odvozovacího pravidla. Samotná transformace má několik fází. Transformace se spouští v momentě, kdy uživatel klikne na pravidlo s existujícím konektorem.

V první fázi je ověřeno, že systém k aplikaci pravidla nepotřebuje žádné další informace. Pravidla, které potřebují další informace, jsou pravidla pro kvantifikátory, pravidlo ekvivalence a pravidlo disjunkce, pokud je vstup pravidla formule k dokázání. V případě ekvivalence má uživatel na výběr, zda chce vstupní formuli rozšiřovat o dvojitou negaci, nebo naopak dvojitou negaci odebírat. U disjunkce zase může uživatel určit, zda chce předpokládat $\neg A$ a dokazovat B, nebo naopak. Pro pravidla existenčních kvantifikátorů je potřeba vědět, jakým výrazem má být nahrazována daná proměnná. Pokud systém zjistí, že potřebuje další informace, spustí se modál pro doplnění těchto informací.

Následuje fáze, ve které dochází k transformaci stavu důkazu. Ta samotná má tři podfáze. Transformace nemusí procházet všemi podfázemi, pokud to definice pravidla nevyžaduje.

Nejdříve je aplikována speciální logika pravidel. Tuto speciální logiku mají pravidlo ekvivalence, pravidla existenčních kvantifikátorů a pravidlo implikace, pokud je jejich vstupem fakt. Pravidlo ekvivalence není odvozovací pravidlo v pravém slova smyslu, tudíž je nutné jeho aplikaci řešit mimo obvyklou transformaci. V této fázi je tedy přidána dvojitá negace, případně je odebrána, pokud vstup dvojitou negaci jako svůj nejvnější symbol obsahuje. V případě existenčních kvantifikátorů je vyhodnoceno, zda výraz, který uživatel zadal, splňuje požadavky dle definice použitého pravidla. Tedy zda se jedná o novou konstantu/validní term. Pokud tomu tak je, dojde k substituci. U pravidla implikace je zkontrolováno, že jeho předpoklad se už nachází v knihovně faktů.

Pokud aplikování speciální logiky proběhne úspěšně, dochází k přesunu do další podfáze. V ní je rozšířena knihovna faktů dle definice použitého pravidla. V případě pravidla disjunkce, kdy jeho vstupem je fakt, dochází k vytvoření nové větve důkazu. V původní větvi je předpokládáno A, kdežto v nové větvi je předpokládáno B. Při vytváření nové větve důkazu je vytvořena hluboká kopie současné větve s informacemi o znalostní bázi a formuli k dokázání.

V poslední podfázi je transformována formule k dokázání do své jednodušší podoby. V případě použití pravidla konjunkce dochází k vytvoření nové větve důkazu. V původní větvi je nutné dokázat A, zatímco v nové je potřeba dokázat B.

Po ukončení transformace směřuje její výstup do poslední fáze. V ní dochází k ověření, zda transformace nevedla v aktuální větvi k dokončení důkazu. Důkaz je dokončen, pokud se dokazovaná formule objevila ve znalostní bázi nebo pokud se ve znalostní bázi vyskytuje kontradikce. Následuje aktualizace dokazovací obrazovky na transformovaný stav důkazu. Pokud navíc byl důkaz dokončen, objeví se modál, který oznamuje vyřešení dané větve důkazu. V rámci tohoto modálu je zkontrolováno, zda existuje další nedokázaná větev. V případě, že neexistuje, je uživatel při zavření modálu přesměrován na obrazovku, která zobrazuje průběh důkazu. Pokud ale existuje, je zkontrolováno, že nově aktivní větev důkazu se nenachází ve stavu dokázáno. K tomuto může dojít například v případě, že původní větev dokazovala lemma, jehož přidání do další větve způsobilo dokončení důkazu v této větvi. Pokud se tak stalo, je otevřen nový modál oznamující dokázání větve. Sekvenční diagram takto provedených kroků je k dispozici v C.

5.4.1 Rozšiřitelnost odvozovacích pravidel

Logika aplikace odvozovacích pravidel je napsána způsobem, aby bylo možné přidávat novou logiku pouhým upravením jejich definice. Každé pravidlo je definováno pomocí šesti základních vlastností:

- **id**: unikátní identifikátor pravidla.
- **symbolId**: unikátní identifikátor logického symbolu, se kterým je pravidlo spojeno.
- **label**: label zobrazený na bloku pravidla.
- **infoForFacts**: obsah zobrazovaného tooltipu, pokud je vstupem pravidla známý fakt.
- **infoForProof**: obsah zobrazovaného tooltipu, pokud je vstupem pravidla dokazovaná formule.
- **symbolsHaveToMatch**: boolean určující, zda lze pravidlo aplikovat pouze v případě, že symbol pravidla je stejný jako vnější symbol bloku, který je vstupem pravidla.

Těchto šest vlastností se stará především o správné vykreslování pravidel.

Navíc definice obsahuje další dvě vlastnosti obsluhující aplikaci logiky odvozovacího pravidla. Jsou to vlastnosti **inferenceRuleFact** a **inferenceRuleProof**. Při aplikaci pravidla je

použita jedna z těchto vlastností v závislosti na tom, zda je vstupem pravidla známý fakt, nebo dokazovaná formule. Obě tyto vlastnosti obsahují následující objekty:

- **addToFacts**: list podformulí, které se mají přidat do znalostní báze.
- **transformProof**: nová podformule k dokázání.
- **extraLogicFncId**: unikátní id speciální logiky, která má být aplikována před transformací stavu.

Při transformaci je odebrán ze vstupní formule vnější symbol, čímž se odhalí nová podformule. Tato podformule je v objektech **addToFacts** a **transformProof** zapsána jako A. Pokud se pravidlo týká binárního symbolu, způsobí jeho aplikace také odhalení podformule B. Nejsnáze lze pak překlad jazyka definic na logiku aplikace odvozovacích pravidel vysvětlit na příkladech.

Výpis kódu 5.4 Definice pravidla implikace.

```
inferenceRuleFact: {
    addToFacts: ["B"],
    extraLogicFncId: "implicationRule"
},
inferenceRuleProof: {
    addToFacts: ["A"],
    transformProof: ["B"]
}
```

První uvádím definici pravidla implikace v 5.4. Objekt **inferenceRuleProof** pro definici pravidla, kdy je vstupem dokazovaná formule, obsahuje v **addToFacts** A a v **transformProof** B. Znamená to, že pro původní formuli ve tvaru $A \implies B$ logika pro transformaci stavu důkazu přidá A mezi fakta a B nastaví jako novou formuli k dokázání. Zatímco objekt **inferenceRuleFact** pro definici pravidla, kdy je vstupem fakt, obsahuje i pole **extraLogicFncId**. To zajistí, že před samotnou aplikací pravidla aplikace najde pomocí unikátního id *implicationRule* konkrétní implementaci extra logiky. Ta v tomto případě ověří, že se A, již nachází mezi fakty. Pokud je to pravda, spustí se samotná transformace, kde se přidá B do znalostní báze.

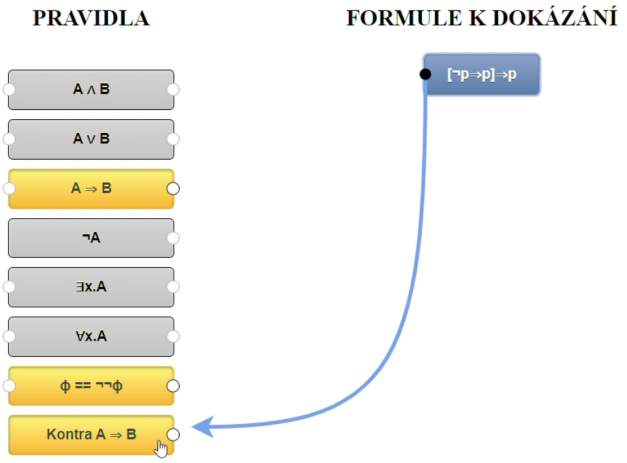
Výpis kódu 5.5 Definice pravidla konjunkce.

```
inferenceRuleFact: {
    addToFacts: ["A", "B"]
},
inferenceRuleProof: {
    transformProof: ["A", "B"]
}
```

Dalším příkladem je definice pravidla konjunkce: 5.5. Objekt **inferenceRuleFact** obsahuje dvouprvkové pole, které říká, že mezi fakta je třeba přidat A i B. Nicméně **inferenceRuleProof** obsahuje také dvouprvkové pole, ale pro vlastnost **transformProof**. Jelikož v jeden moment lze mít vždy pouze jednu dokazovanou formuli, překladač definic to vyhodnotí jako pokyn pro vytvoření nové větve důkazu, ve které se dokazuje B. V původní větvi se dokazovaná formule transformuje z $A \wedge B$ na A.

Výpis kódu 5.6 Definice pravidla disjunkce.

```
inferenceRuleFact: {
    addToFacts: ["case A", "case B"]
},
inferenceRuleProof: {
    addToFacts: [["not A"], ["not B"]],
    transformProof: [["B"], ["A"]]
}
```



Obrázek 5.2 Aktivní pravidlo obměněné implikace s aktivním slotem na straně formule k dokázání.

Nejkomplikovanějším příkladem je definice pravidla disjunkce: 5.6. Objekt **inferenceRuleFact** ve poli **addToFacts** obsahuje klíčové slovo *case*. To překladači říká, že fakta nemají být přidávané do stejné větve. Pokud překladač definic narazí na druhý výskyt *case* ve stejném poli, vytvoří se nová větev, kam se mezi fakta přidá podformule následující toto slovo. Objekt **inferenceRuleProof** obsahuje další dvě syntaktické vlastnosti. První z nich je, že **addToFacts** a **transformProof** obsahují v poli další vnořená pole. To umožňuje, aby to stejné pravidlo provádělo odlišnou transformaci v závislosti na uživatelské volbě. Uživatel si tedy může vybrat, zda aplikace pravidla přidá mezi fakta $\neg A$ a změní dokazovanou formuli na B, nebo naopak přidá k faktům $\neg B$ a dokazovanou formulí se stane A. Druhou novou vlastností je výskyt klíčového slova *not*. To zapříčiní, že se do kořene parsovacího stromu podformule, která následuje toto slovo, přidá uzel negace.

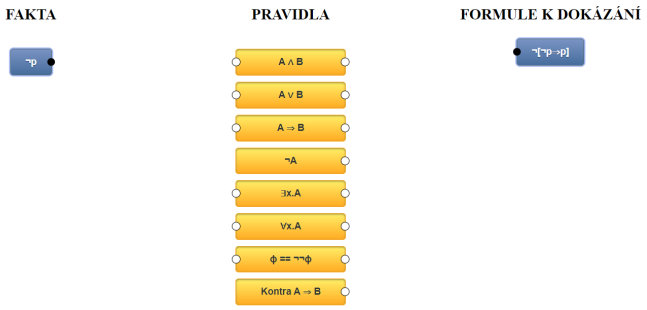
Přidání nového pravidla pouze rozšířením definice budu demonstrovat na příkladu. V tomto příkladu chci přidat pravidlo pro obměnu implikace. Takovéto pravidlo by aplikovalo obměněnou verzi běžné implikace. Definice takového pravidla je v 5.7.

Výpis kódu 5.7 Definice pravidla pro obměněnou implikaci.

```
{
    id: "contraImplicationRule",
    symbolId: "implication",
    label: "Kontra A ⟹ B",
    name: "Obmenena implikace",
    infoForProof: "Nejdrive obmeni implikaci.
        Predpokladejme negaci B a dokazujme negaci A.",
    symbolsHaveToMatch: true,
    inferenceRuleProof: {
        addToFacts: ["not B"],
        transformProof: ["not A"]
    }
}
```

Toto pravidlo se následně přidá na dokazovací obrazovku a rendrovací logika pro spojování s bloky probíhá korektně, viz 5.2.

Po aplikování pravidla dochází k očekáván změně stavu důkazu přidáním $notB$ mezi fakta a transformací dokazované formule na $notA$, viz 5.3.



Obrázek 5.3 Původní formule k dokázání $[\neg p \implies p] \implies p$ se po aplikaci pravidla obměněné implikace změnila na $\neg[\neg p \implies p]$ a mezi fakta se přidal výraz $\neg p$.

5.5 Vkládání lemmatu

Mnohé důkazy vyžadují, aby aplikace umožňovala během dokazování výroku dokázat lemma. Tato skutečnost je i reflektována jako funkční požadavek FP3. Aplikace tedy v rámci dokazovací obrazovky obsahuje tlačítko *Přidat lemma*. To otevře modál, ve kterém se nachází komponenta pro vkládání logické formule do systému. Při vložení nového lemma je vytvořena nová větev důkazu, která obsahuje dosud známá fakta a formule k dokázání se nahradí zadaným lemmatem. Lemma je identifikováno unikátním id. Po dokázání lemmatu je lemma přidáno mezi známá fakta větve důkazu, ve které bylo zadáno. V rámci dokazování lemmatu je možné přidávat další lemmata nutná pro dokázání daného lemmatu.

5.6 Historie důkazu

Pro implementaci požadavku FP4 je nutné v paměti udržovat historii průběhu důkazu. Systém zaznamenává tři různé kroky v průběhu historie důkazu, které mění stav důkazu.

Prvním z nich je krok typu *ruleApplied*. Tento krok se zapíše do paměti v momentě, kdy je úspěšně aplikováno nějaké z pravidel. Tento krok obsahuje informace o stavu důkazu (*proofState*) předtím, než bylo pravidlo aplikováno. Následně tento krok obsahuje informace o tom, jaké pravidlo bylo použito a co bylo vstupem pravidla.

Druhým typem je krok *lemmaAdded*. Tento krok se zapisuje do paměti v momentě, kdy je vytvořena nová větev důkazu pro dokazování lemmatu. Tento krok opět uchovává informace o stavu důkazu před přidáním lemmatu a informace o přidaném lemmatu.

Posledním typem je krok *proofFinished*. Tento krok se zapisuje do paměti v momentě, kdy je dokončen důkaz jedné větve. Obsahuje informace o tom, jak vypadal stav důkazu v momentě dokončení dokazování větve, zda se jednalo o poslední dokazovanou větev, jakým způsobem bylo dokazování dokončeno a jak vypadá původní dokazované tvrzení, které uživatel zadal do systému.

Pro naplnění požadavku FP4 jsem přidal tlačítko *Krok zpět*. Po kliknutí na toto tlačítko se postupně maže konec historie důkazu, dokud se na jejím konci neobjeví krok typu *ruleApplied* nebo *lemmaAdded*. V ten moment se přečtou informace o stavu důkazu uloženém v tomto kroku a současný stav důkazu se nahradí tímto stavem. Tento krok se následně také smaže z historie, čímž dojde ke kompletnímu návratu do stavu, před poslední akcí uživatele, která změnila stav systému.

5.7 Nahlédnutí do průběhu důkazu po jeho dokončení

Funkční požadavky FP5 a FP6 požadují, aby aplikace umožňovala uživateli po dokončení důkazu se podívat na provedené kroky. To je umožněno díky uchovávání historie důkazu z předchozí sekce. Uživatel poté má na výběr ze dvou různých zobrazení historie důkazu. Běžný textový přepis pouze uživateli vypíše provedené kroky, jako by se díval do logu programu. Z uživatelského hlediska je zajímavější lineární vizualizace důkazu.

V lineární vizualizaci uživatel vidí kroky zobrazené vizuálně zajímavým způsobem. Navíc kroky typu *ruleApplied* lze rozkliknout a podívat se do jejich detailů na změnu stavu, kterou provedly. Objekt *ruleApplied* obsahuje stav před aplikací pravidla. Další krok v historii obsahuje stav po aplikaci tohoto kroku. Takže systému stačí nahlédnout do dalšího kroku, aby mohl uživateli ukázat transformaci stavu po provedení pravidla. Navíc v detailu kroku se uživateli zobrazují až tři značky. Tyto značky jsou:

- **Hledání kontradikce** - Tato značka je aktivní, pokud bylo ve větvi důkazu použito pravidlo negace. Značí skutečnost, že probíhá hledání kontradikce mezi fakty.
- **Lemma** - Tato značka je aktivní, pokud daná větev důkazu dokazuje lemma.
- **Případ disjunkce** - Tato značka je aktivní, pokud bylo ve větvi důkazu použito pravidlo disjunkce na známý fakt. Značí skutečnost, že v současné době se mezi známými fakty nachází jedna z podformulí vstupního faktu a že se jedná o důkaz jednoho z případů.

Navíc lineární vizualizace důkazu ještě umožňuje skrývat důkazy lemmat. Jelikož všechny kroky důkazu jednotlivých lemmat mají v sobě zaznamenáno unikátní identifikátor lemmatu, který dokazují, lze je jednoznačně identifikovat. Krok typu *lemmaAdded* obsahuje tlačítko, kterým uživatel může skrýt všechny kroky až do kroku *proofFinished*, který reprezentuje dokončení důkazu tohoto lemmatu. Skrytý důkaz lemmatu lze později opět rozbalit. Při rozbalování jsou skryté kroky přidány zpět do pole zobrazovaných kroků. Pseudokód, zobrazující používání lemmaStorage ke skrývání kroků lemmat, je k vidění v 5.8. Díky ukládání kroků do lemmaStorage lze mít zároveň skryto více vnořených lemmat. Například pokud důkaz lemmatu A obsahuje i důkaz lemmatu B a uživatel následně skryje důkaz lemmatu B a následně lemmatu A. Pokud se rozhodne rozbalit zpět důkaz vnějšího lemmatu A, tak lemma B bude stále skryté, protože jeho kroky jsou uloženy separátně od kroků lemmatu A (i když je také obsahuje).

5.8 Načítání aktivního důkazu

Je důležité, aby při aktualizaci stránky či zavření prohlížeče uživatel nepřišel o veškerou práci. Pro naplnění této potřeby je využit web storage prohlížeče. Web storage umožňuje aplikacím ukládat data v paměti prohlížeče ve formě párů klíč-data. Z uživatelského hlediska se jedná o bezpečné řešení, jelikož nedochází k odesílání soukromých dat na server. Aplikace musí akorát ohlídat, že zde nebude ukládat citlivá data, jelikož obsah web storage může přečíst i jakákoliv jiná aplikace.

Při používání web storage je nutné se zamyslet nad možností XSS útoku, jelikož jeho obsah může i libovolná aplikace přepsat. Protože ale systém používá uložená data pouze pro vykreslení bloků, nikoliv pro načítání jakékoliv spustitelné logiky či nastavování odkazů, stačí ověřit, že samotné vykreslení není k tomuto útoku náchylné. Naštěstí React při rendrování DOM elementů takto přijaté informace ošetřuje. Jsou vždy považovány za textový řetězec, nikdy za spustitelný skript.

Informace o stavu aplikace se ukládá poprvé po zadání formule k dokázání (5.9). Následně vždy když dojde k uživatelem iniciované změně stavu aplikace, tedy při aplikaci pravidla a při zadání lemmatu. Naposledy se informace ve web storage přepíšou při dokončení důkazního procesu.

Výpis kódu 5.8 Pseudokód pro skrývání a načítání kroků lemmatu.

```
function toggleLemma(lemmaId, isShown):
    // Lemma je v soucasne chvili rozbalene, tudiz chceme kroky skryvat
    pokud isShown:
        foundLemmaStart = false
        zobrazovaneKroky = []
        newLemmaStored = {
            lemmaId: lemmaId,
            steps: []
        }

        pro kazdy krok v historii dukazu:
            pokud foundLemmaStart:
                pokud krok je typu "proofFinished"
                a pokud krok dokocil dukaz lemmatu z argumentu funkce:
                    foundLemmaStart = false
                    zobrazovaneKroky.push(step)
                else:
                    newLemmaStored.steps.push(step)
            else:
                zobrazovaneKroky.push(step)
                pokud krok je typu is "lemmaAdded"
                a pokud se jedna o lemma z argumentu funkce:
                    foundLemmaStart = true
        lemmaStorage.save(newLemmaStored)
        return zobrazovaneKroky
    // Lemma je v soucasne chvili skryte, tudiz chceme jeho kroky nacist
    jinak:
        zobrazovaneKroky = []

        pro kazdy krok v historii dukazu:
            zobrazovaneKroky.push(step)
            pokud krok je typu is "lemmaAdded"
            a pokud se jedna o lemma z argumentu funkce:
                zobrazovaneKroky.push(step)
                zobrazovaneKroky.push(lemmaStorage.load(lemmaId))

        lemmaStorage.remove(lemmaId)
        return zobrazovaneKroky
```

Výpis kódu 5.9 Ukládání formule do localStorage po jejím zadání do systému.

```
export const storeInfoAboutNewFormula = (formula, formulaId) => {
    const proofState = [{
        formulaToProve: formula,
        facts: [],
        wasNegationUsed: false,
        lemma: {
            id: undefined,
            value: undefined
        },
        disjunctionAssumedFact: undefined
    }]
    localStorage.setItem("formula", JSON.stringify(formula))
    localStorage.setItem("proofState", JSON.stringify(proofState))
    localStorage.setItem("formulaId", JSON.stringify(formulaId))
    localStorage.setItem("proofAssistantStage", "proving")
}
```

V případě, že uživatel otevře aplikaci s nedokončeným důkazem, aplikace nejdříve přečte obsah web storage. Pokud web storage obsahuje příznak, že poslední důkaz nebyl dokončen, je uživatel přesměrován na danou obrazovku, která si načte všechny potřebné údaje. Pokud jsou očekávaná data ve web storage poškozená nebo chybějí, aplikace ho přesměruje zpět na hlavní obrazovku a smaže příznak, že existuje rozdělaný důkaz.

5.9 Testování

Při psaní testů pro zajištění očekávaného běhu aplikace jsem se zaměřoval na testování logiky funkcí. Pro chování komponent vzhledem k uživatelským akcím nebyly testy implementovány, jelikož se jedná o část kódu, která je obvykle často měněna. Při změně této logiky tedy vždy musí dojít i k opravě veškerých testů. Navíc takovéto testy přináší za vloženou práci pouze velmi malý užitek. Soustředil jsem se spíše na testování těch částí kódu, které používají složitou logiku a řídí stav aplikace. Testy vytvořené pro takovou logiku pak poskytují určitou záruku, že aplikace běží očekávaně. Navíc tato jádrová logika neprochází tak často změnami jako samotné UI, takže je nutné testy měnit s menší frekvencí.

Tabulka pokrytí testy (5.1) zobrazuje celkové pokrytí a pak pokrytí adresářů obsahující logiku, která řídí stav aplikace.

Tabulka 5.1 Pokrytí testy

Název adresáře	Procentuální pokrytí
src	40%
src/components/formulaInserting/src/logic	86%
src/components/modals/src/logic	54%
src/components/proofProcess/src/logic	79%
src/components/proofRecap/src/logic	75%

Kapitola 6

Použití aplikace a testování

V této kapitole jsou čtenáři představeny příklady použití aplikace a je seznámen s výsledky uživatelského testování.

6.1 Příklad z výrokové logiky

V prvním příkladu ukážu použitelnost aplikace na příkladu z výrokové logiky. Budu s pomocí aplikace dokazovat $p \implies [[q \vee r] \wedge \neg q \wedge \neg r]] \implies \neg p$.

Nejdříve je formule nahrána do systému. Při kreslení konektoru je uživatel naváděn, které pravidlo může použít 6.1.

Po aplikování pravidla je stav důkazu změněn dle definice pravidla implikace 6.2.

Po aplikaci pravidla negace bylo možné na známý fakt aplikovat pravidlo implikace (modus ponens). Knihovna znalostí se rozrostla 6.3.

Po aplikaci pravidla konjunkce na známý fakt je možné aplikovat pravidlo disjunkce na známý fakt. To dle definice způsobí rozštěpení důkazu na dvě větve. Uživatel je informován, že v aktuální větvi je předpokládáno q 6.4.

Skutečnost, že došlo k rozvětvení důkazu, je také zobrazena v zásobníku, kde se momentálně nachází dvě důkazní větve 6.5.

Poté, co je dojde k aplikaci pravidla konjunkce na známý fakt $\neg q \wedge \neg r$, se zobrazí modál oznamující dokončení první větve důkazu 6.6.

Je načtena druhá větev důkazu, kde je předpokládáno r. I v této větvi po aplikaci stejného pravidla se stejným vstupem dojde k dokončení důkazu 6.7.

Po dokončení důkazu uživatel může nahlédnout do průběhu důkazu ve vizuálně zajímavé podobě 6.8.

Uživatel si může prohlédnout, jakým způsobem byl důkaz dokončen 6.9.

Pokud chce uživatel pouze vidět soupis použitých kroků, zobrazí si textový log 6.10.

6.2 Příklad s použitím lemmatu

V druhém příkladu ukážu použitelnost aplikace na příkladu, který vyžaduje důkaz lemmatu. V tomto příkladě budu dokazovat $q \implies [[p \wedge q] \vee [\neg p \wedge q]]$.

Formule je nejdříve nahrána do systému 6.11.

Po aplikaci pravidla implikace je uživatel při aplikace disjunkce představen před volbu, kterou podformuli si chce vybrat jako předpoklad 6.12.

Vybraný předpoklad se následně zobrazí mezi fakty 6.13.

Po aplikaci pravidla konjunkce na dokazovanou formuli ihned dojde k dokončení jedné větve důkazu, jelikož se q objevuje i mezi fakty 6.14.

Po aplikaci pravidla negace na známý fakt se mezi fakty nachází q, p a $\neg[p \wedge q]$. Je snadné nahlédnout, že tento stav vede na dokončení důkazu pomocí lemmatu $p \wedge q$ 6.15.

Po přidání lemmatu $p \wedge q$ se aktualizuje formule k dokázání a změní se zásobník důkazu 6.16.

Po aplikace pravidla konjunkce na dokazované lemma dojde k dokončení důkazu lemmatu. Zároveň dojde k dokončení celého důkazu, jelikož dokázané lemma vytvoří kontradikci ve faktech 6.17.

Ve shrnutí důkazu si uživatel může prohlédnout detail kroku, který vedl k dokázání lemmatu 6.18.

Uživatel si také může prohlédnout stav důkazu po tomto kroku 6.19.

Uživatel může v soupisu kroků skrýt všechny kroky, které vedly k dokázání lemmatu 6.20.

6.3 Příklad z predikátové logiky

V třetím příkladu ukážu použitelnost aplikace na příkladu z predikátové logiky. Budu s pomocí aplikace dokazovat $[\exists x.S \implies Q(x)] \implies [S \implies \exists x.Q(x)]$.

Nahrání formule do systému 6.21.

Stav systému po dvou aplikacích pravidla implikace na dokazovanou formuli 6.22

Před aplikací existenčního pravidla na známý fakt je zobrazen modál pro vybrání substituční konstanty 6.23.

Po aplikace existenčního pravidla a modus ponens obsahuje knihovna faktů formuli velmi podobnou té, kterou je potřeba dokázat 6.24.

V knihovně faktů se nachází $Q(a)$. Dokazovanou formuli lze převést na též na $Q(a)$ aplikací existenčního pravidla 6.25.

Po aplikaci tohoto pravidla dojde k dokončení důkazu 6.26.

Uživatel si může prohlédnout průběh důkazu v textovém logu 6.27.

6.4 Uživatelské testování

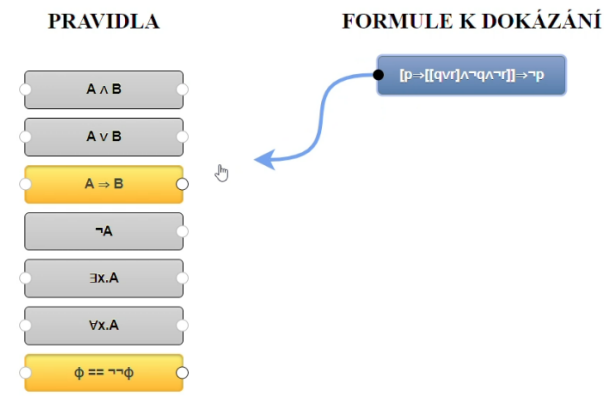
Výše zmíněné příklady byly představeny i pěti uživatelům v rámci uživatelského testování. Uživatelé byli nejdříve seznámeni s dokazovacím systémem aplikace a následně jim byly zadány příklady k dokázání.

Všichni uživatelé se velmi rychle zorientovali ve způsobu, jakým probíhá aplikace pravidel. Díky tomu, že aplikovatelná pravidla se vizuálně odlišují od neaplikovatelných, se uživatelům dařilo velmi rychle aplikovat triviální kroky důkazu. Pozitivní zpětnou vazbu také přineslo zobrazení historie důkazu, která uživatelům přišla přehledná a užitečná. Uživatelé také ocenili design aplikace, který jim přišel přehledný a čistý.

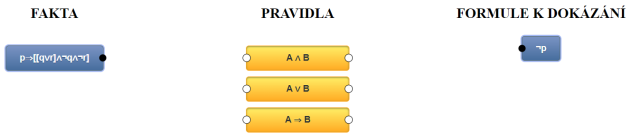
Testování také přineslo informace o funkcích, které by uživatelé ocenili, ale aplikace je nenabízí.

- Uživatel měl problém s trefením se do oblasti, ve které je možné začít kreslit konektor. Na základě této zpětné vazby byla tato oblast zvětšena.
- Uživatel chtěl zavřít modál, ve kterém byla očekávána uživatelská volba. Modál bylo možné zavřít kliknutím mimo něj, ale uživateli tato funkce nebylo zřejmá. Bylo tedy přidáno tlačítko *Zavřít* do všech modálů.
- Uživatel chtěl zrušit nakreslený konektor. Konektor bylo možné zrušit kreslením jiného konektoru. Tato skutečnost ale uživateli nebyla zřejmá. Vzhledem k tomuto bylo přidáno tlačítko, které explicitně konektor maže.

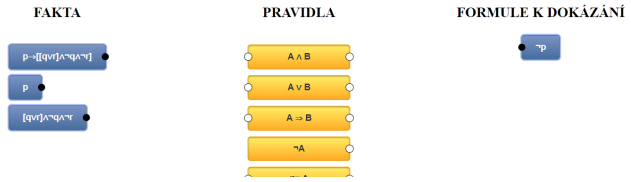
- Uživatel navrhl, že kvůli dlouhé rolovatelné stránce s historií důkazu by bylo uživatelsky přívětivé mít tlačítko pro započetí nového důkazu i na jejím konci. Toto tlačítko bylo přidáno.



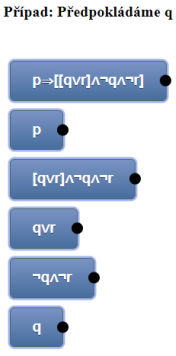
Obrázek 6.1 Nahrání formule do systému.



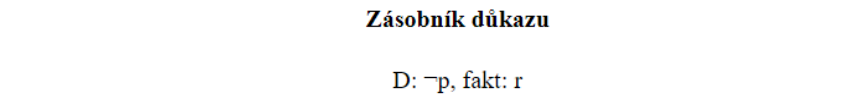
Obrázek 6.2 Aplikace pravidla implikace.



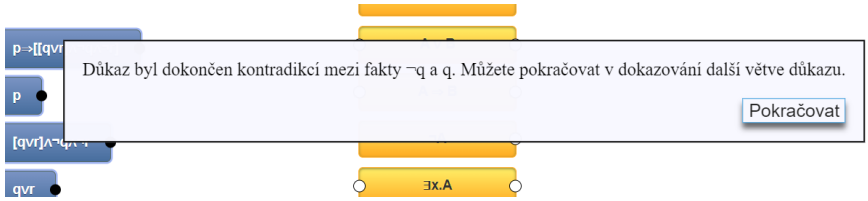
Obrázek 6.3 Knihovna znalostí se rozrostla.



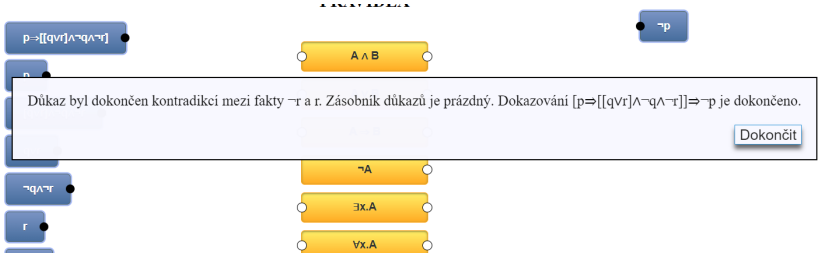
Obrázek 6.4 Rozštěpení důkazu do dvou větví.



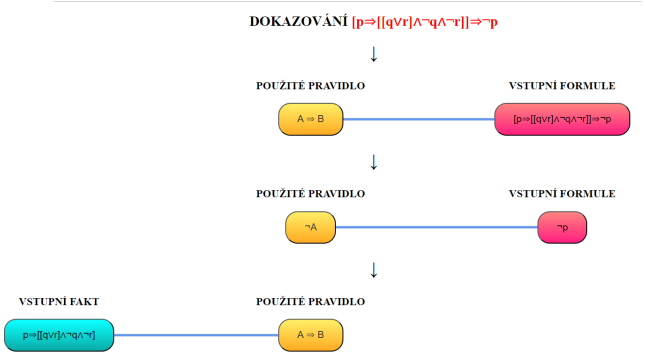
Obrázek 6.5 Obsah zásobníku důkazu.



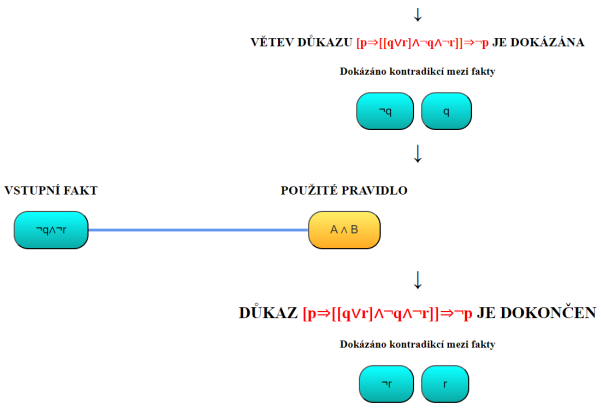
Obrázek 6.6 Dokončení první větve důkazu.



Obrázek 6.7 Dokončení druhé větve důkazu.



Obrázek 6.8 Grafické zobrazení kroků důkazu.



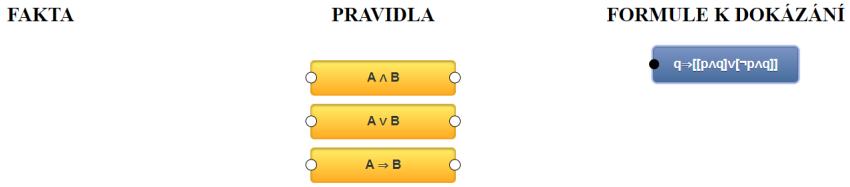
Obrázek 6.9 Způsob dokončení důkazu.

Začátek dokazování [p⇒[[q∨r]∧¬q∧¬r]]⇒¬p

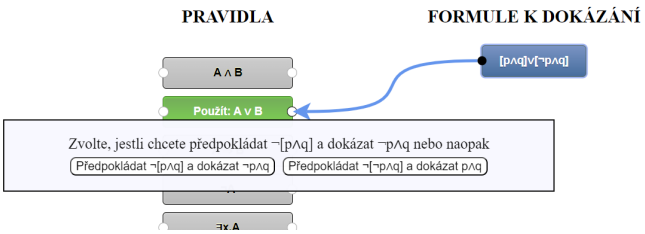
1) Pravidlo implikace bylo použito na [p⇒[[q∨r]∧¬q∧¬r]]⇒¬p jakožto dokazovanou formuli
2) Pravidlo negace bylo použito na ¬p jakožto dokazovanou formuli
3) Pravidlo implikace bylo použito na p⇒[[q∨r]∧¬q∧¬r] z vědomostní báze
4) Pravidlo konjunkce bylo použito na [q∨r]∧¬q∧¬r z vědomostní báze
5) Pravidlo disjunkce bylo použito na q∨r z vědomostní báze
6) Pravidlo konjunkce bylo použito na ¬q∧¬r z vědomostní báze
7) Větev důkazu [p⇒[[q∨r]∧¬q∧¬r]]⇒¬p je dokázána kontradikcí mezi ¬q a q
8) Pravidlo konjunkce bylo použito na ¬q∧¬r z vědomostní báze
9) Důkaz [p⇒[[q∨r]∧¬q∧¬r]]⇒¬p je dokončen kontradikcí mezi ¬r a r

Konec dokazování [p⇒[[q∨r]∧¬q∧¬r]]⇒¬p

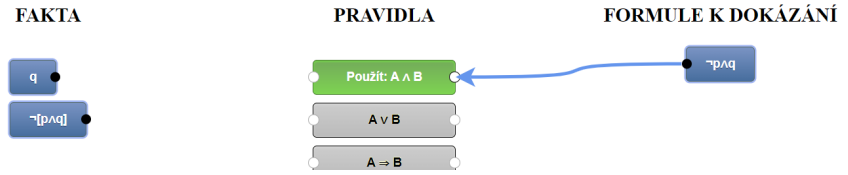
Obrázek 6.10 Textový log důkazu.



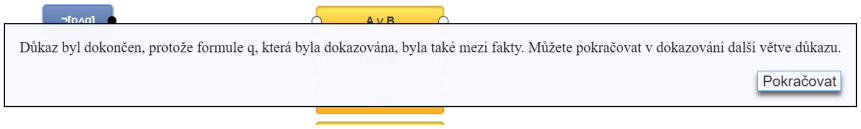
Obrázek 6.11 Formule nahrána do systému.



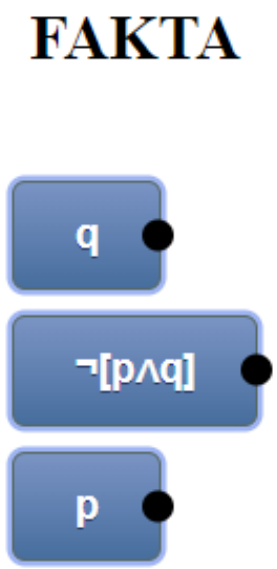
Obrázek 6.12 Aplikace pravidla disjunkce.



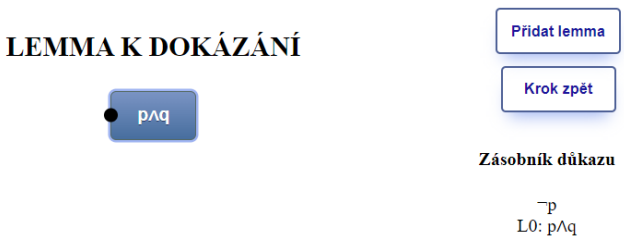
Obrázek 6.13 Předpoklad mezi fakty.



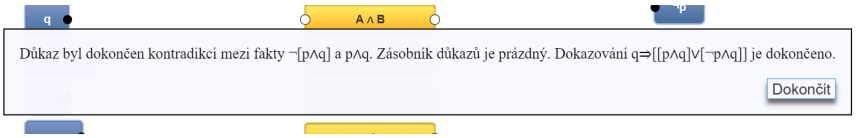
Obrázek 6.14 Dokončení první větve důkazu.



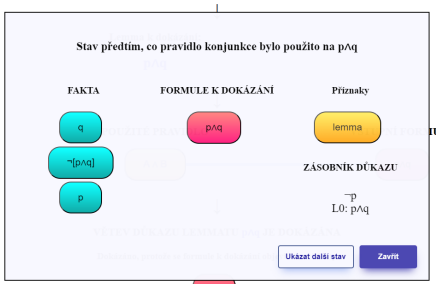
Obrázek 6.15 Knihovna faktů.



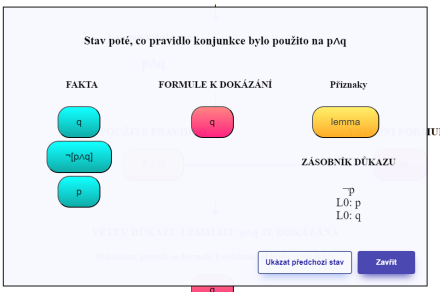
Obrázek 6.16 Přidání lemmatu.



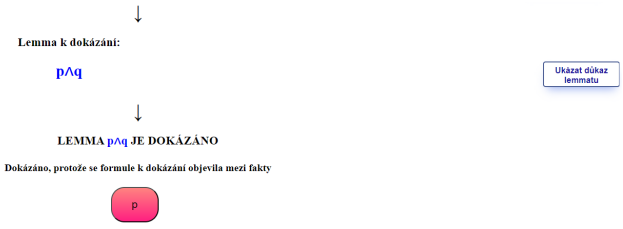
Obrázek 6.17 Dokončení důkazu.



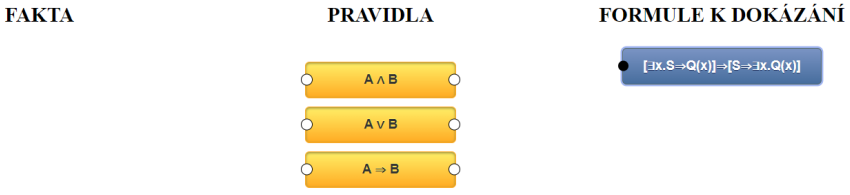
Obrázek 6.18 Detail kroku vedoucího k dokončení lemmatu



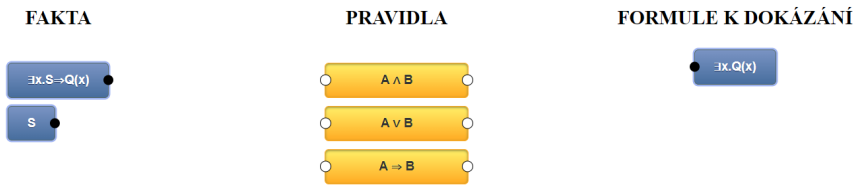
Obrázek 6.19 Stav důkazu po aplikaci odvozovacího pravidla.



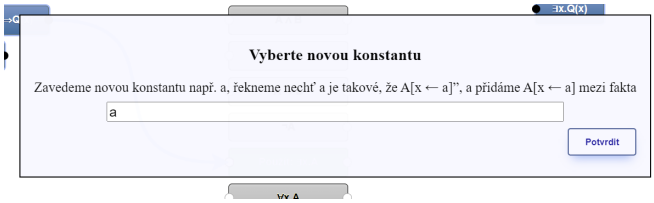
Obrázek 6.20 Lemma se skrytými kroky



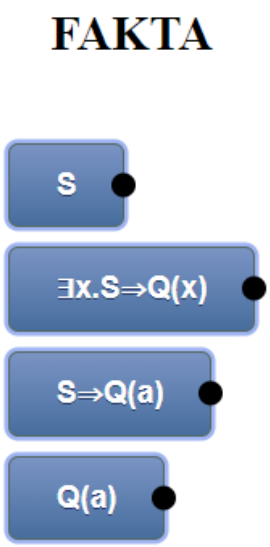
Obrázek 6.21 Formule je nahrána do systému.



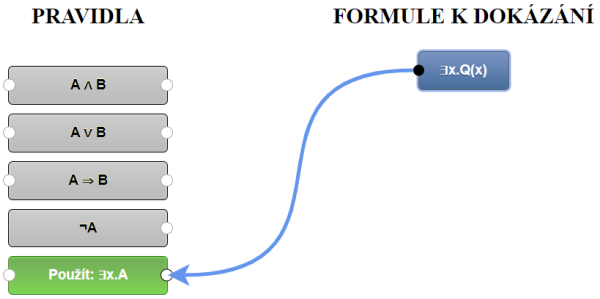
Obrázek 6.22 Stav systému po aplikaci pravidla implikace.



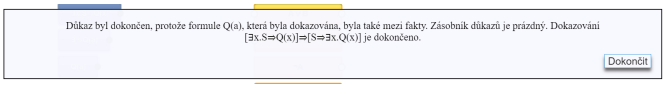
Obrázek 6.23 Modál pro vybrání substituční konstanty.



Obrázek 6.24 Obsah knihovny faktů.



Obrázek 6.25 Výběr existenčního pravidla.



Obrázek 6.26 Dokončení důkazu.

Začátek dokazování [∃x.S⇒Q(x)]⇒[S⇒∃x.Q(x)]

1) Pravidlo implikace bylo použito na [∃x.S⇒Q(x)]⇒[S⇒∃x.Q(x)] jakožto dokazovanou formuli
2) Pravidlo implikace bylo použito na S⇒∃x.Q(x) jakožto dokazovanou formuli
3) Pravidlo existence bylo použito na ∃x.S⇒Q(x) z vědomostní báze
4) Pravidlo implikace bylo použito na S⇒Q(a) z vědomostní báze
5) Pravidlo existence bylo použito na ∃x.Q(x) jakožto dokazovanou formuli
6) Důkaz [∃x.S⇒Q(x)]⇒[S⇒∃x.Q(x)] je dokončen pomocí stejného faktu a formule k dokázání: Q(a)

Konec dokazování [∃x.S⇒Q(x)]⇒[S⇒∃x.Q(x)]

Obrázek 6.27 Textový log důkazu.

Kapitola 7

Současný stav a výhled do budoucna

7.1 Současný stav

V rámci této práce byly naimplementovány všechny funkční požadavky a aplikace splňuje i nefunkční požadavky. Konkrétně se jedná o:

- Vkládání logické formule do systému
- Používání pravidel pro zjednodušování stavu důkazu
- Možnost vkládání lemmat
- Možnost vracet se v důkazu o krok zpět
- Ukládání stavu aplikace do paměti prohlížeče
- Textový log důkazu po jeho dokončení
- Vizualizace důkazu a skrývání lemmat
- Responsivita – aplikace reaguje na uživatelské interakce dostatečně rychle
- Bezpečnost – aplikace je odolná vůči běžným útokům
- Udržovatelnost – aplikace obsahuje dokumentaci
- Rozšiřitelnost – aplikace je napsána modulárně, což usnadňuje její další vývoj

Na letní semestr je naplánován běh předmětu *Formální metody a specifikace*, ve kterém bude zadavatel práce software testovat společně se studenty a případně ho nasadí na školní server. Zatím není jasné, zda bude přístupná pro kohokoliv, či pouze pro studenty FITu. Prozatím ji mám tedy nasazenou pouze soukromě pomocí nástroje Netlify.

7.2 Výhled do budoucna

Aplikace je implementována tak, aby byla snadno rozšiřitelná. Součástí aplikace je dokumentace, která budoucí vývoj usnadní. Nicméně mnohé části kódu jsou připravené pro nové funkcionality. Především přidání nových odvozovacích pravidel je triviální a nevyžaduje téměř žádné zásahy do kódu. Nejvíce požadavků na rozšíření téměř určitě vzejde z testování se studenty v rámci běhu předmětu. Přesto lze vytyčit určité body, kam by se vývoj v budoucnu mohl ubírat:

- **Ukládání dokončených důkazů**: Užitečná funkčnost by byla, kdyby si studenti při přípravě na zkoušku mohli uložit důkazy, které dříve dokončili. Díky tomu by si vždy mohli zobrazit správný postup. Bylo by ale nutné rozmyslet, kam tato řešení ukládat, aby se do systému nevnesla žádná bezpečnostní hrozba.
- **Dokazování vlastností polí a listů**: Předmět Formální metody a specifikace se také zabývá dokazováním vlastností polí a listů. Jistě by šlo jejich dokazování přidat do této aplikace. Je nutné ale nejdříve vyřešit, že dokazování probíhá na základě axiomů namísto odvozovacích pravidel.
- **Volba větve důkazu**: V současné implementaci aplikace neumožňuje uživateli, aby si vybral, kterou větev důkazu chce dokazovat. Přestože uživatel vždy musí dokázat všechny větve a toto omezení mu v tom nebrání, uživatel by mohl preferovat dokázání určité větve přednostně. Pro implementaci této funkcionality je nutné nejdříve napříč aplikací ukládat informaci o aktivní větvi.
- **Možnost pohybovat se v důkazu směrem dopředu**: V současné chvíli po kliknutí na tlačítko Krok zpět je poslední krok nenávratně smazán. Uživatel musí znovu aplikovat pravidlo/přidat lemma, aby se dostal do stavu před kliknutím. Bylo by užitečné, kdyby uživatel mohl smazání kroku vrátit.
- **Stromová vizualizace důkazu**: Samotný důkazní proces probíhá jako strom. V kořeni stromu se nachází dokazované tvrzení a aplikace odvozovacího pravidla dává za vznik novému uzlu. Některá pravidla způsobují rozvětvení tohoto stromu a dokončení důkazu je list takového stromu. Pokud jsou všechny větve zakončeny takovýmto listem, je důkaz dokončen. Implementovaná Lineární vizualizace důkazu je kolapsem všech větví do jediné řady kroků. V případě komplikovaných důkazů by mohlo být užitečné, kdyby si uživatel mohl prohlídnout kroky důkazu opravdu jako strom, ve kterém by bylo možné prohlížet si větve odděleně.

Kapitola 8
Závěr

V rámci této diplomové práce vznikla aplikace pro podporu výuky předmětu Formální metody a specifikace. Aplikace je navržena způsobem, jež studentům usnadňuje práci při používání dokazovacího kalkulu představeného v tomto předmětu. Cílem práce bylo, aby taková aplikace byla funkční, snadná na použití a aby ji bylo možné do budoucna snadno udržovat a rozšiřovat.

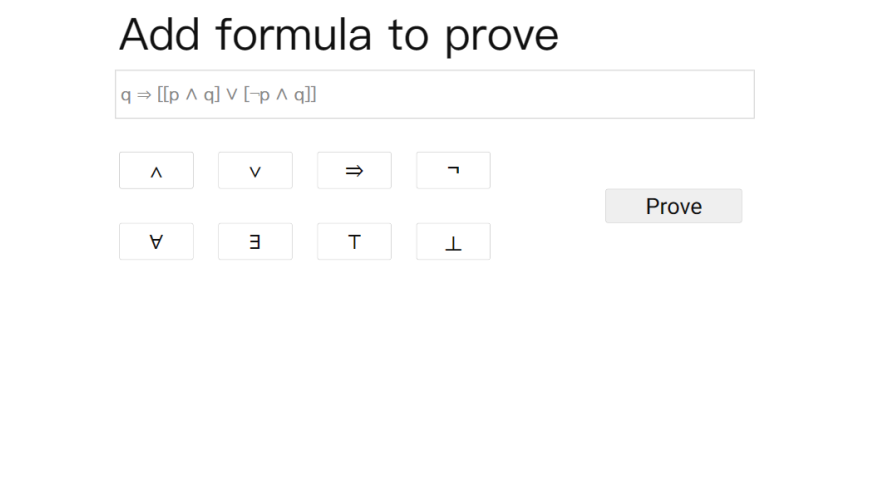
Tyto cíle jsem splnil. Použití aplikace bylo testováno s uživateli, kteří shodně hodnotili, že jim aplikace usnadnila práci se zadaným dokazovacím kalkulem. Navíc, aplikace byla implementována způsobem, který umožňuje ji snadno udržovat a rozšiřovat. Také byla vytvořena dokumentace, která tento proces výrazně usnadní.

Posledním krokem je otestovat schopnosti aplikace jejím zapojením do výuky. Věřím, že studentům usnadní průchod studiem, a třeba některého zaujme tato aplikace natolik, aby pokračoval v jejím vývoji.

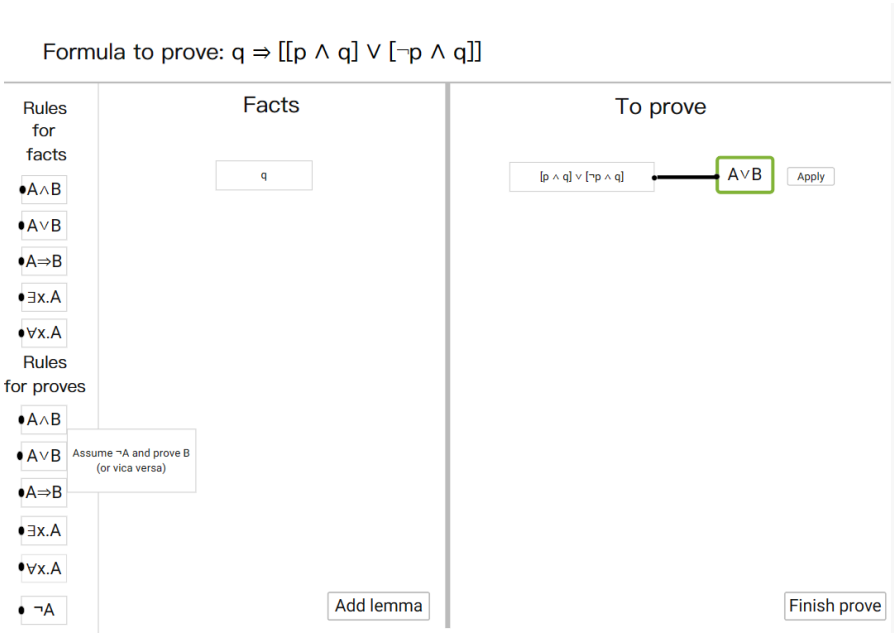
Příloha A

Návrh uživatelského rozhraní

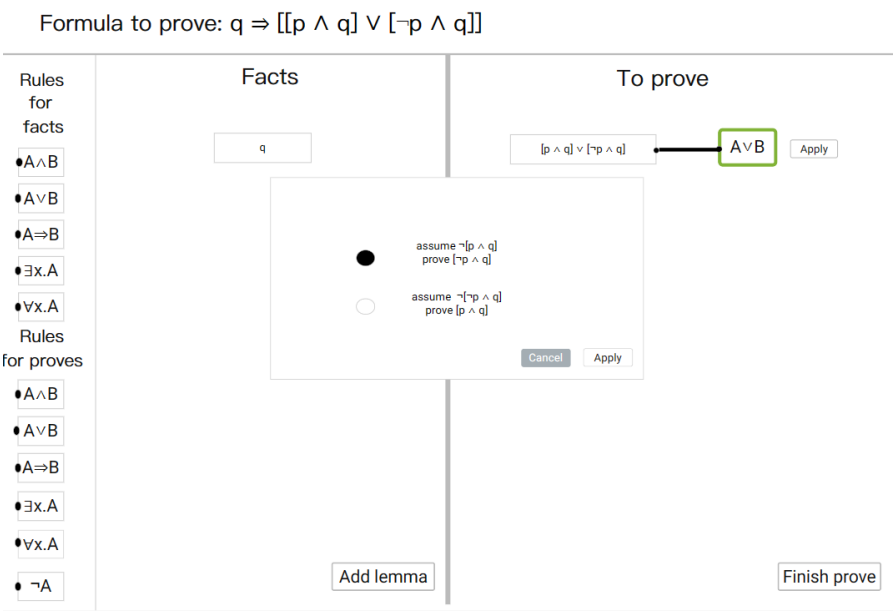
Prototyp uživatelského rozhraní vytvořený v aplikaci Wondershare Mockitt.



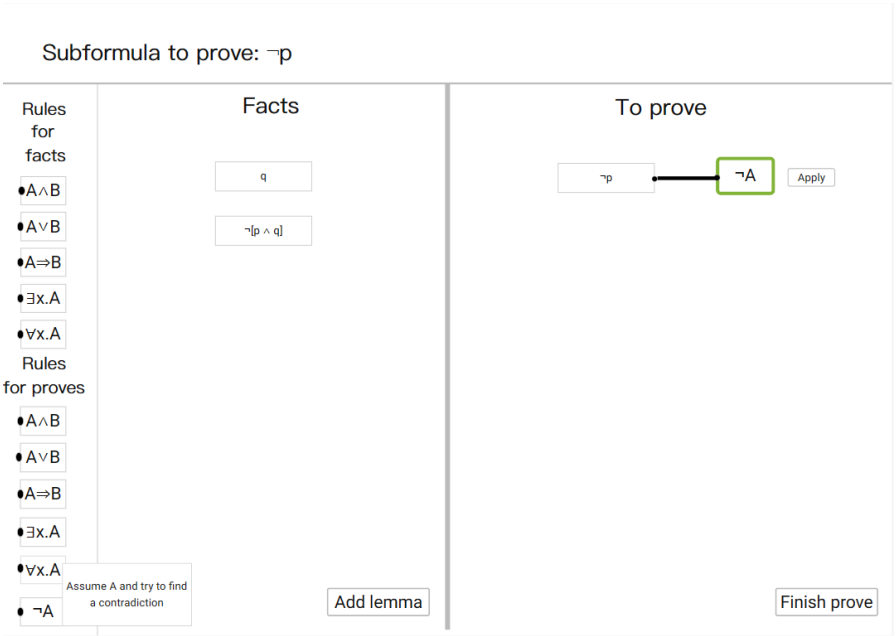
Obrázek A.1 Návrh uživatelského rozhraní obrazovky pro vkládání formulí



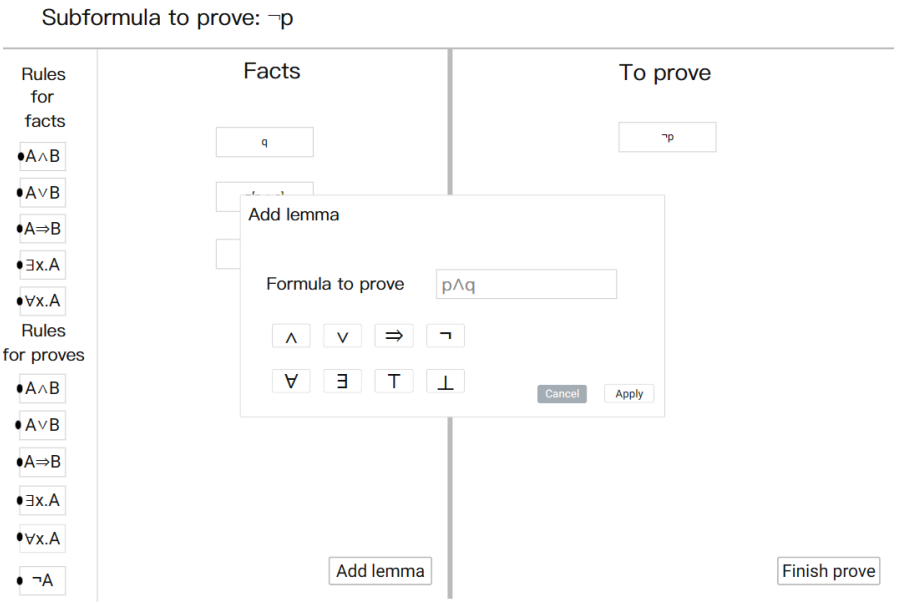
Obrázek A.2 Návrh uživatelského rozhraní obrazovky pro dokazování formulí



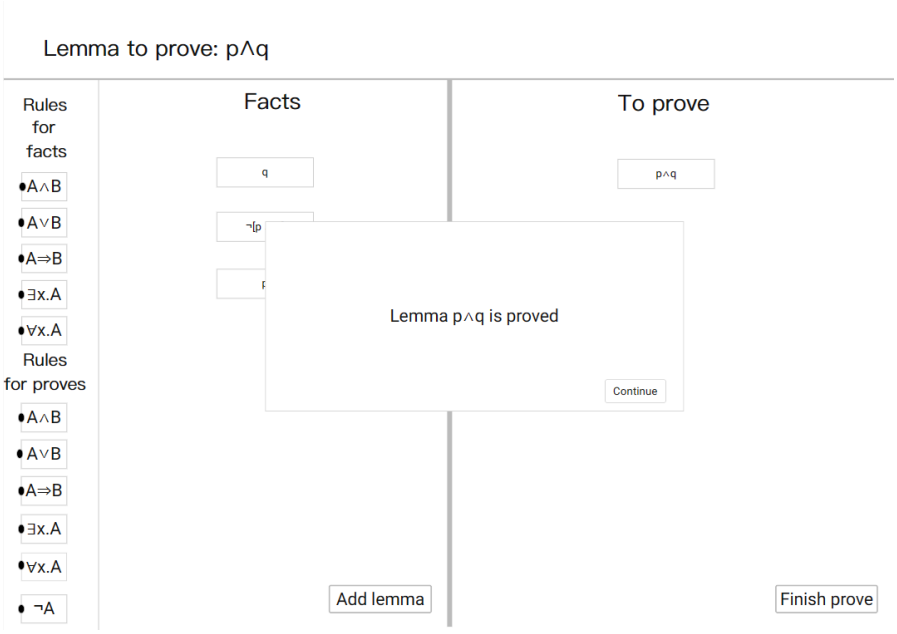
Obrázek A.3 Použití modálu pro uživatelský výběr



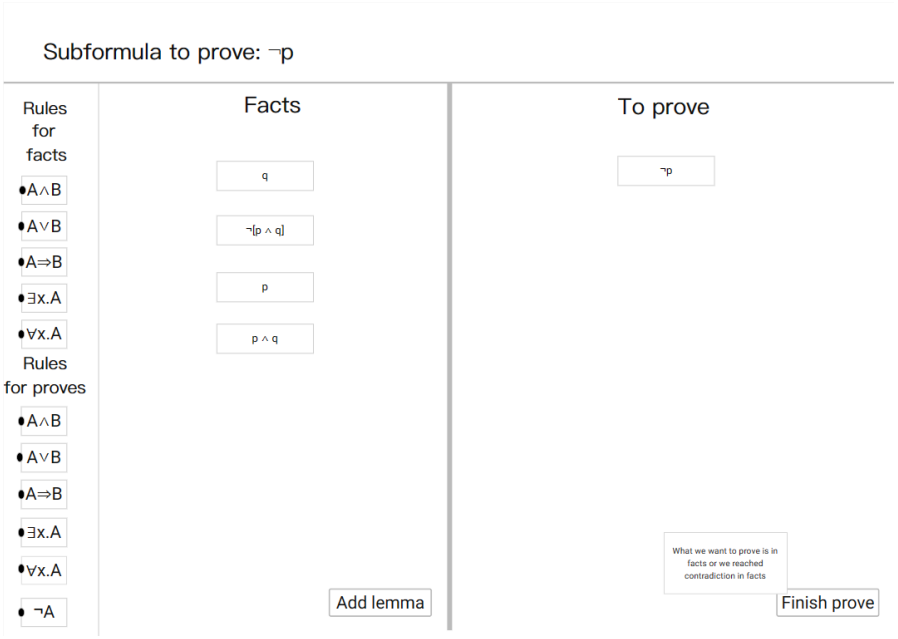
Obrázek A.4 Rozrůstání knihovny faktů



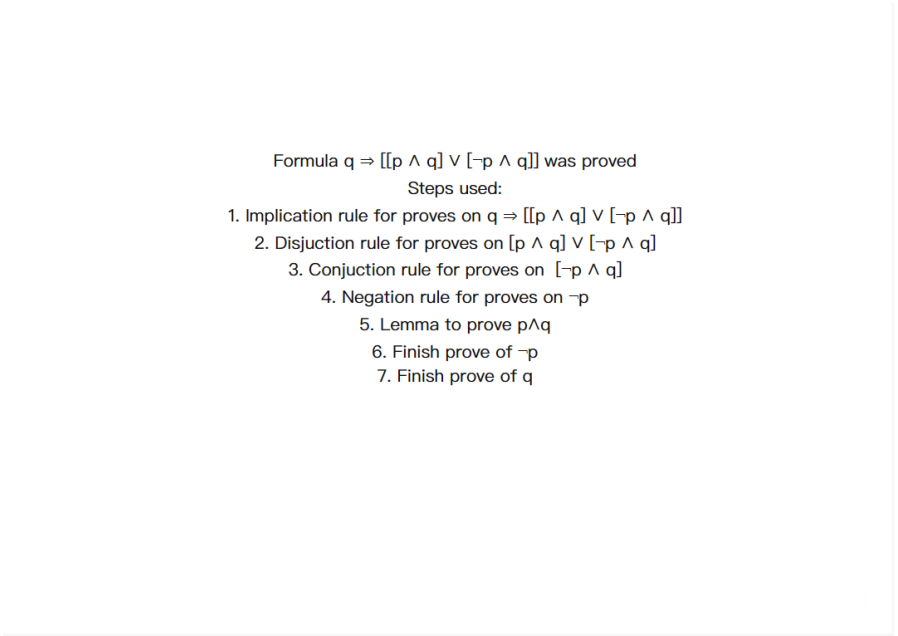
Obrázek A.5 Přidání lemmatu



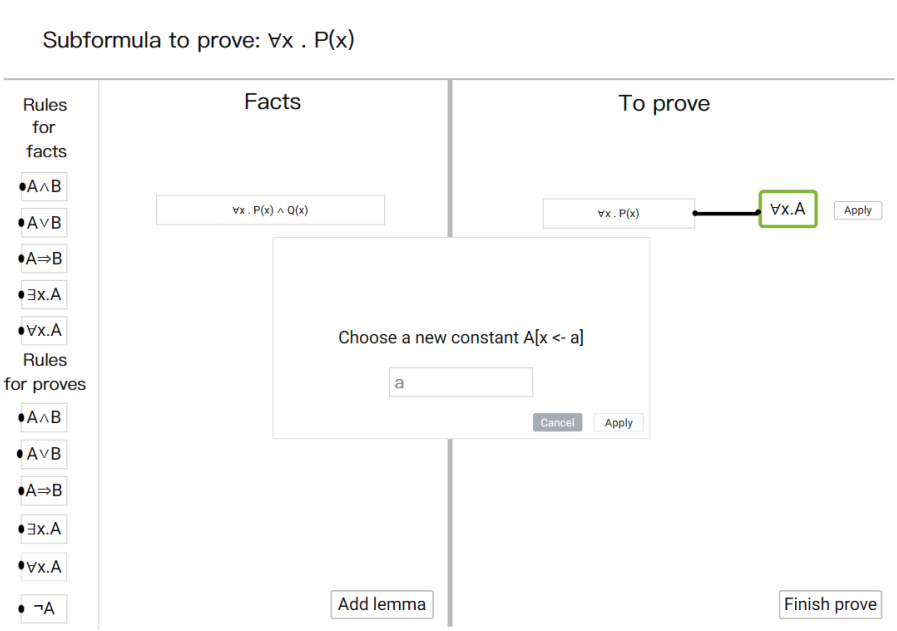
Obrázek A.6 Dokázání lemmatu



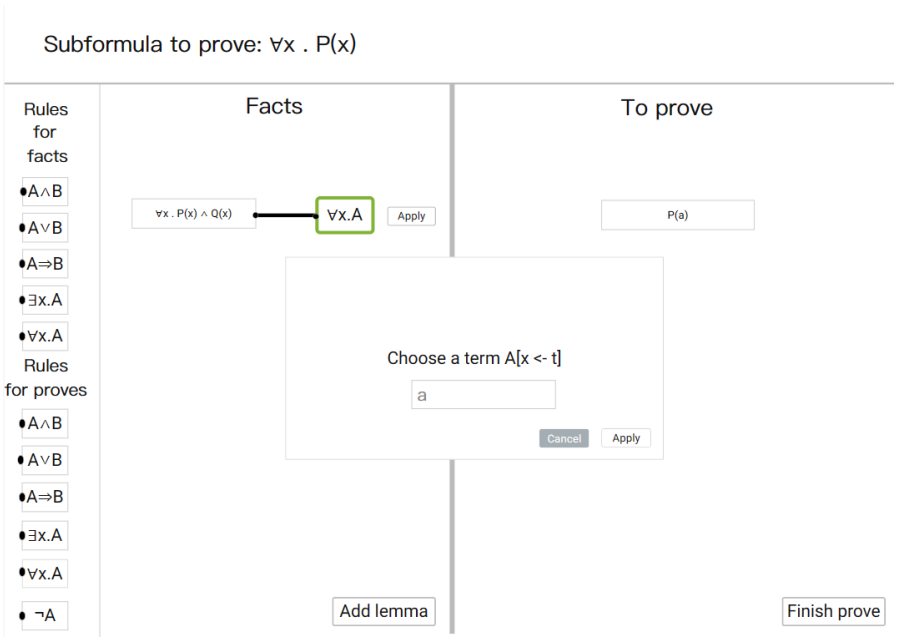
Obrázek A.7 Stav faktů po dokázání lemmatu



Obrázek A.8 Shrnutí postupu důkazu



Obrázek A.9 Přidání nové konstanty

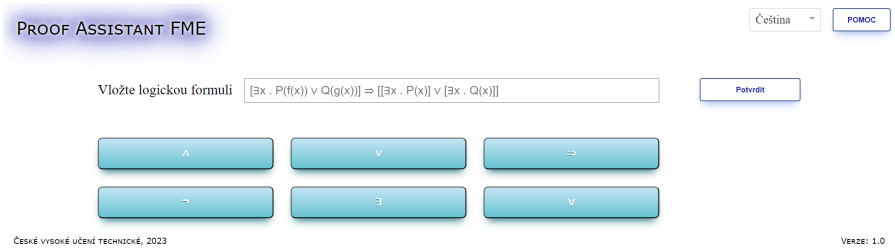


Obrázek A.10 Přidání termu

Příloha B

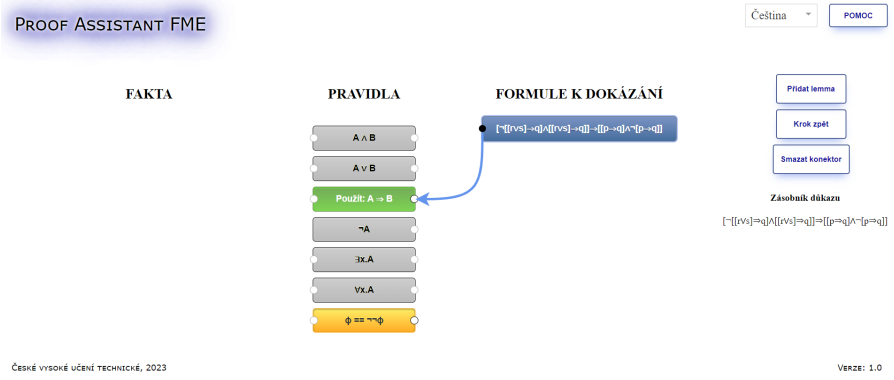
Uživatelské rozhraní

Snímky obrazovek uživatelského prostředí aplikace.

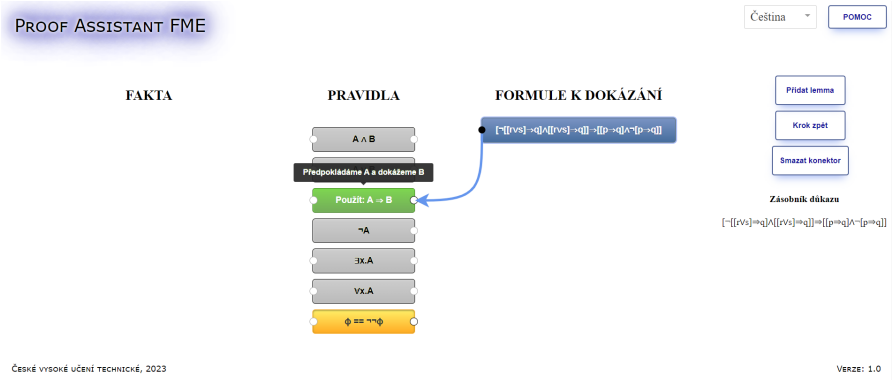


Obrázek B.1 Obrazovka pro vkládání formulí

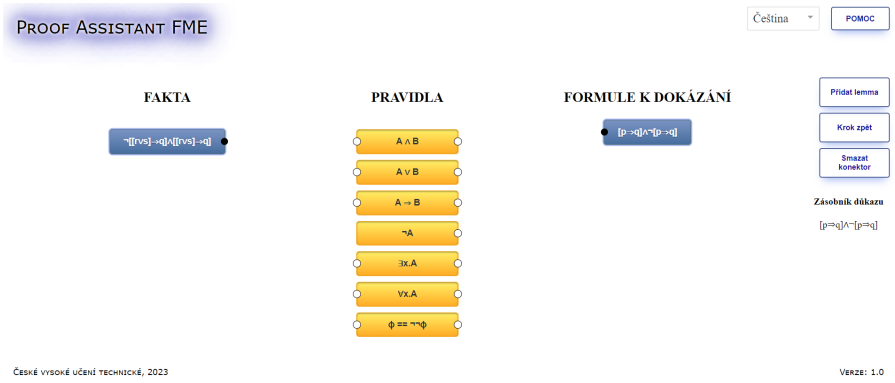
Obrázek B.2 Obrazovka pro dokazování formulí



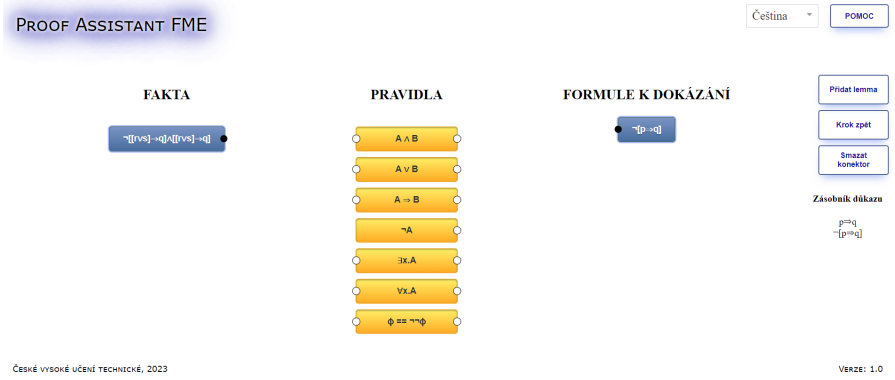
Obrázek B.3 Obrazovka pro dokazování formulí s nakresleným konektorem



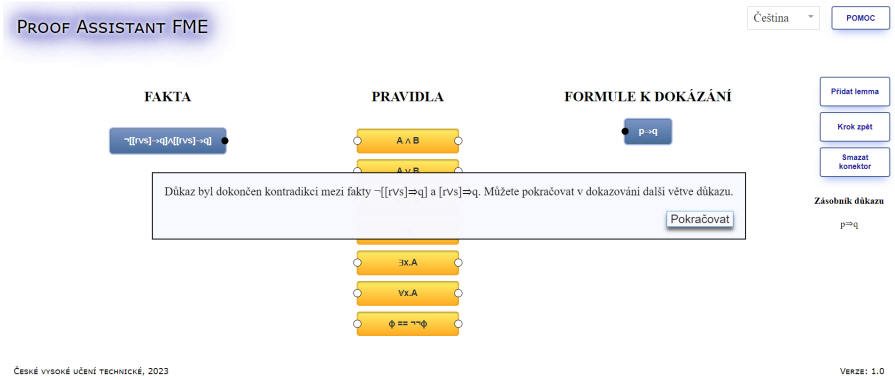
Obrázek B.4 Tooltip ukazující definici odvozovacího pravidla



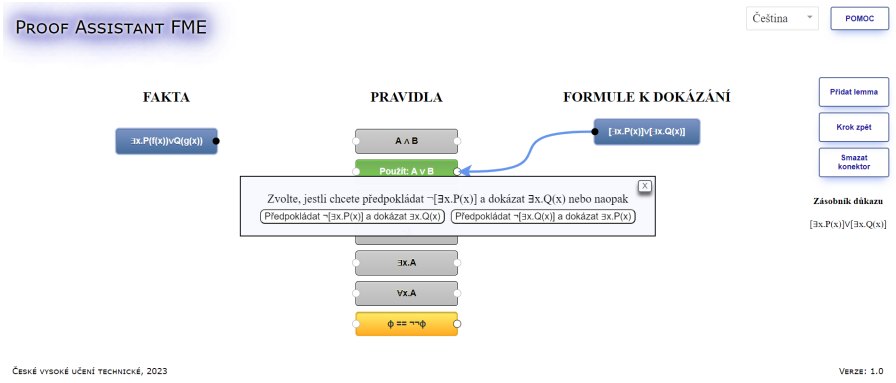
Obrázek B.5 Rozšíření znalostní báze o nový fakt



Obrázek B.6 Rozvětvení důkazu po aplikaci pravidla konjunkce na dokazovanou formuli

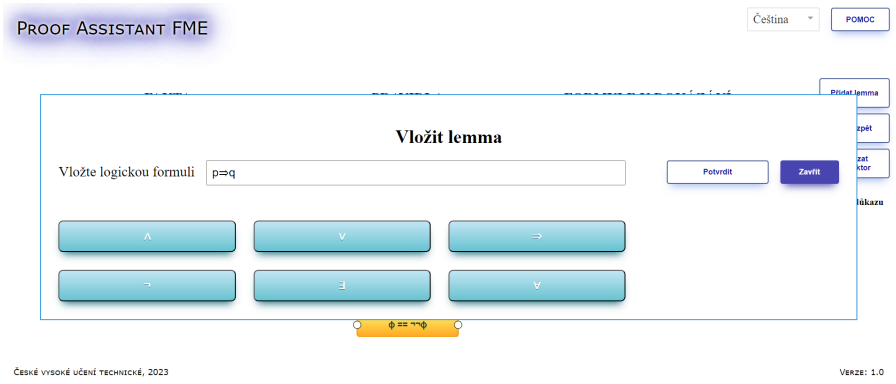


Obrázek B.7 Dokázání větve důkazu

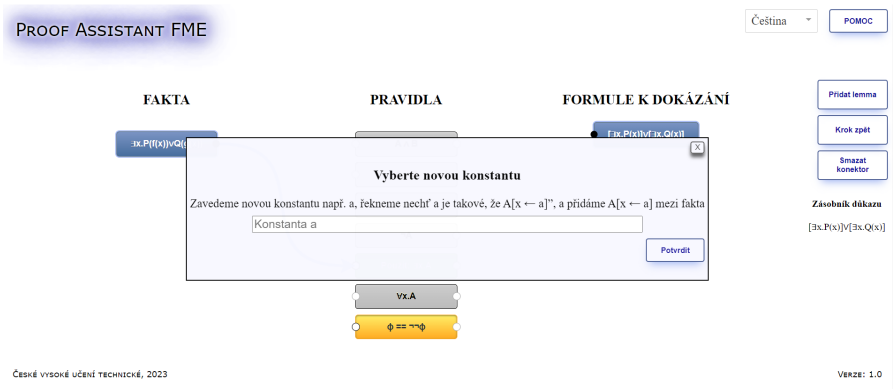


Obrázek B.8 Modál s výběrem pro pravidlo disjunkce

Obrázek B.9 Větev vzniknuvší díky aplikaci pravidla disjunkce na známý fakt



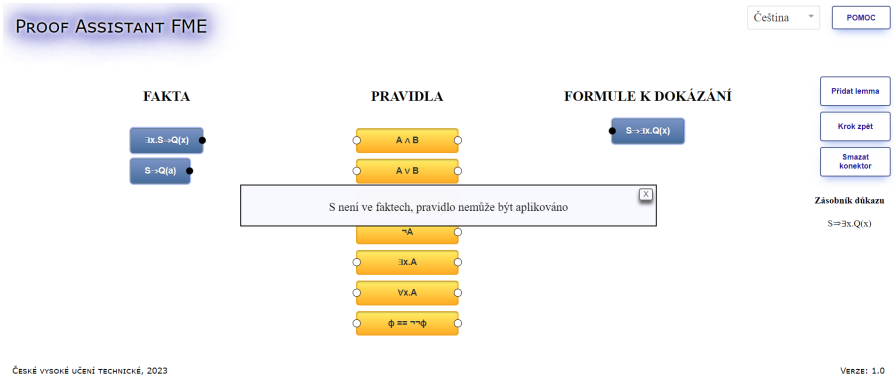
Obrázek B.10 Modál pro přidání lemmatu



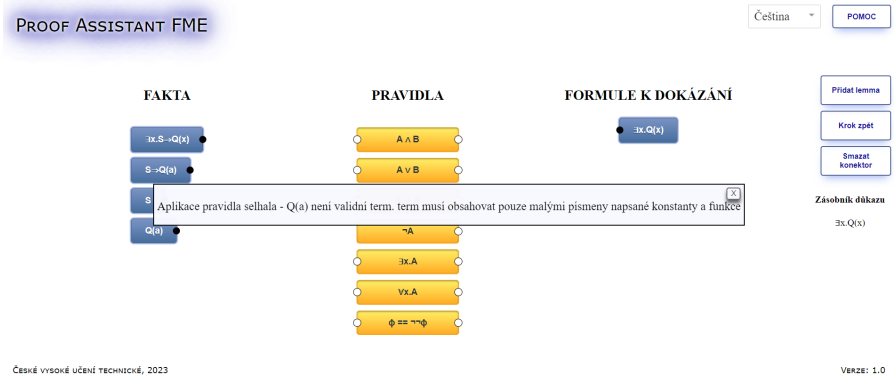
Obrázek B.11 Modál pro substituci novou konstantou

Obrázek B.12 Modál pro substituci novým termem

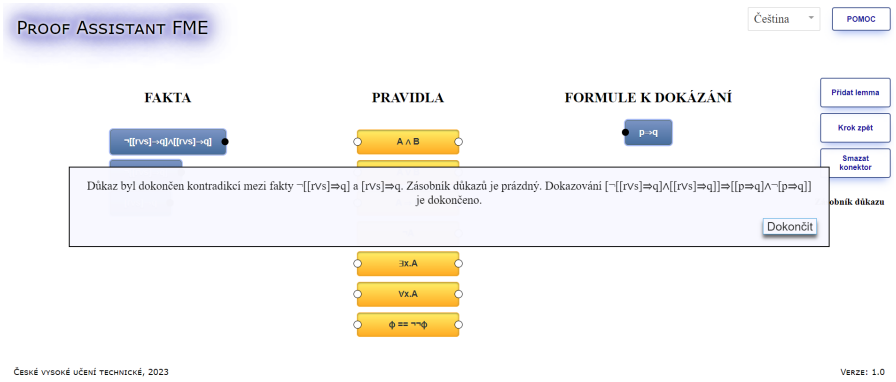
Obrázek B.13 Modál pro substituci novým termem



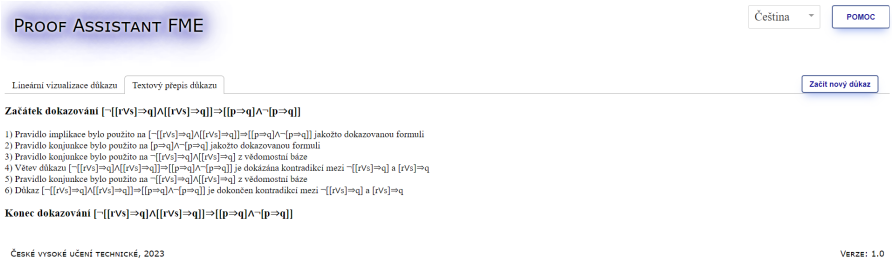
Obrázek B.14 Modál oznamující chybu při aplikaci pravidla implikace na známý fakt



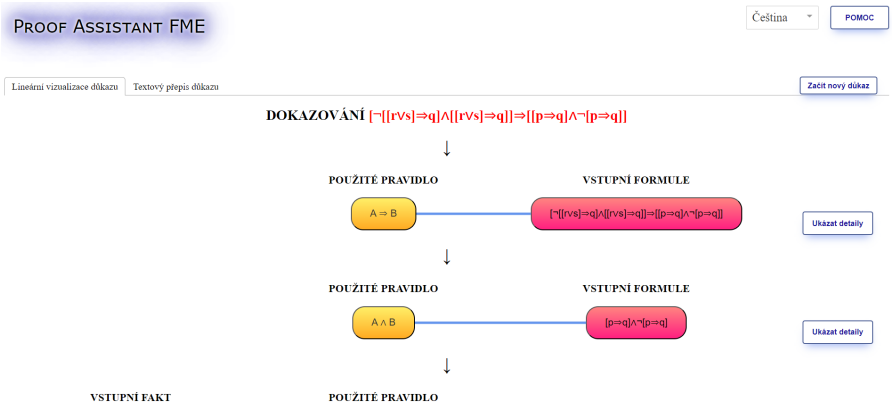
Obrázek B.15 Modál oznamující chybu při substituci proměnné



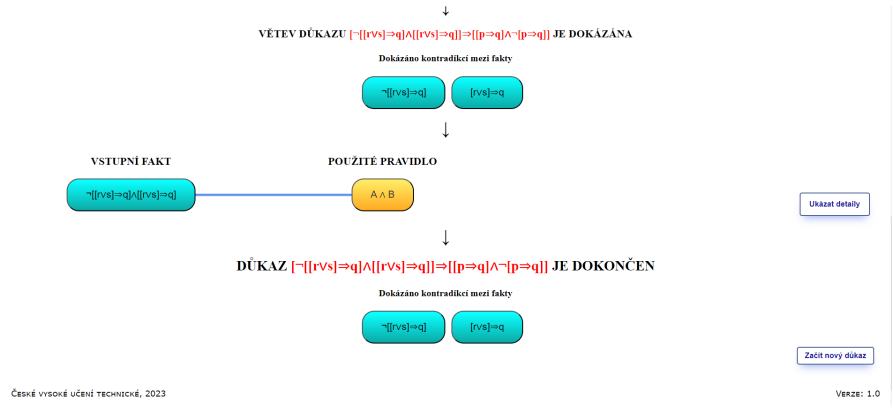
Obrázek B.16 Modál oznamující dokončení důkazu



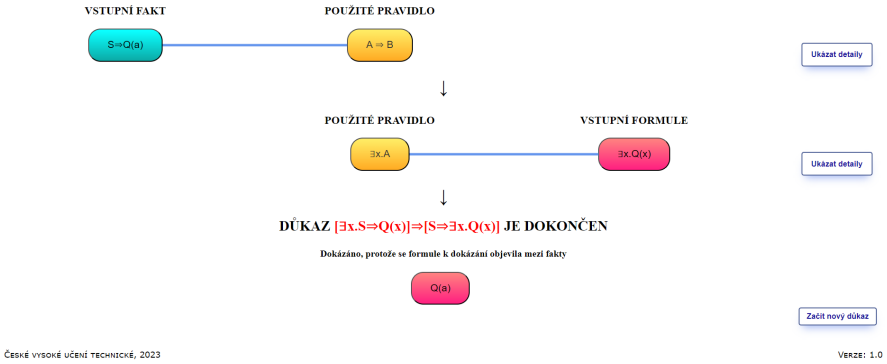
Obrázek B.17 Textový přepis průběhu důkazu



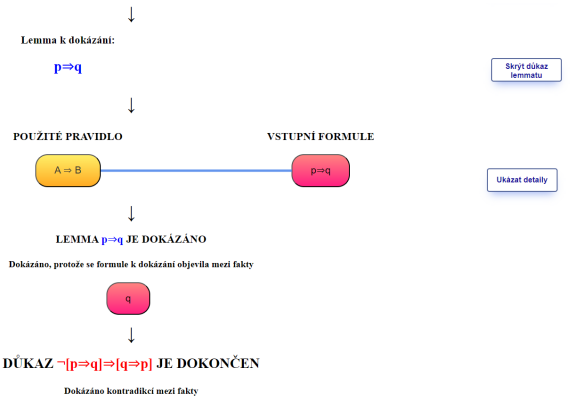
Obrázek B.18 Lineární vizualizace důkazu



Obrázek B.19 Lineární vizualizace důkazu, dokončení důkazu kontradikcí



Obrázek B.20 Lineární vizualizace důkazu, dokončení důkazu shodou s dokazovanou formulí



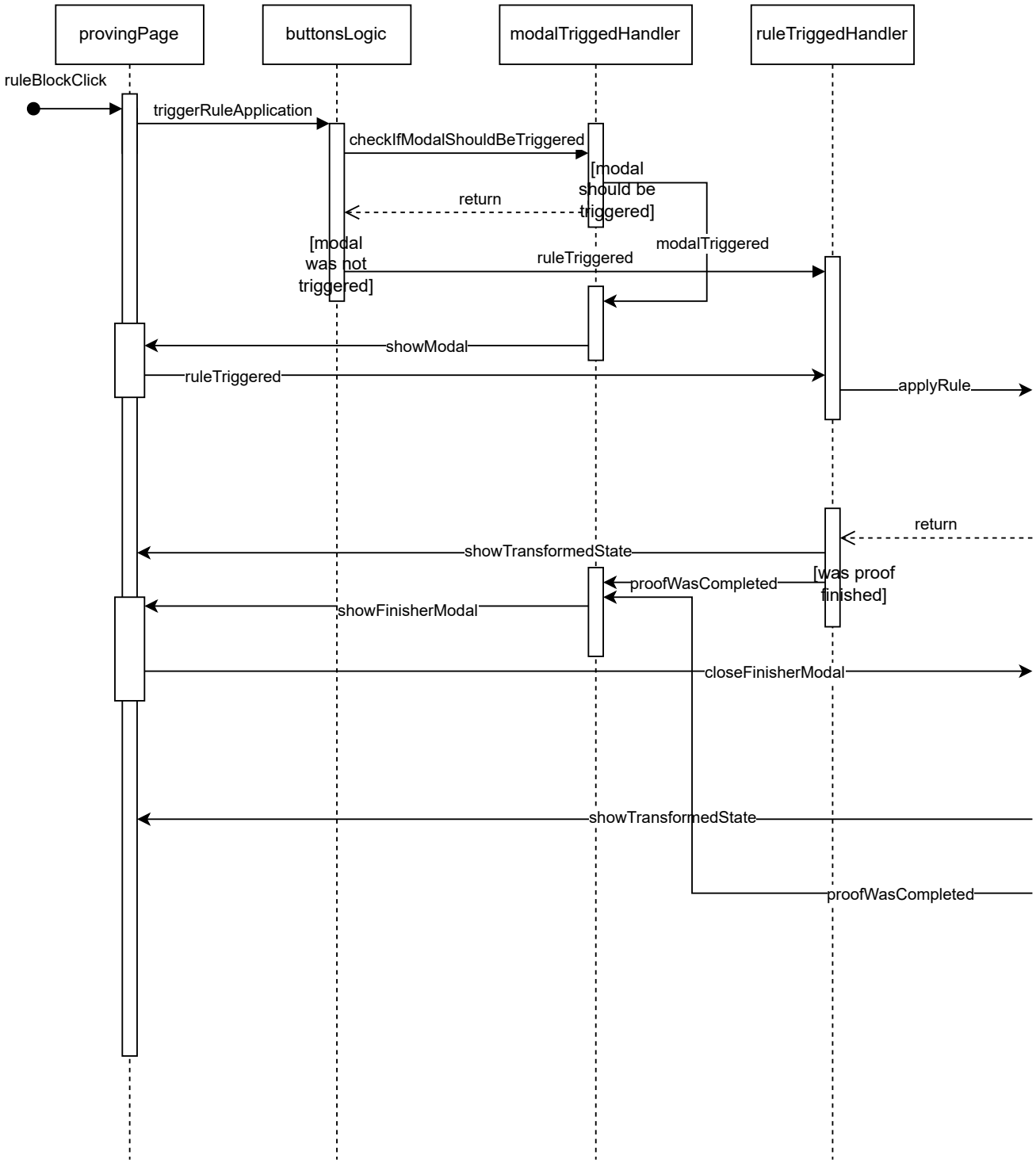
Obrázek B.21 Lineární vizualizace důkazu, dokazování lemmatu

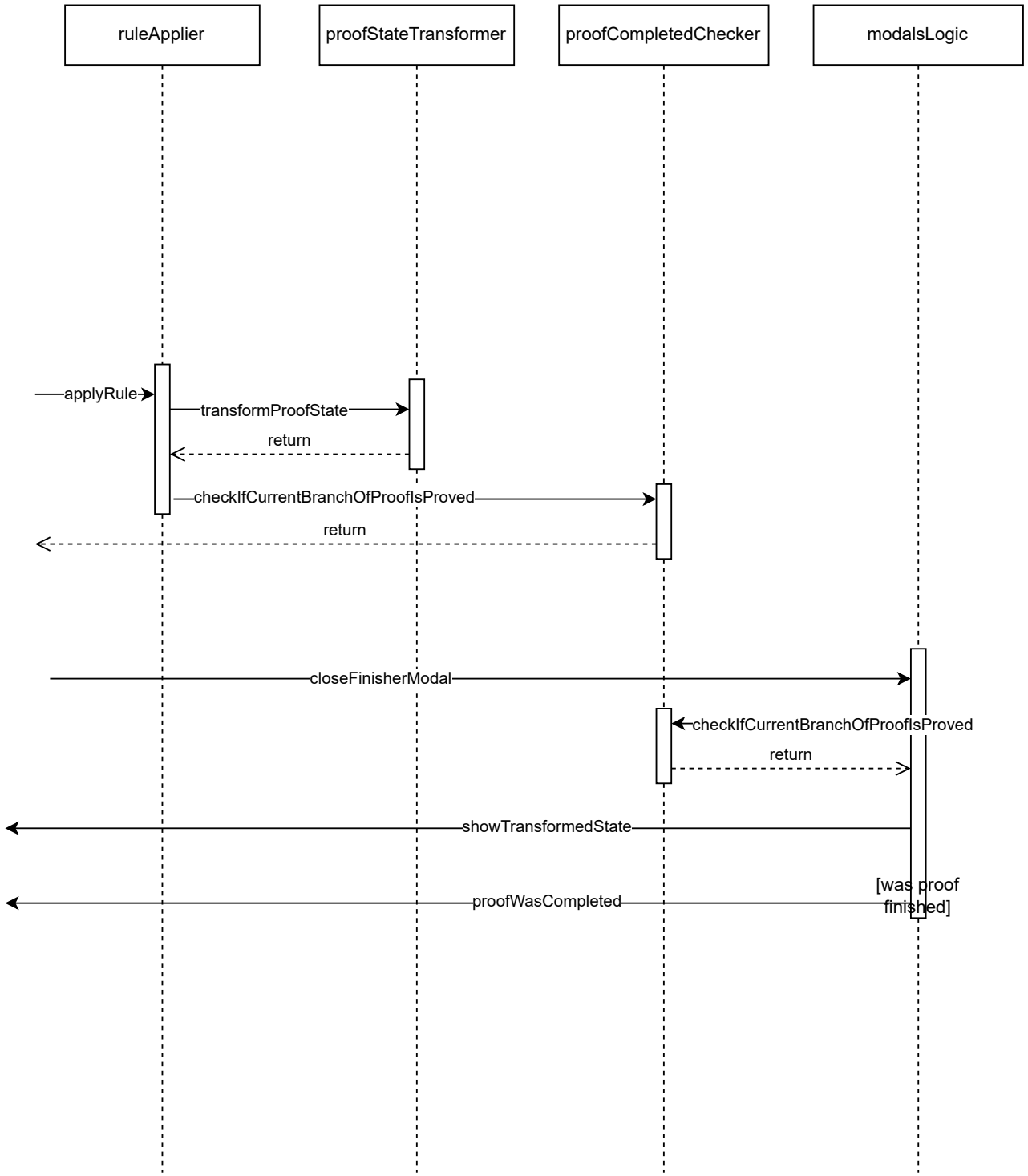
Obrázek B.22 Lineární vizualizace důkazu, skryté lemma

Příloha C

Sekvenční diagram aplikace pravidla

Tato příloha obsahuje zjednodušený sekvenční diagram, který zobrazuje komunikaci mezi jednotlivými funkcemi po kliknutí na blok pravidla. Pro větší přehlednost diagramu jsou vynechány volání obsluhující selhání aplikace pravidla. Navíc je vynechána komunikace mezi funkcemi, které zajišťují transformaci stavu. Tato transformace kvůli překladu z textových definic je delegována mezi množství objektů, což by diagram neúnosně zvětšilo. Volání, které proběhnou pouze při splnění určité podmínky, jsou označeny hranatými závorkami.


provingPage
buttonsLogic
modalTriggedHandler
ruleTriggedHandler
ruleBlockClick
triggerRuleApplication
checkIfModalShouldBeTriggered
[modal should be triggered]
return
[modal was not triggered]
modalTriggered
ruleTriggered
showModal
ruleTriggered
applyRule
return
showTransformedState
[was proof finished]
proofWasCompleted
showFinisherModal
closeFinisherModal
showTransformedState
proofWasCompleted


ruleApplier
proofStateTransformer
proofCompletedChecker
modalsLogic
applyRule
transformProofState
return
checkIfCurrentBranchOfProofIsProved
return
closeFinisherModal
checkIfCurrentBranchOfProofIsProved
return
showTransformedState
[was proof finished]
proofWasCompleted

Bibliografie

1. HUTH, Michael; RYAN, Mark. Introduction. In: *LOGIC IN COMPUTER SCIENCE: Modelling and Reasoning about Systems*. Second. New York, USA: Cambridge University Press, 2004. ISBN 9780521543101.
2. BUCHBERGER, Bruno. Propositional logic semantics. In: *skriptum Johannes Kepler Univerzity v Linci*. 1991.
3. MANCOSU, Palo; GALVAN, Sergio; ZACH, Richard. In: *An introduction to proof theory: Normalization, Cut-Elimination, Consistency Proofs*. First. New York, USA: Oxford University Press, 2021. ISBN 9780191938795.
4. DEMLOVA, Marie. *Predicate Logic*. FEL ČVUT, 2017. Dostupné také z: `https://math.fel.cvut.cz/en/people/gromadan/De02.pdf`. [cit. 2023-12-04].
5. DETLOVS, Vilnis; PODNEIKS, Karlis. Classical Predicate Logic Gödel's Completeness Theorem. In: *Introduction to Mathematical Logic*. 2021. vyd. Riga, Latvia: University of Latvia, 2021, s. 143–163.
6. RATSCHAN, Stefan. *Formal Methods and Specification (SS 2022/23), Lecture 2: Proofs*. 2022. Dostupné také z: `https://courses.fit.cvut.cz/NI-FME/lectures/files/lecture_2_proofs_en.pdf`. [cit. 2023-12-04].
7. *Isabelle Overview*. Munich mirror, 2021. Dostupné také z: `https://isabelle.in.tum.de/overview.html`. [cit. 2023-12-04].
8. NIPKOW, Tobias. *Programming and Proving in Isabelle/HOL*. Technische Universität München, 2023. Dostupné také z: `https://isabelle.in.tum.de/dist/Isabelle2023/doc/prog-prove.pdf`. [cit. 2023-12-10].
9. *Coq Welcome*. Coq Team, 2023. Dostupné také z: `https://coq.inria.fr/`. [cit. 2023-12-10].
10. *Coq Docs: Core language*. Inria, CNRS a contributors, V8.18.0, 2021. Dostupné také z: `https://coq.inria.fr/doc/V8.18.0/refman/language/core/index.html`. [cit. 2023-12-10].
11. *A short introduction to Coq*. Coq Team, 2023. Dostupné také z: `https://coq.inria.fr/a-short-introduction-to-coq`. [cit. 2023-12-10].
12. BREITNER, Joachim. *About The Incredible Proof Machine*. 2015. Dostupné také z: `https://www.joachim-breitner.de/blog/682-The_Incredible_Proof_Machine`. [cit. 2023-12-11].
13. BREITNER, Joachim. Visual Theorem Proving with the Incredible Proof Machine. In: 2016, s. 123–139. ISBN 978-3-319-43143-7. Dostupné z DOI: `10.1007/978-3-319-43144-4_8`.

14. BREITNER, Joachim. *The Incredible Proof Machine*. Dostupné také z: https://incredible.pm/. [cit. 2023-12-11].
15. YANG, Edward Z. *Logitext application*. Dostupné také z: http://logitext.mit.edu/main. [cit. 2023-12-11].
16. YANG, Edward Z. *Logitext tutorial*. 2012. Dostupné také z: http://logitext.mit.edu/tutorial. [cit. 2023-12-11].
17. BENKMANN, Matthias S. *Visualization of Natural Deduction as a Game of Dominoes*. Dostupné také z: http://www.winterdrache.de/freeware/domino/data/article.html. [cit. 2023-12-11].
18. BENKMANN, Matthias S. *Domino On Acid - Natural Deduction Visualized As A Game Of Dominoes*. Dostupné také z: http://www.winterdrache.de/freeware/domino/index.html. [cit. 2023-12-11].
19. LERNER, Sorin; FOSTER, Stephen P.; GRISWOLD, William G. Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors. *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 2015. Dostupné také z: https://api.semanticscholar.org/CorpusID:6987614. [cit. 2023-12-12].
20. LERNER, Sorin; FOSTER, Stephen P.; GRISWOLD, William G. *Polymorphic Blocks: Proof game*. Dostupné také z: https://cseweb.ucsd.edu/~lerner/proof-game/. [cit. 2023-12-12].
21. AURUM, Aybüke; WOHLIN, Claes. *Engineering and Managing Software Requirements*. Springer Berlin, Heidelberg, 2006. ISBN 978-3-540-28244-0.
22. CHUNG, Lawrence; LEITE, Julio. On Non-Functional Requirements in Software Engineering. In: 2009, s. 363–379. ISBN 978-3-642-02462-7. Dostupné z DOI: 10.1007/978-3-642-02463-4_19.
23. BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. *The Unified Modeling Language User Guide*. Addison Wesley, 1999. ISBN 0-201-57168-4.
24. BERNERS-LEE, Tim; FISCHETTI, Mark. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. HarperBusiness, 2000. ISBN 0-06-251587-X.
25. *Memorandum of Understanding Between W3C and WHATWG* [online]. W3C. Dostupné také z: https://www.w3.org/2019/04/WHATWG-W3C-MOU.html. [cit. 2023-10-14].
26. LIE, Håkon Wium. *Cascading Style Sheets*. Oslo, Norway: University of Oslo, 2005. ISSN 1501-7710.
27. *CSS Snapshot 2023* [online]. W3G, 2023. Dostupné také z: https://www.w3.org/TR/css-2023/. [cit. 2023-10-14].
28. SUHRING, Steve. *JavaScript: Krok za krokem*. First. Computer Press, 2008. ISBN 978-80-251-2241-9.
29. *ECMAScript® 2023 Language Specification* [online]. ECMA International, 14th edition, 2023. Dostupné také z: https://262.ecma-international.org/14.0/. [cit. 2023-10-14].
30. HAVERBEKE, Marijn. *Eloquent JavaScript: A Modern Introduction to Programming*. First. No Starch Press, 2011. ISBN 9781593272821. Dostupné také z: https://books.google.cz/books?id=9U5I_tskq9MC.
31. BIERMAN, Gavin; ABADI, Martin; TORGERSEN, Mads. Understanding TypeScript. In: *European Conference on Object-Oriented Programming*. 2014. Dostupné také z: https://api.semanticscholar.org/CorpusID:45852881.
32. BAMPAKOS, Aristeidis; DEELEMAN, Pablo. *Learning Angular: A no-nonsense guide to building web applications with Angular*. Fourth. Birmingham, UK: Packt Publishing Ltd., 2023. ISBN 978-1-80324-060-2.

33. *Introduction* [online]. Google. Dostupné také z: https://docs.angularjs.org/guide/introduction. [cit. 2023-10-15].

34. *Data Binding* [online]. Google, master-snapshot. Dostupné také z: https://docs.angularjs.org/guide/databinding. [cit. 2023-10-15].

35. *Dependency Injection* [online]. Google, master-snapshot. Dostupné také z: https://docs.angularjs.org/guide/di. [cit. 2023-10-15].

36. *Creating Custom Directives* [online]. Google, master-snapshot. Dostupné také z: https://docs.angularjs.org/guide/directive. [cit. 2023-10-15].

37. GACKENHEIMER, Cory. *Introduction to React*. New York, USA: Apress Media, 2015. ISBN 978-1-4842-1245-5.

38. *Describing the UI* [online]. Meta Open Source. Dostupné také z: https://react.dev/learn/describing-the-ui. [cit. 2023-10-15].

39. *Introducing Hooks* [online]. Meta Platforms, v18.2.0. Dostupné také z: https://legacy.reactjs.org/docs/hooks-intro.html. [cit. 2023-10-15].

40. FILIPOVA, Olga. *Learning Vue.js 2*. Birmingham, UK: Packt Publishing, 2016. ISBN 978-1-78646-994-6.

41. *Introduction* [online]. VueJS community, v3. Dostupné také z: https://vuejs.org/guide/introduction.html. [cit. 2023-10-16].

42. *Reactivity in Depth* [online]. VueJS community, v3. Dostupné také z: https://vuejs.org/guide/extras/reactivity-in-depth.html. [cit. 2023-10-16].

43. *Rendering Mechanism* [online]. VueJS community, v3. Dostupné také z: https://vuejs.org/guide/extras/rendering-mechanism.html. [cit. 2023-10-16].

44. HAMM, Matthew J. *Wireframing Essentials: An introduction to user experience design*. Birmingham, UK: Packt Publishing, 2014. ISBN 978-1-84969-854-2.

Obsah přiloženého média

- `readme.txt` — stručný popis obsahu média
- `proofAssistantApp`
  - `src` — zdrojové kódy implementace
  - `out` — dokumentace
  - `build` — build aplikace
  - `README.md` — pokyny k implementaci
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF
  - `thesis` — zdrojová forma práce ve formátu LaTeX