Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Počítačové dokazování

2021

Vedoucí práce: doc. RNDr. Miroslav Kolařík, Ph.D.

Bc. Filip Tůma

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Bc. Filip Tůma
Název práce:	Počítačové dokazování
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	doc. RNDr. Miroslav Kolařík, Ph.D.
Počet stran:	64
Přílohy:	1 CD
Jazyk práce:	český

Bibliographic info

Author:	Bc. Filip Tůma
Title:	Computer proving
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Computer Science, full-time form
Supervisor:	doc. RNDr. Miroslav Kolařík, Ph.D.
Page count:	64
Supplements:	1 CD
Thesis language:	Czech

Anotace

Práce je zaměřena na počítačové dokazování. Nejprve je čtenář seznámen s úvodními pojmy, poté je představeno několik problémů, které byly dokázány s pomocí počítače. Další část je zaměřena na popis jednotlivých významných dokazovacích systémů, včetně jejich využití v praxi. Na závěr je popsán jednoduchý dokazovací systém, vytvořený jako součást této práce.

Synopsis

This thesis is focused on proofs done by computers. First, the reader is acquainted with the introductory concepts, then a few problems proved with the help of computers are introduced. Next part contains description of individual important prooving systems including their use. Finally, a simple proof system created as a part of this work is described.

Klíčová slova: dokazovač, rezoluce, logika

Keywords: prover, resolution, logic

Rád bych tímto poděkoval vedoucímu své diplomové práce doc. RNDr. Miroslavovi Kolaříkovi, Ph.D., za jeho vedení při tvorbě této práce.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

1 Úvod 7

2 Úvodní pojmy a výběr dedukčního kalkulu 9
- 2.1 Teorie typů . . . 9
- 2.2 Curry-Howardův isomorfismus . . . 9
- 2.3 Výběr dedukčního kalkulu . . . 10
  - 2.3.1 Rezoluční metoda . . . 10
  - 2.3.2 Metoda propojení matice . . . 11

3 Problém čtyř barev 12
- 3.1 Kempeho řetězec . . . 13
- 3.2 Kempeho „důkaz“ . . . 15
  - 3.2.1 X má čtyři sousedy . . . 15
  - 3.2.2 X má pět sousedů . . . 16

4 Další příklady problémů dokázaných pomocí počítače 18
- 4.1 Robbinsova domněnka (Robbins Conjecture) . . . 18
- 4.2 Keplerova domněnka (Kepler Conjecture) . . . 18
- 4.3 Hra Connect four . . . 19

5 Základní dokazovací systémy 21
- 5.1 Automath a následníci . . . 21
  - 5.1.1 NuPRL . . . 21
  - 5.1.2 Coq . . . 22
  - 5.1.3 Twelf . . . 22
  - 5.1.4 Agda . . . 23
- 5.2 LCF a následníci . . . 24
  - 5.2.1 HOL . . . 25
  - 5.2.2 Isabelle . . . 26
- 5.3 Mizar . . . 27
  - 5.3.1 Mizar Mathematical Library – MML . . . 29
- 5.4 Prover9 & Mace4 . . . 29
- 5.5 E theorem prover . . . 30
- 5.6 Vampire . . . 30
- 5.7 ACL2 . . . 31
- 5.8 PVS – Prototype Verification System . . . 32
- 5.9 Porovnání jednotlivých systémů . . . 33

6 Výhled do budoucna a využití ATP 35
- 6.1 Výhled do budoucna . . . 35
  - 6.1.1 QED . . . 35
  - 6.1.2 MathWiki . . . 35
- 6.2 Využití ATP . . . 36

6.2.1 Formalizace matematiky 36
6.2.2 Programová verifikace 37

7 Dokazovač – manuál **38**

8 Dokazovač – popis implementace **41**
8.1 Třída Variable 41
8.2 Třída Operator 42
8.3 Třída Formule 43
8.4 Třída Proof 48
8.5 Třída TableProof 51
8.6 Třída Window 51
8.7 Třída HelpWindow 53

Závěr **54**

Conclusions **55**

A Predikátová a výroková logika **56**
A.1 Predikátová logika 56
A.2 Výroková logika 58

B Obsah přiloženého CD **60**

Seznam zkratek **61**

Bibliografie **62**

Seznam obrázků

1 Obarvení anglických hrabství [7] 12
2 Převod mapy na graf [8] 13
3 Kempeho řetězec – oblasti 14
4 Kempeho řetězec – vrcholy [9] 14
5 Kempeho důkaz 1 15
6 Kempeho důkaz 2 15
7 Kempeho důkaz 3 16
8 Kempeho důkaz 4 16
9 Heawoodův protipříklad 17
10 Uspořádání koulí [12] 19
11 Hra Connect four [13] 20
12 Definice násobení a sčítání v Agdě [22] 24
13 Logické spojky a kvantifikátory v jazyce Mizar [28] 27
14 Základní vyhrazené výrazy v jazyce Mizar [29] 28
15 Porovnání jednotlivých dokazovačů 1 [38] 33
16 Porovnání jednotlivých dokazovačů 2 [38] 33
17 Hlavní okno aplikace 38
18 Příklad důkazu rezoluční metodou 40
19 Příklad důkazu tabulkovou metodou 40
20 Třída Operator a její podtřídy 42

Seznam tabulek

1 Curry-Howardova korespondence 10
2 Tabulka chromatických čísel 17
3 Formalizace sta nejzajímavějších vět/důkazů [43] 36

Seznam zdrojových kódů

1 Metoda parse třídy Variable 42
2 Třída Implication 43
3 Metoda makeList – negace 45
4 Metoda removeEquivalences 46
5 Metoda convertToCNF 47
6 Metoda evalForAll 48
7 Metoda resolve_clauses 49
8 Metoda resolution 50
9 Metoda resolut_run – vykreslení důkazu 51
10 Metoda table_run 52
11 Metoda short_version 53

1 Úvod

Automatické uvažování je oblast počítačové vědy považovaná za součást umělé inteligence, avšak do jisté míry je toto téma spojeno i s teoretickou informatikou či filozofií.

Uvažování je schopnost vyvozovat závěry z předpokladů. Automatické uvažování se tedy týká tvorby výpočetních systémů, které tento proces zautomatizují. Přestože je celkovým cílem mechanizace různých forem uvažování, byl tento pojem značně identifikován s deduktivním uvažováním, tak jak je praktikováno v matematice a formální logice. V tomto ohledu je automatické uvažování podobné tomu, když dané věty dokazujeme ručně.

Tvorba programu pro automatické uvažování znamená poskytnutí algoritmického popisu formálnímu kalkulu tak, aby jej bylo možné implementovat a účinně dokazovat věty kalkulu. Důležité aspekty tohoto zahrnují:

- definování třídy problémů, které bude program řešit,
- výběr jazyku, který se bude používat pro reprezentaci poskytnutých informací, stejně tak jako informací odvozených v programu,
- upřesnění mechanismu, který bude program používat k zjištění deduktivních závěrů,
- zjištění, jak všechny tyto výpočty provádět efektivně.

Zatímco základní výzkum pokračuje dál s cílem poskytnout teoretický rámec, toto odvětví se dostalo do bodu, kde se automatické uvažování používá k důkazům otevřených otázek (nebo alespoň k pokusům o důkazy) v matematice a logice, k poskytnutí důležitých aplikací ve výpočetní technice, řešení problémů ve strojírenství a nalezení nových přístupů k otázkám ve filozofii.

Nejvyvinutější součásti automatického uvažování jsou: automatické dokazování vět (ATP – Automated Theorem Proving), jehož součástí je interaktivní dokazování vět, které je méně automatické, ale o to více praktické, a automatická kontrola důkazů (Automated Proof Checking).

Problém v rámci programu automatického uvažování se skládá ze dvou částí:

- množiny výroků, která vyjadřuje všechny informace, které daný program má (předpoklady problému – problem assumptions),
- výroku, který vyjadřuje otázku (závěr problému – problem conclusion).

Řešení daného problému pak tedy znamená dokázání závěru z daných předpokladů. To se v různých programech děje různými způsoby (viz část o výběru dedukčního kalkulu).

Při tvorbě systému pro automatické uvažování nastávají tři důležitá rozhodnutí:

- vymezení domény problémů – doména může být velká, jak je tomu v případě obecných dokazovacích systémů, nebo může být omezena pro nějakou konkrétní třídu problémů. Typickým přístupem je nejprve dát systému dostatečnou logickou sílu (např. v podobě logiky prvního řádu) a poté se dále vymezovat směrem k doméně problémů definovanou pomocí doménových axiomů.

- reprezentace problému – toto zahrnuje jednak problém reprezentace problému pro samotný systém, tedy jak ho formalizovat do podoby srozumitelné pro dokazovací systém, jednak jak bude problém reprezentován interně v rámci programu a v neposlední řadě jak bude výsledek problému reprezentován zpět uživateli. Existuje několik různých formalismů, jejichž výběr je částečně závislý na problémové doméně a na dedukčním kalkulu použitém v dokazovacím programu. Nejčastěji používané formalismy jsou logika prvního řádu (případně i vyšších řádů), typovaný lambda kalkul či klauzální logika .

- výběr dedukčního kalkulu – viz dále.

Zdroj: [**5**].

2 Úvodní pojmy a výběr dedukčního kalkulu

Předpokládá se, že je čtenář obeznámen se základy výrokové a predikátové logiky. V opačném případě nalezne výklad potřebných pojmů v příloze A.

2.1 Teorie typů

Jedná se o teorii na pomezí algoritmiky, matematiky a filozofie. Teorii typů poprvé vyložil Bertrand Russell. Účelem teorie typů bylo vyhnutí se paradoxům v teorii množin.

V systému teorie typů je pojem term protějškem typu. Například 4, 2 + 2, 2 * 2 jsou všechno různé termy s typem *nat* pro přirozená čísla. Obvykle je term následován dvojtečkou a daným typem, tzn. $2 : nat$, což znamená, že číslo 2 je typu *nat*. O typech lze v obecnosti říci málo. Často se interpretují jako druh kolekce (ne nutně množiny) hodnot, na které by se měl term vyhodnotit. Tvorba termů a typů závisí na konkrétních teoriích.

Rozdíly oproti teorii množin:

- Teorie množin je postavena na základě logiky. Vyžaduje pod sebou oddělený systém jako například predikátovou logiku. V teorii typů lze koncepty jako „or“ a „and“ zakódovat v samotné teorii.
- V teorii množin není prvek omezen na jednu množinu. V teorii typů termy náleží pouze jednomu typu.
- Teorie množin kóduje čísla jako množiny (0 je $\emptyset$, 1 je $\{\emptyset\}$, 2 je $\{\emptyset, \{\emptyset\}\}$, atd.). V teorii typů lze čísla kódovat jako funkce pomocí Churchova kódování[1] nebo více přirozeně jako induktivní typy.

Zdroj: [**3**].

2.2 Curry-Howardův isomorfismus

Curry-Howardův isomorfismus (korespondence) je přímý vztah mezi počítačovými programy a matematickými důkazy. Jedná se o zobecnění syntaktické analogie mezi systémy formální logiky a výpočetních kalkulů. Poprvé byl objeven americkým matematikem Haskellem Currym a logikem Williamam A. Howardem.

Ve své obecnější formě je Curry-Howardova korespondence vztah mezi formálními důkazy a typovými systémy pro modely výpočtu. Rozděluje se do dvou částí. Jedna na úrovni formulí a typů, která je nezávislá na konkrétním důkazním systému nebo modelu výpočtu a druhá na úrovni důkazů a programů, která je specifická pro konkrétní výběr důkazního systému a modelu výpočtu.

Zdroj: [**4**].

[1] kódování pro reprezentaci dat a operátorů v lambda kalkulu

Logická strana	Programovací strana
implikace	funkční typ
konjunkce	součinový typ
disjunkce	součtový typ
pravdivá formule	jednotkový typ
nepravdivá formule	prázdný typ
univerzální kvantifikátor	zobecněný součinový typ
existenční kvantifikátor	zobecněný součtový typ

Tabulka 1: Curry-Howardova korespondence

2.3 Výběr dedukčního kalkulu

Tato část vznikla na základě [**5**].

Dedukční kalkul se skládá z množiny logických axiomů a kolekce dedukčních pravidel pro odvozování nových formulí z předchozích formulí. Vyřešení problému v rámci domény pak znamená určení pravdivosti závěru problému (α), v podobě formule, z množiny sestávající z logických a doménových axiomů a předpokladů problému (zn. Γ). To jakým způsobem toho program dosáhne závisí hlavně na dedukčním kalkulu. Některé programy mohou jít přímou cestou a tvořit důkaz krok za krokem; to jsou převážně systémy založené na přirozené dedukci. Jiné programy jdou cestou nepřímou, tedy ukážou, že množina $\Gamma \cup \{\neg\alpha\}$ je nekonzistentní. Toho je docíleno odvozením kontradikce z této množiny. Takovéto programy pak využívají metody jako sekvenční dedukce, metoda propojení matice či rezoluce.

V následujících částech si představíme dvě z autorova pohledu zajímavé metody.

2.3.1 Rezoluční metoda

Tato metoda je v současné době asi nejpopulárnější. Rezoluční pravidlo je modelováno na základě pravidla zřetězení a v zásadě říká, že z $p \vee q$ a $\neg q \vee r$ odvodíme $p \vee r$. V obecnější podobě bychom mohli napsat, že pro dvě klauzule C_1 a C_2 takové, že jedna obsahuje literál l v pozitivní a druhá v negativní formě platí:

$$\frac{C_1, \quad C_2}{(C_1 - l) \vee (C_2 - \neg l)},$$

kde $(C_1 - l)$ znamená klauzule C kromě literálu l.

Herbrandova věta (1930) zajišťuje, že nesplnitelnost kolekce klauzulí lze zjistit pomocí základní rezoluční metody. Avšak přímá implementace této metody pomocí Herbrandovy věty vede ke generování velkého množství základních termů, a proto je neefektivní. Vyšší efektivitu zajistilo využití unifikace a také metoda binární rezoluce (obdobná jako základní rezoluční metoda, ale využívá právě unifikaci).

2.3.2 Metoda propojení matice

Název je odvozen od fungování metody. Daná formule se nejprve převede do konjunktivní normální formy (CNF), poté je vytvořena matice tak, že jednotlivé řádky reprezentují klauzule, v rámci řádků je pak klauzule rozdělena na literály. Metoda pak funguje tak, že postupně vyzkouší všechny průchody maticí shora dolů tak, že z každého řádku se vezme jeden literál. V případě, že daná cesta maticí obsahuje jak negativní, tak pozitivní výskyt konkrétního literálu, tak už se nemusí dále procházet (neumožní splnění původní formule). Výsledkem metody je, že se buď najde cesta, která neobsahuje dva komplementární literály, tudíž tato cesta je řešením formule, v opačném případě je původní formule nesplnitelná.

Pro lepší představu si uveďme příklad: Zadané axiomy jsou $(A \vee B)$, $(\neg A \vee C)$, $(\neg B \vee D)$, $(C \vee \neg D)$ a máme ukázat, že z nich plyne závěr C. Pro dané zadání získáme formuli $(A \vee B) \wedge (\neg A \vee C) \wedge (\neg B \vee D) \wedge (C \vee \neg D) \wedge \neg C$ v konjunktivní normální formě (formule vznikla jako logický součin axiomů společně s negací závěru problému) a následující matici (tabulku):

A	B
¬A	C
¬B	D
C	¬D
¬C	

Z tabulky dostaneme těchto šestnáct cest:

1	A,¬A, ¬B, C, ¬C
2	A, ¬A, ¬B, ¬D,¬C
3	A, ¬A, D, C, ¬C
4	A, ¬A, D, ¬D, ¬C
5	A, C, ¬B, C, ¬C
6	A, C, D, C, ¬C
7	A, C, ¬B, ¬D, ¬C
8	A, C, D, ¬D, ¬C
9	B,¬A, ¬B, C, ¬C
10	B, ¬A, ¬B, ¬D,¬C
11	B, ¬A, D, C, ¬C
12	B, ¬A, D, ¬D, ¬C
13	B, C, ¬B, C, ¬C
14	B, C, D, C, ¬C
15	B, C, ¬B, ¬D, ¬C
16	B, C, D, ¬D, ¬C

Z toho plyne, že žádná z cest není řešením formule (každá cesta obsahuje

minimálně jeden literál v obou podobách). Tím pádem je formule nesplnitelná, což implikuje, že zadaný závěr plyne z axiomů.

3 Problém čtyř barev

Před tím, než se dostaneme k samotným dokazovacím systémům, bych rád přiblížil problém čtyř barev, který je považován za vůbec první problém dokázaný pomocí počítače.

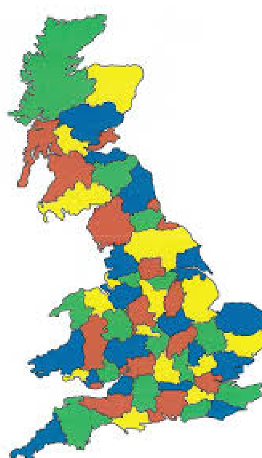
Text této části vznikl na základě [**6**] a [**7**].

Již v roce 1840 formuloval August Ferdinand Möbius podobný problém, problém pěti princů:

V Indii byl král, který měl velké království a pět synů. Ve své poslední vůli král řekl, že po jeho smrti by synové měli mezi sebou rozdělit království tak, aby oblast patřící každému synovi měla společnou hranici (nejen bod) se zbývajícími čtyřmi oblastmi. Oblasti by navíc měly být souvislé. Jak by mělo být království rozděleno?

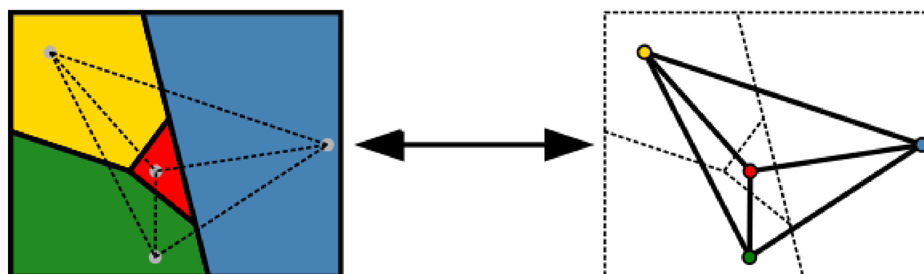
Intuitivně lze vidět, že tento problém nemá řešení.

Problém čtyř barev je jedním z nejslavnějších matematických problémů. Jedna z prvních zmínek o tomto problému se datuje do roku 1852, kdy doktorand University College v Londýně, Francis Guthrie, zaslal tuto úlohu svému bratrovi Frederickovi. Ten obarvil mapu anglických hrabství čtyřmi různými barvami, avšak žádná dvě sousední hrabství nesdílela stejnou barvu. Otázka, kterou si položil byla nasnadě: platí to pro všechny mapy v rovině? Frederick Guthrie žádal o pomoc Augusta De Morgana. Po dlouhou dobu se však nedařilo daný problém dokázat (vyřešit).



Obrázek 1: Obarvení anglických hrabství [**7**]

Z pohledu teorie grafů je tento problém formálně chápan tak, že pro zadaný planární graf bez smyček je chromatické číslo jeho duálního grafu menší nebo rovno čtyřem.

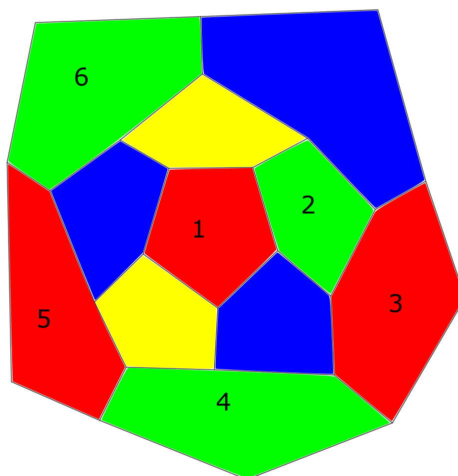


Obrázek 2: Převod mapy na graf [**8**]

V roce 1879 publikoval první „důkaz“ tohoto problému Artur Kempe. V roce 1880 pak přišel další pokus o důkaz od Petera G. Taita. Ačkoliv byly oba důkazy později vyvráceny (Kempeho důkaz v roce 1890 Percy Heawoodem), přinesly zajímavé poznatky. Tait přišel s ekvivalentním vyjádřením problému čtyř barev ve smyslu 3-obarvení hran. Kempe přišel s něčím, čemu se říká Kempeho řetězec.

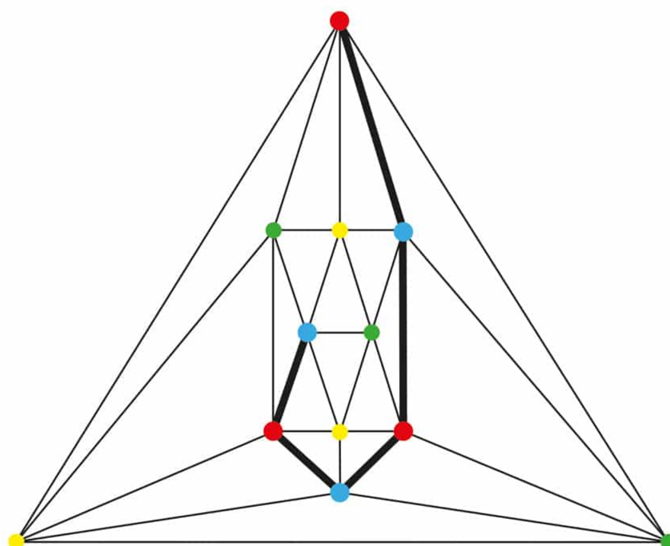
3.1 Kempeho řetězec

Kempeho řetězec obsahující barvy a a b je maximální propojená množina oblastí obarvených jednou z těchto dvou barev. Propojená ve smyslu, že z každé oblasti lze cestovat do kterékoliv jiné oblasti v množině, aniž bychom tuto oblast opustili. Maximální ve smyslu: neexistuje další oblast daných dvou barev, o kterou bychom mohli množinu zvětšit tak, aby byla stále propojená.



Obrázek 3: Kempeho řetězec – oblasti

V novodobější definici problému čtyř barev se oblasti nahrazují uzly duálního grafu. Pak lze duálně formulovat i pojem Kempeho řetězce: Kempeho řetězec obsahující barvy a a b je maximální spojitý podgraf obsahující vrcholy pouze zadaných barev.



Obrázek 4: Kempeho řetězec – vrcholy [**9**]

3.2 Kempeho „důkaz“

Popis důkazu, včetně obrázků, vznikl na základě [**10**].

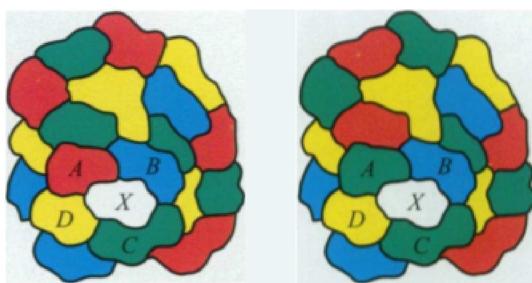
Základní předpoklad je, že mapu, která obsahuje čtyři a méně oblasti, lze obarvit triviálně. Dále předpokládejme, že mapu obsahující n oblastí lze obarvit čtyřmi barvami. Označme si jako M mapu, která má celkem $n+1$ oblastí. M nutně obsahuje oblast, která sousedí s pěti nebo méně oblastmi. Tuto oblast označme jako X. Nyní nastávají dva případy, které Kempeho zajímaly:

- X sousedí s přesně čtyřmi oblastmi,
- X sousedí s přesně pěti oblastmi.

Ostatní případy, tedy kdy X sousedí s jednou až třemi oblastmi, lze vyřešit triviálně, protože máme pořád dostatek barev na obarvení X. Pro důkaz zbylých dvou případů byl použit právě Kempeho řetězec. Myšlenka byla postavena na tom, že v rámci tohoto řetězce, lze barvy prohodit a mapa bude stále správně obarvena.

3.2.1 X má čtyři sousedy

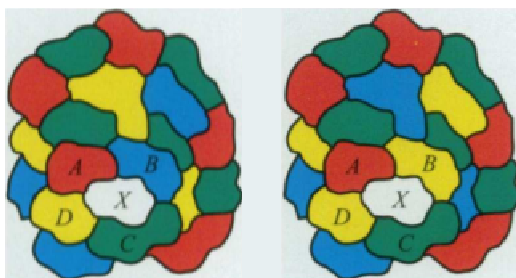
- případ, kdy oblasti A a C nejsou ve stejném červeno-zeleném řetězci:



Obrázek 5: Kempeho důkaz 1

  V této situaci lze přebarvit jeden řetězec tak, že A a C budou mít stejné barvy, viz obrázek.

- případ, kdy oblasti A a C jsou ve stejném červeno-zeleném řetězci:

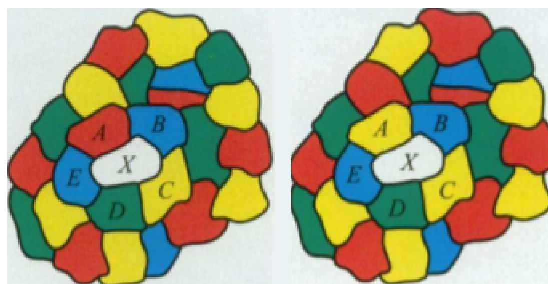


Obrázek 6: Kempeho důkaz 2

V této situaci si nepomůžeme přebarvením oblasti obsahující A a C. Fakt, že A a C jsou ve stejné červeno-zelené oblasti však zaručuje to, že B a D jsou v různých žluto-modrých oblastech a můžeme tedy použít výsledek prvního případu.

3.2.2 *X* má pět sousedů

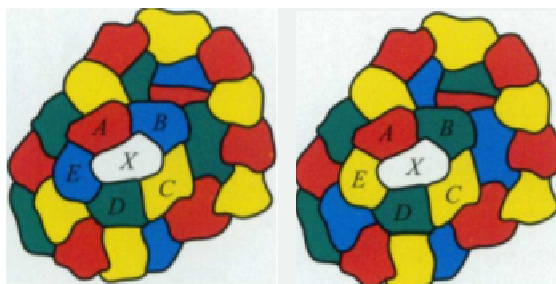
- případ, kdy jsou buď oblasti A a C v různých červeno-žlutých řetězcích, nebo A a D v různých červeno-zelených řetězcích



Obrázek 7: Kempeho důkaz 3

V této oblasti můžeme použít podobné přebarvení jako u případu čtyř sousedů.

- případ, kdy oblasti A a C jsou ve stejném červeno-žlutém řetězci a A a D jsou ve stejném červeno-zeleném řetězci

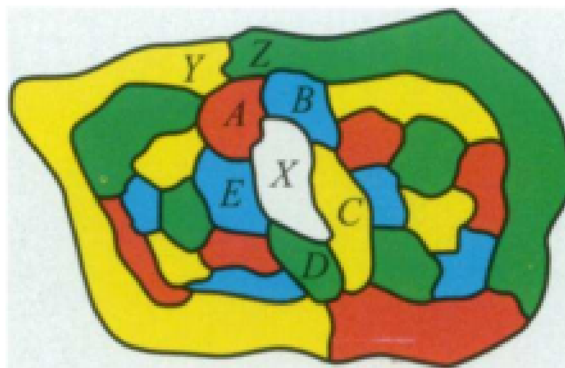


Obrázek 8: Kempeho důkaz 4

Tato situace je nejsložitější (a právě v této situaci udělal Kempe chybu v důkazu). Kempe tuto situaci vyřešil tak, že se přebarví modro-zelený řetězec obsahující B a modro-žlutý řetězec obsahující E.

V této poslední situaci však Kempe nepočítal s možností, že se tyto dva řetězce budou dotýkat.

V roce 1890 Percy Heawood zjistil, že Kempeho metoda nefunguje pro všechny mapy.



Obrázek 9: Heawoodův protipříklad

Podařilo se mu jeho metodu však pozměnit tak, že se mu povedlo dokázat problém pěti barev. Heawood také ukázal, že k obarvení jakékoliv mapy na anuloidu stačí sedm barev. Dále vyslovil domněnku:

Dolní mez pro počet barev potřebných k obarvení mapy na zadané ploše je:

$$X(g) = \left\lfloor \frac{1}{2}(7 + \sqrt{48g+1}) \right\rfloor,$$

kde g je genus[1] *plochy.*

p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
χS_p	4	7	8	9	10	11	12	12	13	13	14	15	15	16	16	16

Tabulka 2: Tabulka chromatických čísel

Poznamenejme, že χS_p udává chromatické číslo (neformálně počet barev potřebných k obarvení) kulové plochy s p uchy.

Během 60. let dvacátého století objevil německý matematik Heinrich Heesch metody k nalezení důkazu pomocí počítače. Dále také rozvinul koncept redukovatelnosti mapy na základní nevyhnutelné konfigurace a odhadl, že je potřeba se umět vypořádat s 10 000 takovými konfiguracemi. V roce 1970 nalezl Wolfgang Haken vylepšení Heeshovy metody k nalezení odstranitelných konfigurací. V roce 1976 přišel první validní důkaz věty o čtyřech barvách, který vydal Haken ve spolupráci s Kennethem Appelem. Důkaz je postaven na dvou technických konceptech: nevyhnutelná množina konfigurací a redukovatelné konfiguraci. Haken

[1] Genus orientované plochy je číslo udávající počet děr v ploše. Tzn. pro kouli a rovinu je genus roven nule, pro anuloid jedné, atd.

s Appelem se v důkazu zvládli vypořádat se všemi konfiguracemi (v té době jich bylo 1936). Řešení problému si vyžádalo přes tisíc hodin strojového času a test odstranitelnosti používal 487 pravidel. Další matematici nalezli menší nevyhnutelné množiny a došlo k vylepšení testu odstranitelnosti, avšak velikost množiny nevyhnutelnosti je stále příliš velká na to, aby mohl být důkaz ověřen člověkem. Tento problém se tak stal první velkou větou dokázanou pomocí počítače.

4 Další příklady problémů dokázaných pomocí počítače

4.1 Robbinsova domněnka (Robbins Conjecture)

V roce 1933 navrhl Edward Huntington novou sadu axiomů pro boolovskou algebru sestávající z těchto axiomů:

$$a \vee (b \vee c) = (a \vee b) \vee c$$

$$a \vee b = b \vee a$$

$$\neg(\neg a \vee b) \vee \neg(\neg a \vee \neg b) = a$$

Nedlouho poté přišel Herbert Robbins s domněnkou, že třetí rovnice lze nahradit za

$$\neg(\neg(a \vee b) \vee \neg(a \vee \neg b)) = a.$$

Robbins a Huntington nemohli nalézt důkaz. Algebry splňující komutativitu, asociativitu a Robbinsovu rovnici se začali nazývat Robbinsovy algebry. Je jasné, že každá booleovská algebra je Robbinsova algebra, takže zajímavým problémem bylo, zda je každá Robbinsova algebra booleovská. Jinými slovy, lze Huntingtonovu rovnici odvodit z komutativity, asociativity a Robbinsovy rovnice?

Důkaz Robbinsova problému byl nalezen 10. října 1996 systémem EQP, který je velmi podobný systému Otter (viz níže), hlavním rozdílem je, že EQP je omezený na rovnicovou logiku (equational logic). Úspěšné vyhledávání trvalo 8 dní na procesoru RS/6000 a využilo přibližně 30 megabajtů paměti.

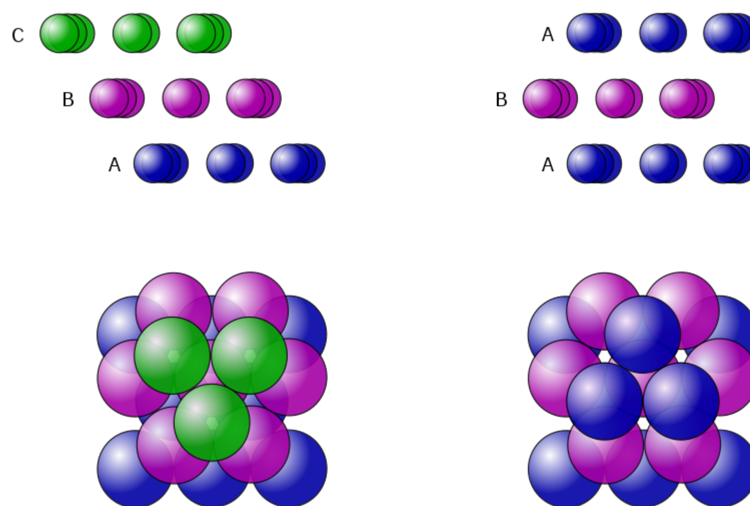
Zdroj: [**11**].

4.2 Keplerova domněnka (Kepler Conjecture)

Tato domněnka je pojmenována po matematikovi a astronomovi Johannesovi Keplerovi.

Představte si, že naplníte velkou nádobu malými kuličkami stejné velikosti. Hustota uspořádání se rovná souhrnnému objemu koulí dělenému objemem nádoby. Maximalizace počtu koulí v kontejneru znamená vytvoření uspořádání s nejvyšší možnou hustotou. Dle experimentů, pomocí náhodného házení kuliček do nádoby se dosáhne hustoty 65%. Lze však dosáhnout vyšší hustoty pomocí pečlivého uspořádání následujícím způsobem: začne se s vrstvou v šestihranné

mřížce a každá další vrstva přijde do nejnižších bodů vrstvy předchozí. V každém kroku jsou dvě možnosti umístění (viz obrázek).



Obrázek 10: Uspořádání koulí [**12**]

Každé z těchto uspořádání má hustotu přibližně 74%. Keplerova domněnka říká, že toto je uspořádání s nejlepší možnou hustotou.

Fejes Tóth ukázal, že problém stanovení maximální hustoty všech uspořádání lze zredukovat na konečný (avšak velmi velký) počet výpočtů.

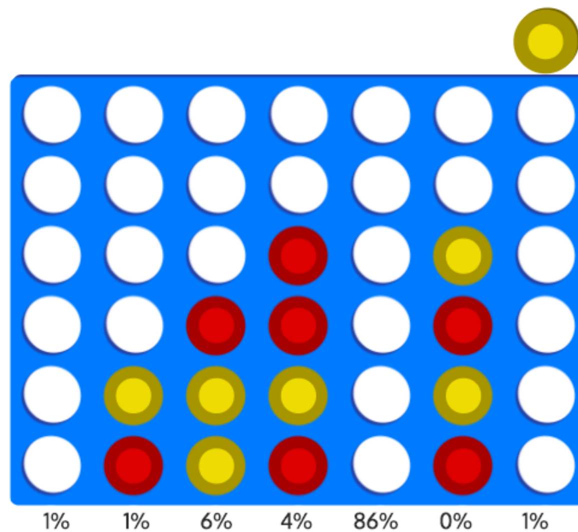
V návaznosti na přístup navržený Tóthem, Thomas Hales určil, že maximální hustotu lze zjistit minimalizací funkce se 150 proměnnými. V roce 1992 se s pomocí svého postgraduálního studenta Samuela Fergusona pustil do výzkumného programu, který by systematicky využíval metody lineárního programování k nalezení dolní hranice hodnoty této funkce pro každou ze sady více než 5000 různých konfigurací sfér. Nalezení dolních hranic pro všechny případy znamenalo řešení více než 100 000 problémů lineárního programování.

V srpnu 1998 Hales oznámil, že důkaz je kompletní. V této fázi obsahoval 250 stránek poznámek a 3GB počítačových programů, dat a výsledků.

Zdroj: [**12**].

4.3 Hra Connect four

Jedná se o hru dvou hráčů, kdy se hráči střídají v házení žetonů do mřížky o velikosti 7x6 polí. Vítězem se stává hráč, který spojí 4 svoje žetony svisle, horizontálně či diagonálně. V případě, že se žádnému hráči nepovedlo spojit čtyři (a více) žetony a pole je zcela zaplněné, jedná se o remízu.



Obrázek 11: Hra Connect four [**13**]

Z pohledu teorie her se jedná o hru s perfektní informací pro obě hrající strany, tzn. že oba hráči mají v každém okamžiku hry informace o tazích, které proběhly, a tazích, které mohou vykonat. Dále se jedná o hru s nulovým součtem, jelikož výhoda jednoho hráče je nevýhoda hráče druhého. Pro přiblížení náročnosti hry z pohledu komplexity: na klasické mřížce (7x6 polí) existuje 4 531 985 219 092 různých pozic (pro 0 – 42 žetonů ve hře).

Tato hra byla poprvé vyřešena Jamesem Dow Allenem 1. října 1988 a nezávisle na něm Victorem Allisem 16. října 1988. V době těchto prvních řešení nebyla možná analýza hrubou silou, vzhledem ke složitosti hry a dostupné počítačové technice.

Řešení této hry je, že při perfektní hře vyhrává první hráč, pokud začne hodem do prostředního sloupce. V případě, že začne v jednom ze sloupců sousedících se středovým sloupcem, jedná se o teoretickou remízu. Pokud však první hráč začne v jednom ze sloupců 1, 2, 6 nebo 7, dává tak možnost druhému hráči vynutit si výhru.

Zdroj: [**14**].

5 Základní dokazovací systémy

Celá sekce o historii počítačového dokazování vznikala na základě [**15**], [**16**]. Každá samostatná subsekce poté obsahuje další pro ni relevantní zdroje na jejím závěru.

5.1 Automath a následníci

Projekt Automath inicioval Nicolaas Govert de Bruijn v roce 1967. Jeho cílem bylo vyvinout systém pro mechanické ověřování matematiky. Souvisejícím cílem bylo vytvoření matematického jazyku, ve kterém bude celou matematiku možno vyjádřit přesně a to v tom smyslu, že lingvistická správnost implikuje matematickou správnost. Tento jazyk měl být kontrolovaný počítačem a měl být užitečný při zlepšování spolehlivosti matematických výsledků.

Automath je přímý předchůdce automatických dokazovačů založených na teorii typů. Mezi nejznámější patří NuPRL nebo Coq.

Automath byl první program, který používal Curry-Howardův isomorfismus pro kódování důkazů. Existují ve skutečnosti dva typy Curry-Howardova přístupu. Jeden, ve kterém je formule reprezentována typem, a druhý, ve kterém formule nejsou typy, ale každá formule má asociován typ důkazních objektů. Automath využívá druhý přístup. Naproti tomu Coq, NuPRL či Agda využívají přístup první.

Mezi Automathem a moderními systémy, které z něj byly odvozeny, existují tři hlavní rozdíly:

- v moderních systémech generuje uživatel důkaz interaktivně s využitím taktik jednotlivých systémů. V Automathu musí být všechny výrazy zapsány ručně. To znamená, že Automath nemá automatizaci, ale důkazy jsou explicitnější.
- Automath nemá induktivní typy. To znamená, že základní typy, které jsou v moderních systémech vytvářeny induktivně, musí být v Automathu zavedeny axiomaticky. Protože má Automath méně datových typů, je nevyhnutelné použití axiomů na začátku vývoje důkazu.
- Teorie typů, na niž je Automath založen, se nepatrně liší od „moderních" teorií typů.

Zdroje: [**15**], [**16**], [**17**].

5.1.1 NuPRL

V roce 1979 Martin-Löf představil rozšiřující verzi své teorie typů. Ve stejném roce založili Bates a Constable výzkumnou skupinu PRL na Cornellově univerzitě, aby vyvinuli systém pro vývoj programů, kde jsou programy vytvářeny

matematickým způsobem pomocí interního zdokonalování (PRL v té době znamenal Program Refinement Logic). V roce 1984 skupina PRL implementovala variantu extenzionální teorie typu Martin-Löfa v systému nazvaném NuPRL.

V širším slova smyslu slouží systém NuPRL jako nástroj pro experimentování se způsoby využití síly počítače při řešení problémů a pro generování přesných vysvětlení řešení, zejména v oblasti výpočetní matematiky.

NuPRL podporuje interaktivní prostředí, jak pro úpravu a generování důkazů, tak pro vyhodnocování funkcí. Styl logiky při důkazech je založen na paradigmatu postupného upřesňování. To znamená, že uživatel svůj cíl důkazu zmenší na podcíle. Například pokud je cíl důkazu $A\&B$ budou dva podcíle dokázání A a dokázání B.

Zdroj: [**18**].

5.1.2 Coq

Coq je formální systém řízení důkazů. Poskytuje formální jazyk pro psaní matematických definic, spustitelných algoritmů a vět spolu s prostředím pro semiinteraktivní vývoj strojově kontrolovaných důkazů. Typické aplikace zahrnují certifikaci vlastností programovacích jazyků (např. Projekt certifikace kompilátoru CompCert, Verified Software Toolchain pro verifikaci C programů nebo Iris framework pro souběžnou separační logiku), formalizace matematiky (např. Plná formalizace Feit-Tompsonovy věty) a výuky.

Coq implementuje programovou specifikaci a výšeúrovňový matematický jazyk *Gallina*, který je založen na expresivním formálním jazyce zvaném *Calculus of Inductive Constructions*, který sám o sobě kombinuje logiku vyššího řádu a bohatě typovaný funkcionální programovací jazyk.

Coq pomocí svého jazyka umožňuje:

- definovat funkce a predikáty, které mohou být efektivně vyhodnoceny,
- uvádět matematické věty a softwarové specifikace,
- interaktivně tvořit formální důkazy těchto vět,
- kontrolovat tyto důkazy strojově, s pomocí relativně malého jádra,
- exportovat programy do jazyků jako je *Objective Caml, Haskell* nebo *Scheme*.

Zdroj: [**19**].

5.1.3 Twelf

V roce 1987, Harper, Honsell a Plotkin představili edinburský Logical Framework, zkráceně Edinburgh LF. Jedná se o poměrně jednoduchou prediktivní teorii typů, obdobnou jako v Automath, ve které lze definovat logické systémy, za účelem uvažování o nich.

První implementace LF byla EFS (Enviroment for Formal Systems). Nedlouho poté, v roce 1989, Pfenning implementoval Elf systém, který přidal úroveň programování na meta levelu. V roce 1999 vyvinuli novou verzi Pfenning a Schürmann, nazvanou Twelf.

Programy v Twelfu (zvané podpisy – signatures) mohou být vykonány pomocí vyhledávací procedury, což umožňuje využívat Twelf jako logický programovací jazyk. Jeho jádro je více sofistikované než Prolog, jelikož je závisle typované a v logice vyššího řádu, ale je omezeno na čisté operátory – neobsahuje cut pravidlo nebo jiné extralogické operátory, které se vyskytují v implementaci Prologu. Kvůli tomu je Twelf méně vhodný pro praktické aplikace logického programování.

Twelf tedy obsahuje:

- implementaci logického rámce LF, která může být použitá ke kontrole typů LF reprezentace,
- logický programovací jazyk založený na LF,
- kontrolor metateorémů[2], který lze použít pro ověření důkazů vět o LF reprezentaci.

Twelf může dokázat ∀∃–metateorémy[3] jedním ze dvou způsobů. První metoda, dokazovač vět, je neúplná a v současné době se nedoporučuje ji používat. Umožňuje uživateli přímo specifikovat ∀∃–výroky o LF termech a poté požádat Twelf, aby toto prohlášení ověřil. Druhou metodou je napsat důkaz ∀∃–výroku v Twelfu ručně a potom Twelf využít k ověření totality tvrzení a tedy k prokázání správnosti důkazu.

Zdroje: [**20**], [**21**].

5.1.4 Agda

Agda je závisle typovaný funkcionální programovací jazyk, původně vytvořený Ulfem Norellem a Catarinou Coquand na Chalmers University of Technology. Jedná se o rozšíření Martin-Löfovy teorie typů. V současné době je vydána Agda 2, která by se měla považovat za nový jazyk, jelikož je to kompletně přepsaná stará verze.

Díky silnému typování a závislým typům, lze Agdu použít jako dokazovač, který umožňuje dokazovat matematické věty a spouštět tyto důkazy jako algoritmy.

Agda a další jazyky založené na teorii typů jsou úplné jazyky ve smyslu, že program typu T vždy skončí výpočet s hodnotou v T. Nemůže dojít k žádné chybě za běhu programu a nelze zapsat program, který neskončí (myšleno omylem).

[2]metateorém je teorém o objektovém jazyce
[3]označení pro kvantifikované teorémy

```
data Nat : Set where        _+_ : Nat -> Nat -> Nat      _*_ : Nat -> Nat -> Nat
  zero : Nat                zero  + m = m                zero  * m = zero
  suc  : Nat -> Nat         suc n + m = suc (n + m)      suc n * m = m + n * m
```

Obrázek 12: Definice násobení a sčítání v Agdě [**22**]

Zdroje: [**23**], [**24**].

5.2 LCF a následníci

LCF přístup k interaktivnímu dokazování vět má původ v práci Robina Milnera, který se od počátku své kariéry zajímal speciálně o interaktivní důkazy.

Milner se přestěhoval do Stanfordu, kde pracoval v letech 1971–1972. Společně s Whitfieldem Diffiem, Richardem Weyhrauchem a Malcolmem Neweyem vytvořili interaktivní dokazovač, který Milner nazval *Logic of Computable Functions (LCF)*. Tento formalismus navrhl Dana Scott v roce 1969, ačkoliv byl publikován až mnohem později. Byl zamýšlen k úvahám o rekurzivně definovaných funkcích na uspořádaných množinách. Dokazovací systém známý jako Stanford LCF byl zamýšlen spíše pro využití v oblasti počítačové vědy než pro čistou matematiku. Přestože se jednalo spíše o kontrolor důkazů než ATP, poskytoval výkonný mechanismus pro automatickou simplifikaci a podporu pro zpětné důkazy.

Důkazy se provádějí pomocí deklarace hlavního cíle a rozdělení tohoto na více podcílů. Využívá se k tomu fixní množina příkazů (například indukce ke generování základu a kroku důkazu). Podcíle jsou buď opět rozděleny, nebo přímo vyřešeny. Tímto jsou popsány také dva hlavní nedostatky systému:

- při opakovaném dělení na podcíle dochází k rychlému naplnění paměti,
- množina příkazů je fixní a není možné ji přizpůsobit.

V roce 1973 se Milner přesunul do Edinburgh University a založil projekt na vytvoření následníka Stanford LCF, později nazvaném Edinburgh LCF. Dva výše zmíněné problémy byly vyřešeny následovně:

- Místo ukládání celých důkazů si systém pamatoval pouze výsledky důkazů. Jednotlivé kroky byly sice provedeny, ale nebyly zaznamenány. K zajištění, že věta mohla být vytvořena pouze korektním důkazem, Milner využil abstraktní datový typ, jehož předdefinované hodnoty byly instancemi axiomů a jejichž operace byly odvozovacími pravidly. Přísná kontrola typů pak zajistila, že získané hodnoty byly pouze ty, které šlo získat z axiomů pomocí odvozovacích pravidel.
- Pro zvětšení množiny příkazů a možnost její úpravy byl vytvořen programovací jazyk ML (zkratka pro Meta Language). Tento jazyk byl striktně typovaný pro zajištění „bezpečnosti“ důkazů.

Ve Stanford LCF byly axiomy a odvozovací pravidla přímo zakódovány v mechanizmu simplifikace a vytváření podcílů. Uživatel tedy mohl důkazy tvořit pouze zpětně ze zadaných cílů. Kódováním logiky pomocí abstraktního typu, bylo v Edinburgh LCF umožněno psát důkazy dopředně. Cílem bylo implementovat důkaz pomocí nástrojů poskytovaných v ML. ML byl, kvůli těmto účelům, funkcionální jazyk, aby strategie tvorby podcílů mohly být reprezentovány jako funkce a operace kombinující tyto strategie jako funkce vyššího řádu. ML také obsahuje systém výjimek, který je nutný pro případ, že by strategie selhala (například v případě aplikace na nevhodný cíl).

LCF byl v Edinburghu použit v několika aplikacích a to motivovalo další vývoj systému. Gérard Huet přenesl Edinburgh LCF do LeLispu a MacLispu, což sloužilo jako základ pro racionalizaci a přestavbu systému Larry Paulsonem v Cambridge. Výsledkem této přestavby bylo Cambridge LCF. Edinburgh LCF byl interpretován. Během spolupráce Paulsona a Hueta byl vytvořen kompilátor ML, který poskytl dvacetinásobné zrychlení. Paulson během práce v Cambridge udělal velké pokroky, jak v pochopení designu a programování dokazovacích nástrojů, tak v samotné implementaci LCF.

Zdroj: [**25**].

5.2.1 HOL

Gordon se začal zajímat o formální verifikaci hardwaru, což byl důvod pro vývoj HOLu. Podporovaná modelovací technika reprezentuje hardwarová zařízení jako relace mezi vstupními a výstupními signály. Interní signály jsou schované pomocí existenciální kvantifikace. Signály se reprezentují pomocí funkcí z oboru času do oboru hodnot (znázornění stavů drátů). Z tohoto pohledu je tedy nutná logika vyššího řádu a možnost kvantifikace.

První verze HOL systému byla vytvořena jako modifikace Cambridge LCF parseru a printeru pro podporu logiky vyššího řádu. Termíny HOL byly zakodované pomocí LCF konstruktů tak, aby podporovaly maximální opětovné použití LCF kódu. Mnoho aspektů LCF– jako například typová kontrola, byly přeneseny do HOLu nezměněny. Primitivní axiomy a odvozovací pravidla byla upravena tak, aby byla správná pro logiku vyššího řádu.

Systém HOL, na rozdíl od LCF, zdůrazňuje spíše definici než postulaci axiomů jako metodu vývoje nových teorií. Logika vyššího řádu přímo umožňuje vývoj mnoha matematických objektů čistě definičně (například čísla, seznamy, atd.), což je v HOL systému podporováno. Jednoduché integrované definiční principy jsou nízkoúrovňové, ale odvozené principy vyšší úrovně lze naprogramovat v ML.

Melham implementoval odvozený princip definice typu, který převádí popisy rekurzivních datových typů do primitivních definic a poté automaticky odvozuje principy přirozené indukce a primitivní rekurze pro daný datový typ. Tento nástroj byl pravděpodobně zodpovědný za to, že se na HOL přestalo nahlížet jako na nástroj pro formální verifikaci a začal se brát jako obecný nástroj pro kontrolu a tvorbu důkazů.

Zdroj: [**26**].

5.2.2 Isabelle

V dnešní době je nejrozsáhlejší verzí Isabelle Isabelle/HOL, která umožňuje práci s logikou vyšších řádů. Isabelle/HOL dále obsahuje nástroje pro specifikaci jako jsou datové typy, indukční definice či rekurzivní funkce s komplexním porovnáváním vzorů.

Formálním dokazovacím jazykem systému je Isar. Cílem Isaru je překlenout pomyslnou propast mezi interními pojmy potřebnými k důkazům a rozumnou mírou abstrakce pro jednodušší práci ze strany uživatele. Isar byl vytvořen tak, aby byl deklarativní a zároveň rychle spustitelný pomocí Isar/VM interpretru. Na rozdíl od jiných systémů se Isar vyhýbá několika nedostatkům. Toho je docíleno tím, že je systém postaven pouze na několika základních principech, je nezávislý na základní logice a integruje širokou škálu metod automatického dokazování.

Subsystém Isar je pevně integrován do implementace metalogiky Isabelle/Pure. Teorie, věty a důkazní procedury mohou být zaměnitelně použity mezi klasickými dokazovacími skripty a Isabelle/Isar dokumenty. Isar je stejně obecný jako Isabelle a je schopen podporovat širokou škálu objektů logiky. Aktuální nastavení je pro koncového uživatele Isabelle/HOL.

Isabelle poskytuje vynikající podporu notací, to znamená, že lze zavádět nové pojmy na základě běžných matematických symbolů. Definice a důkazy mohou zahrnovat zdroje z LaTeXu, ze kterých Isabelle může rovnou generovat vysázené dokumenty.

Isabelle/HOL dále umožňuje převést spustitelné specifikace přímo do kódu v jazycích SML, OCaml, Haskell nebo Scala.

Isabelle přichází s velikou knihovnou formálně ověřených matematických nástrojů:

- základní teorie čísel,
- analýza – základní vlastnosti limit, derivace a integrály,
- algebra,
- teorie množin.

Isabelle, i přestože sama o sobě nabízí spoustu nástrojů pro automatické uvažování (např. přepisování termů nebo tabulkové dokazování), umožňuje využívat jiné automatické dokazovače pomocí nástroje Sledgehammer. Mezi podporované ATP patří například E, SPASS či Vampire.

Zdroj: [**27**].

5.3 Mizar

Mizar byl vytvořen v sedmdesátých letech týmem polských matematiků, lingvistů a informatiků, vedeného Andrzejem Trybulcem, kteří analyzovali matematické texty a snažili se najít formální protějšek orientovaný na člověka. Jazyk jako takový se snaží napodobit standardní matematickou praxi, což znamená, že verifikační engine je navržen tak, aby zachoval lidské porozumění důkazu a důkazy jako takové jsou postaveny na klasické logice a deklarativním stylu dokazování. Systém používá klasickou logiku prvního řádu a přirozenou dedukci.

Dokazovací jazyk je vytvořen tak, aby se co nejvíce blížil matematické mluvě a bylo možné jej automaticky ověřit/dokázat. Aby toho bylo dosaženo, je jazyk tvořen podmnožinou slov standardní angličtiny používaných v matematice. Dále je jazyk vysoce strukturovaný a umožňuje prefixovou, postfixovou i infixovou notaci funktorů. Formule jsou v Mizaru konstruovány pomocí predikátů, konstruktory predikátů jsou nazývány funktory. Při vytvoření nové proměnné musí být určen typ proměnné.

$\neg\alpha$	`not` α
$\alpha \wedge \beta$	α `&` β
$\alpha \vee \beta$	α `or` β
$\alpha \rightarrow \beta$	α `implies` β
$\alpha \leftrightarrow \beta$	α `iff` β
$\exists_x \alpha$	`ex x st` α
$\forall_x \alpha$	`for x holds` α
$\forall_{x:\alpha} \beta$	`for x st` α `holds` β

Obrázek 13: Logické spojky a kvantifikátory v jazyce Mizar [**28**]

according	aggregate	all	and
antonym	are	as	associativity
assume	asymmetry	attr	be
begin	being	by	canceled
case	cases	cluster	coherence
commutativity	compatibility	connectedness	consider
consistency	constructors	contradiction	correctness
def	deffunc	define	definition
definitions	defpred	do	does
end	environ	equals	ex
exactly	existence	for	from
func	given	hence	hereby
holds	idempotence	identify	if
iff	implies	involutiveness	irreflexivity
is	it	let	means
mode	non	not	notation
notations	now	of	or
otherwise	over	per	pred
prefix	projectivity	proof	provided
qua	reconsider	reduce	reducibility
redefine	reflexivity	registration	registrations
requirements	reserve	sch	scheme
schemes	section	selector	set
sethood	st	struct	such
suppose	symmetry	synonym	take
that	the	then	theorem
theorems	thesis	thus	to
transitivity	uniqueness	vocabularies	when
where	with	wrt	

Obrázek 14: Základní vyhrazené výrazy v jazyce Mizar [**29**]

5.3.1 Mizar Mathematical Library – MML

MML je knihovna všech článků napsaných v Mizaru, na které se mohou uživatelé odkazovat při psaní vlastních článků. Po schválení článku verifikátorem jsou takto nově napsané články hodnoceny komisí, ohledně stylu a vhodnosti. Pokud jsou přijaty, jsou přidány do MML a publikovány v Journal of Formalized Mathematics.

Zdroj: [**30**].

5.4 Prover9 & Mace4

Prover9 je ATP systém pro logiku prvního řádu, pracující na bázi rezoluce a paramodulace[4]. Jeho přímým předchůdcem byl dokazovač Otter, tento systém už ale není nadále vyvíjen. Oba tyto systémy vytvořil William McCune.

Prover9 má plně automatický mód, ve kterém mu uživatel jednoduše předá formule reprezentující problém. V dřívějších verzích Prover9 a také v systému Otter byly klauzule a formule dva různé typy objektů. V současné verzi jsou klauzule podmnožiny formulí. Formule i klauzule mohou mít volné proměnné, proto je zavedeno pravidlo, že proměnné začínají malými písmeny $u - z$, ostatní termy jsou konstanty.

Odvozovací pravidla v Prover9 probíhají na klauzulích. Pokud jsou na vstupu neklauzální formule, Prover9 je přeloží do požadované podoby pomocí NNF (Negation Normal Form), skolemizace[5] nebo CNF (Conjuctive Normal Form).

Prover9 bývá nejčastěji párován s programem Mace4, který hledá konečné modely a protipříklady. Mace4 tak může pomoci se vyhnout plýtvání časem pro dokazování tím, že najde protipříklad, případně může debugovat logické specifikace. Oba programy bývají spuštěné zároveň, při úspěchu jednoho tak může být druhý ukončen.

Dalším užitečným programem je Prooftrans. Když Prover9 najde důkaz věty, je tento důkaz vytisknut do výstupního souboru ve standardní formě. To znamená, že pro každý krok je přítomno odůvodnění kroku s dostatkem detailů tak, aby se dal důkaz zkontrolovat bez dalšího hledání. Prooftrans může důkazy ve výstupním souboru přetransformovat různými způsoby:

- žádná transformace,
- přečíslování kroků,
- zjednodušení odůvodnění jednotlivých kroků,
- vyprodukovat důkazy ve formě XML,

[4]paramodulace je odvozovací pravidlo, které generuje všechny „stejné“ verze klauzulí, modulované podmínkou s informací o rovnosti. Podrobnější vysvětlení tohoto pojmu lze nalézt například na [**47**].

[5]skolemizace je proces odstranění existenčních kvantifikátorů pomocí nových funkčních symbolů

- předložit důkazy pro kontrolu systémem IVY.

Zdroj: [**31**].

5.5 E theorem prover

Dokazovací systém E vyvinul Stephan Schulz. Jeho vývoj začal na Technické univerzitě v Mnichově, k prvnímu oficiálnímu vydání došlo v roce 1998.

E je dokazovací systém pro logiku prvního řádu s rovností. Přijímá problémovou specifikaci typicky sestávající z několika formulí prvního řádu a domněnky (výrok) v podobě klauzule nebo formule. Pokud je nalezen důkaz, tak systém může poskytnout seznam kroků důkazu, které mohou být samostatně ověřeny. Novější verze systému mohou také poskytnout možné hodnoty, v případě, že je výrok existenciální (tzn. pro nějaké X platí vlastnost P).

Hlavní dokazování probíhá ve čtyřech odlišných fázích:

- algoritmus klauzifikace převádí vstup do normální formy tak, že výsledná formule je nesplnitelná právě když původní problém je dokazatelný,
- volitelná fáze preprocessingu zjednodušuje výsledný problém,
- automatický režim analyzuje problém a vybírá vhodné vyhledávací parametry (opět volitelné),
- algoritmus postavený na základě superpozičního kalkulu[6] se snaží odvodit prázdnou klauzuli a tedy nesplnitelnost problému.

Zdroje: [**32**], [**33**].

5.6 Vampire

Vampire byl vyvinut Andreiem Voronkovem a jeho týmem. První oficiální vydání bylo v roce 1994, ačkoliv tomu předcházely další systémy. Vůbec první Voronkovův systém, který byl schopen něco dokázat, byl nazván Drakosha a vyvinut v roce 1989.

Vampire je ATP pro logiku prvního řádu. Mezi hlavní vlastnosti Vampire patří:

- vysoká rychlost – Vampire vyhrál soutěž CASC (viz níže) v různých divizích více než padesátkrát,
- běží na všech základních platformách, současně je použitelný i pro nezkušeného uživatele,

[6]kalkul používaný pro uvažování v rámci logiky prvního řádu. Místo aplikace rovnice v obou směrech se využívá redukční uspořádání. Podrobnější vysvětlení tohoto pojmu lze nalézt například na [**47**].

- implementuje unikátní strategii omezených zdrojů (limited resource strategy), která umožňuje hledat důkazy rychle i v omezeném čase,
- implementuje eliminaci symbolů, což umožňuje automaticky objevovat vlastnosti programů,
- má speciální mód pro práci se znalostními bázemi velkého rozsahu,
- umí dokazovat věty kombinací logiky prvního řádu a teorií, například celočíselné aritmetiky,
- je plně kompatibilní s TPTP syntaxí (viz níže). Zároveň podporuje i jiné syntaxe, pro provedení programové analýzy je schopen číst i programy napsané v jazyce C.

Zdroj: [**34**].

5.7 ACL2

ACL2 (A Computational Logic for Applicative Common Lisp) je systém skládající se z logického programovacího jazyka, ve kterém lze vytvářet modely počítačových systémů, a nástrojů pro dokazování vlastnosti modelů. Za předchůdce ACL2 by se dal považovat systém Nqthm. Nqthm je systém založený Robertem S. Boyerem a J. S. Moorem, jejichž cílem bylo vytvořit plně automatický logický dokazovač. Oba systémy patří do „rodiny" Boyer-Moore ATP.

Logika Nqthm je prvního řádu s rovností a obsahuje dva principy „rozšíření", podle nichž může uživatel zavádět nové koncepty se zárukou konzistence.

- The Shell Principle – umožňuje uživateli přidat axiomy zavádějící nové induktivně definované abstraktní datové typy.
- The Definitional Principle – umožňuje uživateli definovat nové funkce v logice.

Nqthm je pak mechanizace předešlé logiky. Systém je plně automatický v tom smyslu, že po spuštění už nepřijímá žádné rady od uživatele; jediný způsob, jak uživatel může do průběhu zasáhnout je ukončení pokusu o důkaz. Na druhou stranu je systém interaktivní v tom smyslu, že může získat větší důkazní sílu pomocí lemmat, které uživatel definoval a systém poté dokázal. Kvůli tomu je také nutná rozsáhlejší práce uživatele při dokazování obtížnějších vět – nutnost definovat jednodušší lemmata, která směřují k důkazu složitějších lemmat, potažmo vět.

Zdroje: [**35**], [**36**].

5.8 PVS – Prototype Verification System

PVS je mechanizované prostředí pro formální specifikaci a verifikaci. Skládá se z jazyka, který je založený na klasické, typované logice vyššího řádu, velkého množství předdefinovaných teorií, kontroloru typů, interaktivního dokazovacího systému, který podporuje použití několika rozhodovacích procedur, různých nástrojů včetně generátoru kódu či testeru, dokumentace, formalizovaných knihoven, a příkladů, které ilustrují různé metody využití systému v odlišných oblastech aplikace. [**37**]

Dokazovací system PVS poskytuje kolekci mocných primitivních odvozovacích procedur, které jsou interaktivně aplikovány pod uživatelovým dohledem v rámci sekvenčního kalkulu. Primitivní odvozovací pravidla zahrnují výroková a kvantizační pravidla, indukci, simplifikaci využívající rozhodovací procedury a lineární aritmetiku, abstrakci dat a predikátů. Implementace těchto primitiv jsou optimalizovány pro velké důkazy – například automatické přepisování je ukládáno do cache paměti pro vyšší efektivitu. Uživatelsky definované procedury mohou kombinovat tato primitiva a získat tak vyšší úrovně důkazních strategií. Mezi funkcemi jazyka PVS a dokazovacím systémem existuje určitá synergická interakce, takže informace o typu spojené s termem jsou využívány odvozovacími mechanismy a naopak automatizace v dokazovacím systému je užitečná při automatickém plnění/vyvracení TCC (type-correctness condition – podmínka správnosti typu). Důkazy dávají skripty, které mohou být editovány a připojeny k dalším formulím a znovu použity. Toto umožňuje efektivní dokazování vět, úpravu důkazů dle změn v požadavcích či designu a podporu vývoje čitelných důkazů.

Zdroj: [**37**].

5.9 Porovnání jednotlivých systémů

Porovnání jednotlivých systémů je celkem komplikované. Je to hlavně tím, že většina systémů je tvořena za nějakým účelem, některé jsou pro klasickou logiku prvního řádu, některé pro logiky řádů vyšších, některé jsou plně automatické, jiné naopak interaktivní pro spolupráci uživatele na důkazu. Následující tabulka tak znázorňuje jednotlivé hlavní parametry:

Characteristics	ACL2	E	Isabelle	Atelier B	Escher	Metamath	Twelf	Agda
System Type	TP	TP	TP	TP	TP	TP	TP	TP
Theorem Prover Category	ITP	ATP	ATP+ITP	ITP	ATP	ITP	ITP	ITP
System Based on	Syllogism	Syllogism	Syllogism	Syllogism	Syllogism	Syllogism	LF	FP
Logic Used	FOL	FOL	HOL	FOL	FOL+HOL	FOL+HOL	TT	HOL
System's Truth Value	Binary	Binary	Binary	Binary	Bin+Tri	Binary	Intuition	Binary
Calculus	Inductive	Deductive	Ded+Indu	Deductive	Deductive	Deductive	Deductive	Inductive
Set Theoretic Support	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Programming Paradigm	Func	Impe	Func	Impe	Func+Imp	Func	LP	Func
System Architecture	Modular	Mod+Mono	Modular	Monolithic	Monolithic	Mod+Mono	Monolithic	Modular
Programming Language	ACL2	C	ML+Scala	Prolog	C++	MM	SML	Haskell
User Interface	CLI	CLI	GUI	GUI	CLI+GUI	CLI+GUI	CLI	GUI
Platform Support	Cross	Cross	Cross	Cross	Win+Linux	Cross	Cross	Cross
Scalability	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Multi-threaded	Yes	No	Yes	No	No	Yes	No	No
IDE	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Library Support	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Programmability	Yes	No	Yes	No	No	Yes	Yes	Yes
Tactic Language Support	Yes	No	Yes	Yes	No	Yes	No	Yes

Obrázek 15: Porovnání jednotlivých dokazovačů 1 [**38**]

Characteristics	Mizar	HOL	RedPRL	Class & Int	Geo	Coq	PVS	CVC4
System Type	TP	TP	TP	TP	TP	TP	TP	TP4SMT
Theorem Prover Category	ITP	ITP	ITP	ATP	ATP	ITP	ITP	ATP
System Based on	Syllogism	Syllogism	Syllogism	Syllogism	GT	DP	Syllogism	DPLL
Logic Used	FOL	HOL	TT	FOL	PCL	HOTT	HOL	FOL
System's Truth Value	Binary	Binary	Intuition	Binary	3-value	Binary	Binary	Binary
Calculus	Deductive	Deductive	Deductive	Deductive	Deductive	Ded+Indu	Deductive	Deductive
Set Theoretic Support	Yes	No	No	Yes	No	Yes	Yes	Yes
Programming Paradigm	Decl	Func	Func	Decl	Decl	Func	Func+OO	Logical
System Architecture	Modular	Modular	Modular	Modular	Monolithic	Modular	Modular	Modular
Programming Language	Pascal	SML	SML	C++	C++	OCaml	C Lisp	C++
User Interface	CLI	CLI	GUI	CLI	CLI	CLI+GUI	GUI	CLI
Platform Support	Cross	Cross	Cross	Windows	Mac	Cross	Mac+Linux	Mac+Win
Scalability	Yes	Yes	No	Yes	Yes	Yes	Yes	No
Multi-threaded	No	No	No	Yes	No	No	Yes	Yes
IDE	Yes	No	Yes	Yes	No	Yes	Yes	No
Library Support	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Programmability	No	Yes	Yes	No	No	No	Yes	No
Tactic Language Support	No	Yes	Yes	Yes	No	Yes	Yes	No

Obrázek 16: Porovnání jednotlivých dokazovačů 2 [**38**]

Jako další možnost porovnání se nabízí soutěže pořádané mezi jednotlivými systémy. Mezi nejznámější patří CADE ATP System Competition. Tato soutěž je spojována s konferencemi CADE (Conference on Automated Deduction) a IJCAR (International Joint Conference on Automated Reasoning), které jsou organizovány Asociací pro automatické uvažování (AAR). Tyto konference slouží jako hlavní fóra pro prezentaci nových poznatků ohledně automatického dokazování.

Samotná CASC vyhodnocuje výkon plně automatických ATP systémů a to na základě:

- počtu vyřešených problémů,
- počtu vyřešených úloh s řešením na výstupu,
- průměrné doby běhu u vyřešených problémů.

Toto testování probíhá na základě TPTP knihovny (Thousands of Problems for Theorem Provers). Tato knihovna obsahuje jak testovací problémy pro ATP systémy, tak například seznam referencí pro dané problémy, instance obecných problémů, nástroje pro převod problémů do formátů stávajících systémů ATP, obecné pokyny pro hodnocení, či standardy pro vstup a výstup systémů ATP.

Na základě soutěže CASC jsou v posledních letech nejúspěšnější systémy Vampire a E.

Zdroj: [**39**]

6 Výhled do budoucna a využití ATP

6.1 Výhled do budoucna

6.1.1 QED

Cílem projektu QED je vybudovat jediné distribuované počítačové úložiště, které důsledně představuje všechny důležité zavedené matematické znalosti. Konstrukce tohoto systému bude významným vědeckým počinem vyžadujícím spolupráci mnoha matematiků, informatiků, výzkumných skupin, výzkumných agentur, univerzit a korporací. Tento systém bude mít výhody pro matematiku, vědu, technologii a vzdělávání. [**40**]

Devět důvodů, proč by měl být projekt QED zrealizován: [**41**]

- pomoci matematikům vyrovnat se s rozmachem znalostí z matematiky,
- pomůcka při vývoji vysoce komplexních systémů IT,
- pomoci v matematickém vzdělávání,
- poskytnutí kulturní památky,
- ochrana matematiky před korupcí,
- pomoc pro snížení „hladiny hluku“ publikované matematiky (při vysoké produkci matematických článků se může snadno stát, že dva různí lidé publikují stejné výsledky),
- zvýšení soudržnosti matematiky,
- přidání do těla výslovně formulované matematiky (většina matematiků se zabývá složitými problémy, na druhou stranu při formalizaci je potřeba formalizovat vše, včetně jednoduchých problémů),
- zlepšení nízké úrovně sebevědomí v matematice.

6.1.2 MathWiki

Pro lepší a častější využívání automatických dokazovačů je nezbytné mít rozsáhlé uložiště (či knihovnu) formalizované matematiky. Pro tvorbu takové knihovny je ovšem potřeba spoustu energie a času. Způsob jak toho dosáhnout je společné distribuované úsilí. Touto cestou se vydala MathWiki, která je de facto považována za Wikipedii formalizované matematiky. Každý, kdo má přístup k internetu, může snadno přispět napsáním svojí stránky na tomto webu. V době psaní této práce obsahuje MathWiki 1192 stránek v anglickém jazyce.

Kvůli koherenci a stabilitě webu MathWiki rozlišujeme dva typy stránek:

- formální stránky skládající se z anotovaného kódu důkazu pomocí automatického dokazovače,

- neformální stránky skládající se z popisu matematiky na vysoké úrovni s vykreslenými částmi formálního kódu.

U neformálních stránek lze postupovat podle postupů známých z Wikipedie. U formálních stránek může konzistenci zkontrolovat samotný dokazovač. [**16**]

6.2 Využití ATP

Automatické dokazovací systémy nalezly uplatnění hlavně ve dvou odvětvích: formalizace matematických vět a formální verifikace počítačových systémů. Každý ze systémů byl navržen za jiným účelem. Mizar byl navržen spíše pro formalizaci matematiky, o čemž svědčí i samotná Mizar Mathematical Library (MML), naopak HOL byl navržen spíše z důvodů verifikace.

6.2.1 Formalizace matematiky

V roce 1999 prezentovali Paul a Jack Abad „Sto nejzajímavějších vět/důkazů“. Tyto věty, případně důkazy, byly vybrané na základě následujících tří kritérií:

- výskyt věty v literatuře – množství a důležitost,
- kvalita důkazu,
- neočekávanost výsledku.

Seznam vět může čtenář najít na [**42**].

V době psaní práce bylo formalizováno 97 vět. Následující tabulka ukazuje nejúspěšnější systémy a počet formalizací, které provedly.

Dokazovací systém	Počet formalizací
HOL Light	86
Isabelle	84
Matemath	74
Coq	74
Mizar	69
Lean	55
ProofPower	43
PVS	22
nqthm/ACL2	18
NuPRL/MetaPRL	8

Tabulka 3: Formalizace sta nejzajímavějších vět/důkazů [**43**]

6.2.2 Programová verifikace

Formální verifikace má dva základní přístupy. Jedním je ověřování modelů (model checking), které spočívá v systematickém zkoumání matematického modelu daného systému. To obvykle spočívá v prozkoumání všech stavů a přechodů modelu. Výhodou tohoto přístupu je často plná automatizace, nevýhodou pak horší použití pro větší systémy.

Druhou metodou je deduktivní verifikace, která spočívá ve vygenerování matematických reprezentací ze systému a systémových specifikací a jejich následné dokazování pomocí ATP, například HOL, PVS či ACL2. Nevýhodou tohoto přístupu je, že uživatel by měl dobře znát systém a proč funguje správně a musí být schopen tyto informace předat verifikátoru.

V dokazovacích systémech bylo provedeno mnoho studií programové korektnosti. Některé systémy jsou využívané pro průmyslové projekty. Například NASA používá PVS k ověřování softwaru pro leteckou kontrolu; Intel zase využívá HOL Light k ověření návrhu nových čipů.

ACL2 se využívá pro ověření, že popis základních aritmetických obvodů pro práci s plovoucí čárkou v AMD Athlon procesoru implementuje IEEE standard. Další průmyslové využití ACL2 lze najít v [**44**].

7 Dokazovač – manuál

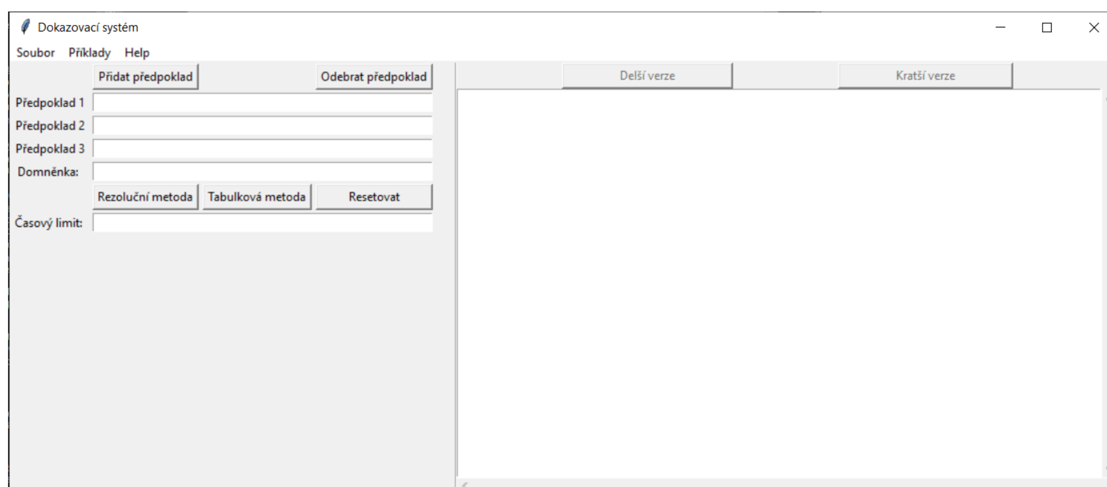
Aplikace umožňuje uživateli provádět důkazy rezoluční metodou a tabulkovou metodou (sémantické vyplývání). Aplikace se dá použít i pro vykreslení tabulky pro jednu konkrétní formuli a její převod do konjunktivní normální formy.

Pro jednodušší zápis formulí pomocí klávesnice (bez nutnosti psát logické spojky složitými klávesovými zkratkami) se v aplikaci používá následující zápis:

− (znak mínus)	negace proměnné
+ (znak plus)	disjunkce dvou proměnných
∗ (znak hvězdička)	konjunkce dvou proměnných
> (znak větší)	implikace
= (znak rovno)	ekvivalence

Proměnné mohou být libovolně (rozumně) dlouhé řetězce alfanumerických symbolů (písmena a čísla) s tím, že první symbol musí být písmeno, tzn. například „a“, „B“, „AbCd1234“ jsou legitimní názvy proměnných. Naopak „1“, „/“, „1234AbCd“ jako jména proměnných použít nelze.

Při spuštění aplikace se objeví následující okno:



Obrázek 17: Hlavní okno aplikace

Okno se skládá z několika částí:

- levá část pro zadávání předpokladů, domněnky a časového limitu,
- pravá část pro zobrazení tabulek a důkazu,
- horní část obsahující menu.

V menu se nachází následující položky:

- *nový* – vytvoření nového důkazu. Je umožněno uložení důkazu, pokud tak uživatel již neučinil sám.
- *uložit* – pro uložení důkazu.
- *ukončit* – pro zavření celé aplikace.
- *příklady* – zde jsou obsaženy předpřipravené množiny předpokladů a domněnek, které si může uživatel projít pro lepší pochopení zápisu formulí.
- *help* – pro otevření okna s nápovědou.

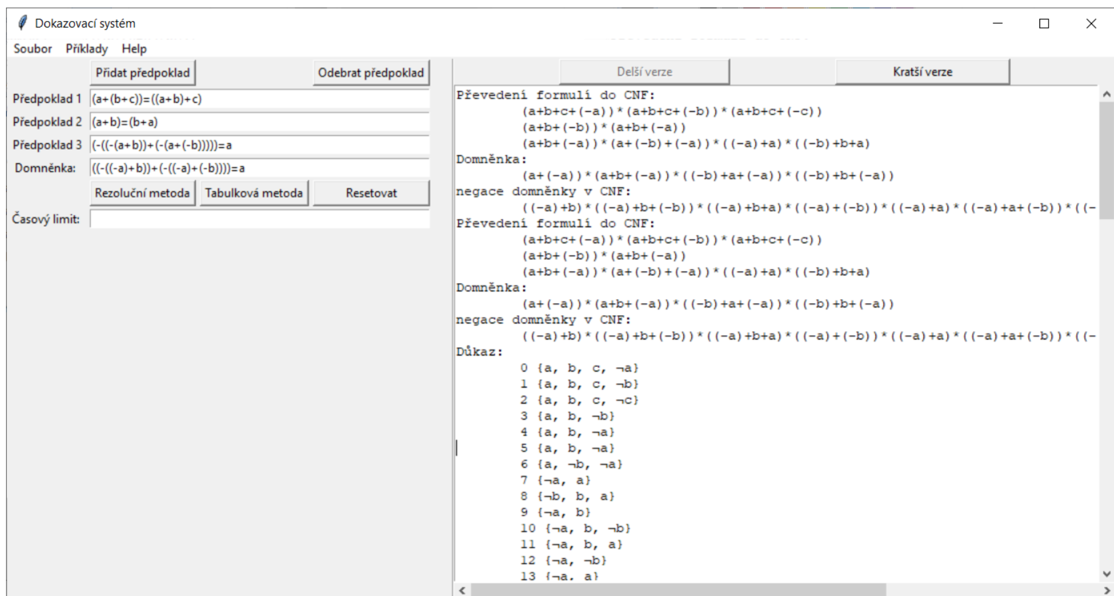
Levá část umožňuje zápis předpokladů a domněnky, zároveň je zde možno zadat časový limit běhu důkazu (pokusu o důkaz). V horní části se nacházejí tlačítka pro přidávání a odebírání předpokladů. Pro jednodušší běh aplikace je potřeba nenechávat nevyplněná pole pro předpoklady, tedy když máte čtyři předpoklady, tak pomocí zmíněných tlačítek vytvoříte čtyři vstupní pole. Počet předpokladů je omezen na 0 až 10. Časový limit se udává v sekundách. Pokud časový limit není zadán, tak se automaticky bere hodnota 60 sekund. Zároveň platí, že časový limit se používá pouze pro rezoluční metodu. Po spuštění je potřeba vyčkat (než začne uživatel vykonávat další akce) na sdělení výsledku nebo uběhnutí časového limitu.

Po nastavení potřebného počtu vstupních polí a jejich vyplnění (ať už ručně nebo pomocí předpřipravených příkladů) se může spustit samotné dokazování.

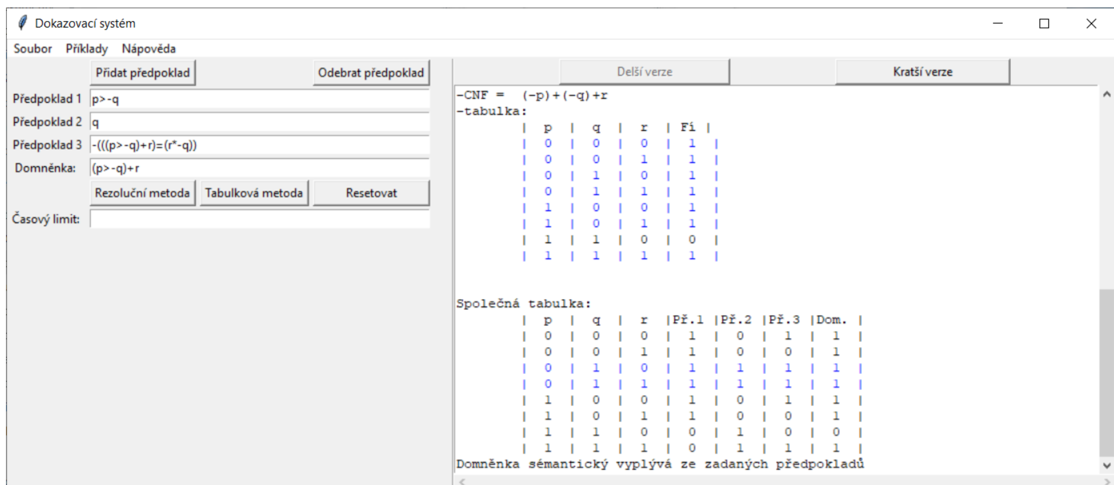
Ke spuštění důkazu slouží tlačítka pod vstupním polem pro domněnku. Konkrétně tedy tlačítka *Rezoluční metoda* a *Tabulková metoda*. Tlačítko *Resetovat* vše vynuluje.

Po spuštění konkrétního důkazu se v pravé části okna vypíše odpovídající text. Tím je v případě rezoluční metody buď celý důkaz, nebo jeho úvod (převedení formulí do CNF) a hláška, že domněnku nelze dokázat ze zadaných předpokladů nebo vypršel čas. V případě tabulkové metody se vykreslí všechny tabulky, tzn. tabulky pro předpoklady, domněnku a společná tabulka, ze které jde vidět, zda domněnka sémanticky vyplývá ze zadaných předpokladů.

Po vypsání důkazu lze určit délku důkazu, na výběr je krátká a dlouhá verze. Krátká verze v případě rezoluční metody vypíše pouze ty řádky, které byly nutné pro samotný důkaz, u tabulkové metody se pak vykreslí pouze tabulka domněnky a společná tabulka. V případě dlouhých verzí se vypisuje vše.



Obrázek 18: Příklad důkazu rezoluční metodou



Obrázek 19: Příklad důkazu tabulkovou metodou

Po vypsání důkazu je možné si důkaz uložit, to už je velice jednoduché a lze to udělat pomocí tlačítka *uložit*, které bylo zmíněno výše. Po kliknutí na něj se otevře klasické okno pro ukládání, kde se vyplní jméno a důkaz je uložen.

8 Dokazovač – popis implementace

Aplikace je tvořena několika základními třídami:

- Variable – reprezentace proměnných ve formuli,
- Operator – reprezentace logických spojek,
- LeftParenthesis a RightParenthesis – reprezentace závorek,
- Formule – reprezentace samotných formulí,
- Proof – důkaz pomocí rezoluční metody,
- TableProof – důkaz pomocí tabulkové metody,
- Window – třída pro hlavní okno celé aplikace,
- HelpWindow – třída pro okno s nápovědou.

8.1 Třída Variable

Tato třída se používá pro reprezentaci proměnné. Proměnná je pro potřeby této práce chápána jako řetězec alfanumerických symbolů, začínající na libovolné písmeno, tzn. například „a“, „A“, „AbCd1234“ jsou legitimní instance proměnné.

Třída Variable implementuje následující metody:

- konstruktor – ten bere jako argumenty jméno proměnné a její hodnotu (myšleno 0 nebo 1), tento argument nemusí být při konstrukci proměnné zadán, hodnota proměnné je poté nastavena na *None*. To je z toho důvodu, že pro důkaz pomocí rezoluční metody je hodnota proměnné irelevantní, stejně tak pro tabelaci formule, kde se prochází všechna pravdivostní ohodnocení.
- parse – tato metoda je zavolaná při konstrukci proměnné. Jako argument dostává jméno proměnné v podobě typu *string*. V případě neplatného znaku je vyvolána výjimka, viz následující kód.
- eval – metoda bez argumentu, vrací hodnotu proměnné. V případě, že hodnota nebyla inicializovaná, tak vyvolá výjimku.
- str – metoda bez argumentu, vrací jméno proměnné.

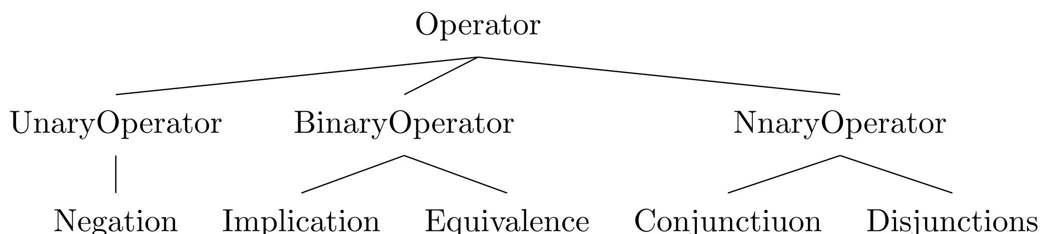
```
def parse(self, string):
    #pokud začíná string písmenem, je to proměnná
    #pokud čímkoliv jiným, tak vrátit chybu
    #po písmenu může být řetězec písmen a čísel
    index = 0
    if string[0].isalpha():
      for s in string:
        if not s.isalnum():
          raise Exception("Neplatný vstup")

    return True
```

Zdrojový kód 1: Metoda parse třídy Variable

8.2 Třída Operator

Logické spojky jsou rozděleny do tří podtříd – *UnaryOperator*, *BinaryOperator* a *NnaryOperator*. Do první zmíněné patří negace, do druhé implikace a ekvivalence a do třetí konjunkce a disjunkce. Ačkoliv jsou konjunkce a disjunkce binární operace, pro potřeby reprezentace jsou brány jako operace n-ární. Formule (viz níže) je reprezentovaná jako list v prefixové podobě, proto je jednodušší mít uloženo například [konjunkce A B C] místo [konjunkce A [konjunkce B C]].



Obrázek 20: Třída Operator a její podtřídy

Každá ze tříd implementuje metody str (pro výpis na obrazovku), konstruktor (pro vytvoření nové instance), precedence (důležitost dané spojky oproti ostatním) a eval (vyhodnocení dané spojky se zadanými operandy). Všechny tyto metody jsou pro každou třídu různé, v závislosti na logický význam spojky a znaku, který ji v aplikaci reprezentuje.

U metody eval je vždy nejprve nutno určit operandy pro logické vyhodnocení. U vyhodnocování formule se totiž může stát, že by například implikace dostala jako jeden argument proměnnou a jako druhý už výsledek evaluace jiné spojky, což je pravdivostní hodnota. Takovýto případ by skončil chybou programu.

Pro lepší názornost je v následujícím kódu představena třída *Implication*.

```
class Implication(BinaryOperator):
 def __init__(self):
  super().__init__(">")

 def precedence(self):
  return 0

 def __str__(self):
  return ">"

 def eval(self):
  if isinstance(self.op1, Variable):
   op1 = self.op1.eval()
  else:
   op1 = self.op1

  if isinstance(self.op2, Variable):
   op2 = self.op2.eval()
  else:
   op2 = self.op2

  if op1:
   if op2:
    return True
   else:
    return False
  else:
   return True
```

Zdrojový kód 2: Třída Implication

8.3 Třída Formule

Tato třída reprezentuje formuli výrokové logiky a umožňuje s ní pracovat.

Metody třídy Formule:

- metoda split – tato metoda rozdělí řetězec znaků na vstupu do jednotlivých částí, aniž by řešila jejich syntaktickou správnost. Na vstupu se tedy očekává argument typu *string* a výstupem je seznam částí. Metoda vždy vezme první nezpracovaný znak a mohou nastat následující možnosti:
  - Znak je písmeno – v tomto případě jde o jméno proměnné. Jako jméno proměnné se vezme aktuální znak a všechny následující alfanumerické znaky. Takto získané jméno se použije pro vytvoření nové proměnné (pokud proměnná s tímto jménem ještě neexistuje), která se uloží do seznamu pro výstup.

- Znak je jeden z vyhrazených znaků pro operátory – vytvoří se nová instance daného operátoru a uloží se do výstupního seznamu.
- Znak je mezera – v tomto případě se nestane nic, pouze se přejde na následující znak.
- Znak na vstupu neodpovídá žádnému z předcházejících případů – je vyvolána výjimka.

- metoda *parse* – metoda pro kontrolu syntaktické správnosti vstupu. Jako vstup bere seznam částí vzniklých metodou *split*. Výstupem je pravdivostní hodnota *True* pokud je tento vstup syntakticky správný, *False* v opačném případě. Kontrola probíhá na základě několika jednoduchých pravidel:
  - jestliže je délka vstupu rovna jedné a jeho jediný prvek je proměnná, tak je vstup validní;
  - pokud je prvním prvkem negace, tak je vstup validní, jestliže je validní i bez prvního prvku (bez negace);
  - jestliže je prvním prvkem levá závorka a posledním znakem pravá závorka, pak je vstup validní, pokud je validní obsah mezi závorkami;
  - jestliže je prvním prvkem levá závorka, ale posledním prvkem není pravá závorka, tak je formule ve tvaru *(smt1) op smt2*. V tomto případě je vstup validní, jestliže *op* je operátor (kromě negace), a *smt1* i *smt2* jsou validní;
  - jestliže je prvním prvkem proměnná, pak je formule validní, jestliže druhy prvek je operátor (kromě negace) a zbytek vstupu je validní;
  - v každém jiném případě vstup není validní.

- metoda *makeList* – metoda ze seznamů částí vytvoří seznam reprezentující formuli v prefixové podobě. Při tvorbě seznamu se postupuje následovně:
  - délka vstupu je nula (podmínka pro ukončení rekurze), jako výsledek se vrátí prázdný seznam;
  - první prvek na vstupu je proměnná. V takovém případě je buď délka vstupu rovna jedné a tím pádem je vrácená samostatná proměnná, nebo je délka větší než 2 (zde je potřeba si uvědomit, že vstupní seznam prošel metodou *parse*, takže možnost, že délka je rovna dvěma nemůže nastat), v tom případě je formule ve tvaru *(var op rest)* a jako výsledek se vrátí
  $$[op var makeList(rest)];$$
  - první prvek na vstupu je negace. Zde opět mohou nastat různé možnosti:
    * za negací je proměnná. V tomto případě se postupuje obdobně jako v předchozím bodě s tím rozdílem, že negovaná proměnná se ukládá jako [neg var],

* za negací je levá závorka. Tento případ odpovídá následujícímu kódu, kde *index* určuje pozici operátoru,

```
elif isinstance(parts[0], Negation):
  # pokud je za negací závorka, tak se může negace vztahovat bud' k
    celému výrazu nebo jen k jeho první části
  # potřeba rozdělit na prvvní operátor druhý
  if isinstance(parts[1], LeftParenthesis):
    first, second, index = self.getFirstSecond(parts[1:])
    first_vysl = self.makeList(first[1:-1])
    if len(second) > 0:
      second_vysl = self.makeList(second)
      # neg first op second ---> [op [neg first] second]
      return [parts[index], [parts[0], first_vysl], second_vysl]
    else:
      return [parts[0], first_vysl]
```

Zdrojový kód 3: Metoda makeList – negace

* za negací je další negace. V tomto případě se vytvoří list z prvků za druhou negací (aplikuje se zákon dvojí negace).
  - prvním prvkem na vstupu je levá závorka. Zde se určí první a druhá část formule (v závislosti na pořadí operátorů). Vytvoří se list pro první část formule a v případě, že druhá část není prázdná (prázdná by byla v případě, kdy celá formule je v závorkách), tak i list pro druhou část formule, a je vrácen list ve tvaru [operace makeList(first) makeList(second)]. Je-li operace konjunkce nebo disjunkce, tak se zkouší, zda druhá část neobsahuje stejnou operaci. Tady se právě využívá n-árnost těchto dvou spojek.

- metoda *removeEquivalences* – metoda pro odstranění ekvivalencí ve formuli. Ekvivalence nejsou nahrazeny za dvě implikace, nýbrž rovnou za konjunkci disjunkcí. $A \Leftrightarrow B$ je tedy nahrazeno za $(A \vee \neg B) \wedge (\neg A \vee B)$. Metoda, stejně jako i další, probíhá rekurzivně, takže se odstraní ekvivalence z A i B před tím, než se odstraní ekvivalence v $A \Leftrightarrow B$.

- metoda *removeImplications* – metoda odstraňuje implikace ve formuli. Každý výskyt $A \Rightarrow B$ je nahrazen za $\neg A \vee B$.

- metoda *removeDoubleNegation* – metoda odstraňující dvě po sobě jdoucí negace. Každý výskyt $\neg\neg A$ je nahrazen za A.

- metoda *deMorgan* – metoda implementující de Morganovy zákony. Výskyt $\neg(A \vee B)$ nahradí za $(\neg A \wedge \neg B)$ a $\neg(A \wedge B)$ nahradí za $(\neg A \vee \neg B)$

```
1  def removeEquivalences(self, lst):
2   if isinstance(lst, list):
3    if isinstance(lst[0], Equivalence):
4     removedFirst = self.removeEquivalences(lst[1])
5     removedSecond = self.removeEquivalences(lst[2])
6     return [Conjunction(), [Disjunction(), removedFirst, [Negation
         (), removedSecond]], [Disjunction(), [Negation(),
         removedFirst], removedSecond]]
7    else:
8     removedFirst = self.removeEquivalences(lst[1])
9     if len(lst) == 3:
10     removedSecond = self.removeEquivalences(lst[2])
11     return [lst[0], removedFirst, removedSecond]
12    elif len(lst) > 3:
13     l = len(lst)
14     vysl = []
15     for op in range(1, l):
16      vysl.append(self.removeEquivalences(lst[op]))
17      return [lst[0]] + vysl
18    else:
19     return [lst[0], removedFirst]
20  else:
21   return lst
```

Zdrojový kód 4: Metoda removeEquivalences

- metoda *orOverAnd* – metoda pro distribuci disjunkce přes konjunkci. Čili $A \vee (B \wedge C)$ se nahradí za $(A \vee B) \wedge (A \vee C)$. Je-li první operací konjunkce, tedy list dané formule je ve formě

  $$[konjunkce, operand1, operand2, \dots],$$

  tak se vrátí seznam ve tvaru

  $$[konjunkce, orOverAnd(operand1), orOverAnd(operand2), \dots].$$

  Je-li první operací disjunkce, tedy formule je ve tvaru

  $$[disjunkce, operand1, operand2, \dots],$$

  pak je nutné zjistit, zda některý z operandů (rovněž po aplikaci metody *orOverAnd*) obsahuje konjunkci. Pokud ano, tak dojde k distribuci disjunkce přes danou konjunkci, v opačném případě je formule už v CNF a vrátí se původní formule.

- metoda *simplify* – metoda pro zjednodušení seznamu ve smyslu nadbytečného zanořování stejných spojek. Například $(A \vee B) \vee (C \vee D)$, což by bylo reprezentováno jako

  $$[or, [or, A, B], [or, C, D]]$$

se nahradí za $(A \vee B \vee C \vee D)$ s následující reprezentací

$$[or, A, B, C, D].$$

- metoda *eraseSameLiterals* – metoda pro odstranění dvou (či několika) shodných literálů v rámci jedné klauzule. Například $(A \vee B \vee A)$ je nahrazeno za $(A \vee B)$.

- metoda *eraseSameClauses* – metoda pro odstranění dvou (či několika) shodných klauzulí v rámci jedné formule. Například $(A \vee B) \wedge (A \vee B)$ je nahrazeno za $(A \vee B)$. Funkce funguje tak, že se pro každou klauzuli zavolá předcházející metoda (*eraseSameLiterals*), poté se všechny klauzule převedou do formy řetězce symbolů (vezmou se jména proměnných a pro disjunkci se použije znak nula). Takto reprezentované klauzule se setřídí (to se dělá z toho důvodu, že z pohledu logické hodnoty klauzule jsou $(A \vee B)$ a $(B \vee A)$ shodné). Po setřídění se vezmou všechny unikátní řetězce a jim odpovídající klauzule a tyto jsou vráceny (společně s konjunkcí) jako výsledek.

- metoda *convertToCNF* – metoda pro převod formule do konjunktivní normální formy, která je potřebná pro důkaz rezoluční metodou. Využívá k tomu výše popsané metody. V přiloženém kódu lze vidět, že metody pro odstranění dvojích negací, provedení de Morganových zákonů a distribuci disjunkce přes konjunkci jsou opakovány vícekrát (*depth* v kódu určuje konkrétní počet, tento počet je roven hloubce daného listu, např. [1, [[2], 3],4] by měl hloubku 3). Opakování těchto metod je nutné, jelikož jedno provedení nezaručuje převod do CNF.

```
1  def convertToCNF(self):
2     self.original = self.list
3     self.list = self.removeEquivalences(self.list)
4     self.list = self.simplify(self.list)
5     self.list = self.removeImplications(self.list)
6     self.list = self.simplify(self.list)
7     self.list = self.removeDoubleNegation(self.list)
8     depth = self.depth(self.list)
9     for i in range(depth):
10       self.list = self.deMorgan(self.list)
11       self.list = self.removeDoubleNegation(self.list)
12       self.list = self.orOverAnd(self.list)
13       self.list = self.simplify(self.list)
14    self.list = self.eraseSameClauses(self.list)
15    return self.list
```

Zdrojový kód 5: Metoda convertToCNF

- metoda *eval/evalForAll* – metody pro získání pravdivostní hodnoty formule. Metoda *eval* bere jako argument seznam binárních hodnot, které se navážou na proměnné ve formuli dle abecedního pořadí proměnných. Po navázání hodnot se zavolá pomocná funkce *eval_list*, která už provede samotné vyhodnocení listu.

  Metoda *evalForAll* pak počítá všechna pravdivostní ohodnocení formule a využívá k tomu právě metodu *eval*. Funkce *getValues* v kódu níže převádí celé nezáporné číslo do binární reprezentace v podobě seznamu. Zároveň tato metoda vytvoří tabulku pravdivostních ohodnocení, která je použitá při vykreslování.

```
1  def evalForAll(self):
2     self.allResults = []
3     self.eval_table = []
4     l = len(self.variables)
5     for i in range(2**l):
6        values = self.getValues(i)
7        vysl = self.eval(values)
8        self.allResults.append(vysl)
9        if vysl == True:
10          vysl_int = 1
11       elif vysl == False:
12          vysl_int = 0
13       self.eval_table.append(values + [vysl_int])
14    return True
```

Zdrojový kód 6: Metoda evalForAll

- konstruktor – při vytvoření nové formule se rovnou provedou metody *split* a *parse* pro kontrolu správnosti zápisu formule a poté je pomocí *makeList* vytvořen list, který tuto formuli reprezentuje interně.

8.4 Třída Proof

Třída slouží pro reprezentaci, respektive tvorbu důkazů pomocí rezoluční metody.

Metody třídy Proof:

- metoda *get_clauses* – metoda pro vytvoření listu klauzulí. Jako vstup bere list reprezentující formuli. Pro formuli $(A \vee \neg B) \wedge (\neg A \vee B)$ a její reprezentaci ve tvaru

$$[and, [or, A, [neg, B]], [or, [neg, A], B]]$$

  dostaneme

$$[[[A, 1], [B, 0]], [[A, 0], [B, 1]]],$$

  kde [A, 1] značí pozitivní výskyt literálu a [A, 0] naopak negovaný výskyt literálu.

- metoda *resolve_clauses* – metoda pro provedení rezoluční metody přes dvě klauzule. Vstupem jsou tedy dvě klauzule, výstupem je seznam všech možných klauzulí, které lze odvodit ze dvou původních.

```
1  def resolve_clauses(self, clause1, clause2):
2    resolvents = []
3    for literal_one in clause1:
4      for literal_two in clause2:
5      #     pokud je proměnná stejná a hodnota proměnné je opačná
6        if (literal_one[0] == literal_two[0]) and (literal_one[1] !=
             literal_two[1]):
7          tmp1 = [x for x in clause1 if x != literal_one]
8          tmp2 = [x for x in clause2 if x != literal_two]
9          unique = tmp1
10         for item in tmp2:
11           if item not in unique:
12             unique.append(item)
13         resolvents.append(unique)
14
15       else:
16         continue
17   return resolvents
```

Zdrojový kód 7: Metoda resolve_clauses

- metoda *resolution* – metoda pro provedení důkazu rezoluční metodou. Jako vstup bere číslo odpovídající časovému limitu pro její běh. Pokud toto číslo není určeno uživatelem, je momentálně nastaveno na 60 sekund. Metoda funguje tak, že pro každé dvě klauzule si určí resolventy (klauzule z nich odvozené). V případě, že některá resolventa je prázdná množina, pak byl důkaz úspěšný (například pro $\{a\}$ a $\{\neg a\}$ se odvodí $\{\}$). V opačném případě se takto získané resolventy přidají do množiny *new*, která obsahuje všechny nově získané klauzule v aktuálním kole. Po průchodu přes všechny dvojice klauzulí se zkontroluje, zda množina *new* není podmnožinou původních klauzulí, to by znamenalo, že důkaz bude cyklit a tedy neskončí. Pokud podmnožinou není, tak se všechny klauzule z této množiny přidají do množiny původních klauzulí a celý průběh se opakuje.

```
def resolution(self, limit):
 new = []
 if limit == None:
  limit = 60

 with self.time_limit(limit):
  while(True):
   for clause1 in self.all_clauses:
    for clause2 in self.all_clauses[self.all_clauses.index(clause1
        )+1:]:
     resolvents = self.resolve_clauses(clause1[1], clause2[1])
     if any(len(x) == 0 for x in resolvents):
      self.all_clauses.append((self.numberOfClauses, [], clause1
          [0], clause2[0]))
      return True
     else:
      for item in resolvents:
       if item not in [x[1] for x in new]:
        new.append((self.numberOfClauses, item, clause1[0],
            clause2[0]))
        self.numberOfClauses += 1
   if all(x in [y[1] for y in self.all_clauses] for x in [y[1] for
       y in new]):
    return False
   else:
    for item in new:
     if item not in self.all_clauses:
      self.all_clauses.append(item)
   new = []
```

Zdrojový kód 8: Metoda resolution

- metoda *get_all_previous* – metoda pro určení všech klauzulí, které byly potřebné pro odvození zadané klauzule.
- metoda *get_froms* – metoda, která pro zadanou klauzuli vrací dvě klauzule, z nichž byla odvozena zadaná klauzule.
- metoda *get_short_version* – v určitých případech je důkaz zbytečně dlouhý, některé klauzule nejsou vůbec použity. Teoretický příklad: máme množinu předpokladů ve tvaru $\{a, b, c\}$ a chceme dokázat $\{a\}$. Důkaz by vypadal následovně:

0	$\{a\}$
1	$\{b\}$
2	$\{c\}$
3	$\{\neg a\}$
4	$\{\}$ 0 3

V tomto teoretickém příkladu se jedná pouze o dva nadbytečné řádky, ale u delších důkazů se může jednat řádově o desítky či stovky. K odstranění těchto řádků, které jsou pro pointu důkazu zbytečné, slouží právě metoda *get_short_version*. Funguje tak, že zavolá *get_all_previous* na výslednou klauzuli.

8.5 Třída TableProof

Třída slouží pro vytvoření tabulky, které obsahuje všechna pravdivostní ohodnocení pro všechny předpoklady a domněnku. Nejprve se určí všechna unikátní jména proměnných. Ty se jednak uloží do seznamu, z kterého se později vytvoří záhlaví tabulky a jednak se dle počtu proměnných generují pravdivostní ohodnocení. Pro konkrétní ohodnocení se vždy vypočítají hodnoty všech předpokladů a domněnky a společně s ohodnocením se uloží jako řádek tabulky. Takto získaná tabulka je poté přístupné pro tisk, společně se seznamem pro záhlaví.

8.6 Třída Window

Třída reprezentující hlavní okno celé aplikace. Okno využívá knihovnu Tkinter (knihovna v pythonu pro tvorbu GUI). Z hlediska designu je okno rozděleno pomocí *tkinter.grid* na levou a pravou část. Levá část je opět rozdělena pomocí gridu a zahrnuje potřebné labely (popisky), tlačítka a vstupní pole pro zadání potřebných údajů. Pravá strana obsahuje textové pole pro výpis důkazu a tlačítka pro možnost volby délky důkazu.

Z hlediska funkčnosti třída obsahuje klasické metody pro ukládání důkazu, resetování celého okna, pro ukončení programu, či otevření okna s nápovědou. Mezi z autorova pohledu zajímavější metody patří následující:

- metoda *resolut_run* – metoda pro spuštění důkazu rezoluční metodou. Nejprve se ze vstupu zjistí časový limit (ten nemusí být specifikován), předpoklady a domněnka. Všechny zjištěné formule se převedou do CNF a vytvoří se instance důkazu (třída *Proof*). Poté se zavolá metoda *proof.resolution*, v případě jejího úspěchu se důkaz vypíše do textového pole.

```
1 if self.proof.resolution(time_limit):
2    self.print_resolution_proof(self.proof.all_clauses)
3
4 else:
5    self.textbox.insert(END, "Domněnku ze zadaných předpokladů nelze
         dokázat nebo vypršel čas \n")
6 self.short_btn['state'] = NORMAL
```

Zdrojový kód 9: Metoda resolut_run – vykreslení důkazu

- metoda *table_run* – metoda pro vytvoření tabulek pro všechny formule.

```
1  def table_run(self):
2     self.resolution_prooved = False
3     self.table_prooved = True
4     self.clear_canvas()
5     self.entries = []
6     self.axioms = []
7     for i in range(self.numberOfAxioms):
8       try:
9         entry = self.axiom_entries[i].get()
10        fle = Formule(entry)
11        fle.evalForAll()
12        fle.convertToCNF()
13        self.entries.append(fle)
14      except Exception:
15        messagebox.showerror("Chyba", "V předpokladu {} se nachází
            nepovolený znak nebo je špatně zapsán".format(i+1))
16        return
17    try:
18      entry = self.conc_entry.get()
19      self.conclusion = Formule(entry)
20      self.conclusion.evalForAll()
21      self.conclusion.convertToCNF()
22    except Exception:
23      messagebox.showerror("Chyba","V domněnce se vyskytl neznámý
          znak")
24      return
25    self.print_table_proof(True)
26    self.short_btn['state'] = NORMAL
```

Zdrojový kód 10: Metoda table_run

- metody *add_axiom* a *del_axiom* – tyto metody se starají o přidávání a odstraňování labelů a vstupů pro předpoklady. Vždy je nutné posunout tlačítka a vstup pro domněnku, o to se v rámci těchto metod stará pomocná funkce *move_buttons* (ta pouze posouvá konkrétní tlačítka atd., v rámci mřížky v levé části okna). Zároveň funkce hlídají, aby počet předpokladů byl v určeném rozmezí (to je v rámci aplikace nastaveno na 0 – 10 předpokladů).
- metody *long_version* a *short_version* – metody vyvolané stiskem tlačítka pro dlouhou/krátkou verzi důkazu. Stisk konkrétního tlačítka ho učiní neaktivním a naopak druhé tlačítko aktivním. Zjistí se, který typ důkazu se má vykreslit (rezoluční nebo tabulkovou metodou) a zavolá se příslušná metoda pro vykreslení důkazu.

```
1  def short_version(self):
2     self.long_btn['state'] = NORMAL
3     self.clear_canvas()
4     if self.resolution_prooved:
5        short_version = self.proof.get_short_version()
6        self.print_resolution_proof(short_version)
7     elif self.table_prooved:
8        self.print_table_proof(False)
9     self.short_btn['state'] = DISABLED
```

Zdrojový kód 11: Metoda short_version

- metody *print_resolution_proof* a *print_table_proof* – slouží k vykreslení daného typu důkazu.

8.7 Třída HelpWindow

Tato třída reprezentuje okno, obsahující nápovědu, respektive manuál. Obsah tohoto okna odpovídá části „Dokazovač – manuál“.

Závěr

Hlavní část práce se zaměřuje na počítačové dokazování. Nejprve jsou zavedeny potřebné pojmy. Poté se pozornost věnuje problémům dokázaným pomocí počítače, speciálně na problém čtyř barev, jakožto první problém, který byl s pomocí počítače dokázán. V další části jsou popsány jednotlivé, z historického pohledu zajímavé, dokazovací systémy, jejich vzájemné porovnání z různých hledisek a využití těchto systémů.

Jako další část práce jsem implementoval jednoduchý dokazovací systém. Systém umí provádět důkazy na syntaktické úrovni pomocí rezoluční metody a také umožňuje ověřování toho, že daná formule sémanticky vyplývá ze zadané množiny formulí výrokové logiky.

Aplikace by mohla sloužit jako pomůcka pro pochopení důkazů pomocí rezoluční metody, případně pro výuku základů výrokové logiky, kde by se dalo názorně předvést sémantické vyplývání a pravdivostní ohodnocení formule.

Conclusions

The main part of the thesis focuses on computer proofs. First, the necessary concepts are introduced. Then, attention is paid to computer-proven problems, especially the four-color problem, as the first computer-proven problem. The next part describes the individual, historically interesting, proof systems and their mutual comparison from different perspectives and the use of these systems.

As another part of the work, I implemented a simple proof system. The system can perform proofs at the syntactic level using the resolution method and also allows verification that a given formula semantically follows from a given set of propositional logic formulas.

The application could serve as an aid for understanding the proofs using the resolution method, or for teaching the basics of propositional logic, where the semantic inference and truth evaluation of the formula could be clearly demonstrated.

A Predikátová a výroková logika

Části o výrokové a predikátové logice vznikly takřka doslovným přepisem z [**1**] a [**2**].

A.1 Predikátová logika

Definice 1 (Jazyk PL)
Jazyk $\mathcal{J}$ PL obsahuje (a tím je určen):

- relační proměnné $x, y, z, \ldots, x_1, x_2, \ldots$
- relační symboly $p, q, r, \ldots, p_1, p_2, \ldots$, ke každému relačnímu symbolu r je definováno nezáporné celé číslo $\sigma(r)$ nazývané arita symbolu r. Množinu relačních symbolů značíme R, tato množina nesmí být prázdná.
- funkční symboly $f, g, h, \ldots, f_1, f_2, \ldots$, ke každému funkčnímu symbolu f je definováno nezáporné celé číslo $\sigma(f)$ nazývané arita symbolu f. Množinu funkčních symbolů značíme F.
- symboly pro logické spojky $\neg$ (negace), $\Rightarrow$ (implikace), $\wedge$ (konjunkce), $\vee$ (disjunkce), $\leftrightarrow$ (ekvivalence).
- symboly pro kvantifikátory – $\forall$ (univerzální), $\exists$ (existenční).
- pomocné symboly – různé typy závorek a čárka.

Jazyk tedy může být reprezentovaný trojicí $\langle R, F, \sigma \rangle$.

Definice 2 (Term PL)
Term jazyka typu $\langle R, F, \sigma \rangle$ je induktivně definován takto:

- každá proměnná x je term,
- je-li $f \in F$ n-ární a jsou-li $t_1, \ldots, t_n$ termy, pak $f(t_1, \ldots, t_n)$ je term.

Definice 3 (Formule PL)
Formule jazyka typu $\langle R, F, \sigma \rangle$ je induktivně definována takto:

- je-li $r \in R$ n-ární a jsou-li $t_1, \ldots, t_n$ termy, pak $r(t_1, \ldots, t_n)$ je formule,
- jsou-li φ a ψ formule, pak $\neg\varphi$, $(\varphi \Rightarrow \psi)$ jsou také formule,
- je-li φ formule a x proměnná, pak $(\forall x)\varphi$ je formule.

Termy a formule jsou definovány induktivně. Každý term použitý při konstrukci termu t se nazývá podterm termu t. Každá formule použitá v konstrukci formule φ se nazývá podformule formule φ.

Říkáme, že proměnná se vyskytuje v termu nebo ve formuli, jestliže je některým symbolem termu nebo formule jakožto řetězce symbolů s tou výjimkou, že výskyty za $\forall$ se nepočítají (tj. proměnná x nemá výskyt ve formuli $(\forall x)r(y,z)$). Množinu všech proměnných, které se vyskytují v termu t označujeme $var(t)$; množinu všech proměnných, které se vyskytují ve formuli φ označujeme $var(\varphi)$.

Definice 4 (Substituce v PL – term)

Nechť t a s jsou termy, x proměnná. Výsledek substituce termu s za x v t je term $t(x/s)$ definovaný následovně:

- je-li t proměnná, pak

$$t(x/s) = \begin{cases} \text{s} & \text{pro } t = x \\ \text{t} & \text{pro } t \neq x \end{cases}$$

- pro $t = f(t_1, \ldots, t_n)$, kde $f \in F$ je n-ární a $t_1, \ldots, t_n$ jsou termy, je $t(x/s) = f(t_1(x/s), \ldots, t_n(x/s))$.

Definice 5 (Substituce v PL – formule)

Pro formuli φ, term s a proměnnou x je výsledkem substituce termu s za x ve φ formule $\varphi(x/s)$ definovaná následovně:

- pro $\varphi = r(t_1, \ldots, t_n)$, kde $r \in R$ je n-ární a $t_1, \ldots, t_n$ jsou termy, je $\varphi(x/s) = r(t_1(x/s), \ldots, t_n(x/s))$
- $(\neg\varphi)(x/s) = \neg(\varphi(x/s))$,
  $(\varphi \Rightarrow \psi)(x/s) = \varphi(x/s) \Rightarrow \psi(x/s)$
- $((\forall y)\varphi)(x/s) = (\forall y)\varphi \quad \text{pro} \quad y = x$,
  $((\forall y)\varphi)(x/s) = (\forall y)(\varphi(x/s)) \quad \text{pro} \quad y \neq x$

Definice 6 (Substituovatelný term)

Substituce termu t za proměnnou x ve formuli φ je korektní (t se nazývá substituovatelný za x ve φ), jestliže pro každou $y \in var(t)$ platí, že žádná podformule formule φ, která je ve tvaru $(\forall y)\psi$, neobsahuje výskyt x, který je volným výskytem (proměnná není vázaná kvantifikátorem) x ve φ.

Definice 7 (Obecnější substituce)

Substituce θ se nazývá obecnější než substituce σ, jestliže existuje substituce τ, pro kterou $\sigma = \theta\tau$.

Definice 8 (Unifikační substituce (unifikace))

Substituce θ se nazývá unifikační substituce (také unifikace) množiny $\{\varphi_1, \ldots, \varphi_n\}$ formulí (popř. termů), pokud $\varphi_1\theta = \varphi_2\theta = \cdots = \varphi_n\theta$.

Definice 9 (Nejobecnější unifikace)

Unifikace θ množiny $\{\varphi_1, \ldots, \varphi_n\}$ se nazývá nejobecnější unifikace (zkratka mgu z anglického „most general unifier“) této množiny, jestliže je obecnější než každá jiná unifikace množiny $\{\varphi_1, \ldots, \varphi_n\}$.

A.2 Výroková logika

Výrokovou logiku lze chápat jako podmnožinu logiky predikátové. Z toho důvodu budou definované pouze pojmy, které nebyly uvedeny v předchozí části.

Definice 10 (Pravdivostní ohodnocení)

Pravdivostní ohodnocení je libovolné zobrazení výrokových proměnných daného jazyka výrokové logiky do množiny $\{0, 1\}$, tj. libovolné zobrazení e přiřazující každé výrokové proměnné p hodnotu $e(p) \in \{0, 1\}$.

Definice 11 (Pravdivostní hodnota)

Nechť je dáno ohodnocení e. Pravdivostní hodnota formule φ při ohodnocení e, označujeme ji $\|\varphi\|_e$, je definována následovně:

- Je-li φ výrokovou proměnnou p, pak $\|p\|_e = e(p)$.
- Je-li φ složená formule, tedy má tvar $\neg\psi$ nebo $(\psi \Rightarrow \theta)$, pak
  $\|\neg\psi\|_e = 1$ pokud $\|\psi\|_e = 0$;
  $\|\neg\psi\|_e = 0$ pokud $\|\psi\|_e = 1$;
  $\|\psi \Rightarrow \theta\|_e = 1$ pokud $\|\psi\|_e = 0$ nebo $\|\theta\|_e = 1$;
  $\|\psi \Rightarrow \theta\|_e = 0$ jinak.

Je-li $\|\varphi\|_e = 1$ ($\|\varphi\|_e = 0$), říkáme, že formule φ je při ohodnocení e pravdivá (nepravdivá).

Definice 12

Formule VL se nazývá:

- tautologie, je-li při každém ohodnocení pravdivá,
- kontradikce, je-li při každém ohodnocení nepravdivá,
- splnitelná, je-li pravdivá při alespoň jednom ohodnocení.

Definice 13 (Sémantické vyplývání)

Mějme formule $\psi_1, \ldots, \psi_n$, kde $n \geq 0$. Formule φ *sémanticky plyne* z formulí $\psi_1, \ldots, \psi_n$ (značíme $\psi_1, \ldots, \psi_n \models \varphi$), jestliže $\|\varphi\|_e = 1$ pro každé pravdivostní ohodnocení e takové, že $\|\psi_1\|_e = 1, \ldots, \|\psi_n\|_e = 1$.

Definice 14 (Normální formy)

Nechť V je množina výrokových proměnných. Pak:

- literál nad V je libovolná výroková proměnná z V nebo její negace,
- klauzule nad V je konečná množina literálů (tvořících disjunkci),
- elementární konjunkce nad V je libovolná konjunkce literálů,
- elementární disjunkce nad V je libovolná disjunkce literálů,
- konjunktivní normální forma (CNF) nad V je konjunkce elementárních disjunkcí nad V,
- disjunktivní normální forma (DNF) nad V je disjunkce elementárních konjunkcí nad V.

Definice 15 (Důkaz formule)

Důkaz formule φ z množiny formulí T je libovolná posloupnost formulí $\varphi_1, \ldots, \varphi_n$ taková, že $\varphi_n = \varphi$ a každá $\varphi_i \quad (i = 1, \ldots, n)$:

- je axiomem (formule, kterou automaticky přijímáme jako platnou),
- nebo náleží do T,
- nebo vzniká z předchozích formulí důkazu pomocí odvozovacího pravidla modus ponens (toto pravidlo říká, že z formulí φ a $\varphi \Rightarrow \psi$ lze odvodit formuli ψ), tedy existují indexy $j, k < i$ tak, že φ_k je formule ve tvaru $\varphi_j \Rightarrow \varphi_i$.

Formule φ je dokazatelná z T (zapisujeme $T \vdash \varphi$), pokud existuje důkaz formule φ z T. Pokud $\vdash \varphi$, pak říkáme, že φ je dokazatelná (z prázdného systému předpokladů).

Definice 16 (Sporná množina)

Množina formulí T se nazývá sporná (nekonzistentní), jestliže je z ní dokazatelná jakákoliv formule. Není-li T sporná (tj. existuje formule, která není z T dokazatelná), nazývá se bezesporná (konzistentní).

B Obsah přiloženého CD

V této sekci je uveden stručný popis obsahu přiloženého CD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

`bin/`

Soubor run.bat pro spuštění aplikace přímo z CD na operačním systému Windows. Soubor run.sh pro spuštění aplikace přímo z CD na Unixových systémech.

`doc/`

Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

`src/`

Kompletní zdrojové kódy a další soubory potřebné pro bezproblémové spuštění programu.

`readme.txt`

Instrukce pro instalaci a spuštění programu, včetně všech požadavků pro jeho bezproblémový provoz.

Bibliografie

[1] Radim Bělohlávek, Matematická logika - poznámky k přednáškám, srpen 2006.

[2] Radim Bělohlávek, Vilém Vychodil, Diskrétní matematika pro informatiky, Olomouc 2006.

[3] Wikipedie: Type Theory. Webové stránky dostupné na: https://en.wikipedia.org/wiki/Type_theory.

[4] Wikipedie: Curry-Howard correspondence. Webové stránky dostupné na: https://en.wikipedia.org/wiki/Curry-Howard_correspondence.

[5] Portoraro, Frederic, "Automated Reasoning", The Stanford Encyclopedia of Philosophy (Spring 2019 Edition), Edward N. Zalta (ed.), https://plato.stanford.edu/archives/spr2019/entries/reasoning-automated/.

[6] Neil Robertson, Daniel P. Sanders, Paul Seymour and Robin Thomas, The Four Color Theorem. Webové stránky dostupné na: people.math.gatech.edu/t̃homas/FC/

[7] Robin Wilson, Four colours suffice. Webové stránky dostupné na: https://www.math.ku.dk/formidling/moed-math/moed-math-2014/oplaegholdere/WilsonFOURCS.pdf

[8] Four color theorem. Webové stránky dostupné na: https://en.wikipedia.org/wiki/Four_color_theorem.

[9] Will an old problem yield a new insight? Perhaps an elegant proof of the 4-colour theorem? Webové stránky dostupné na: https://medium.com/research-outreach/will-an-old-problem-yield-a-new-insight-perhaps-an-elegant-proof-of-the-4-colour-theorem-4aa65619afbf.

[10] Sipka Timothy. "Alfred Bray Kempe's 'Proof' of the Four-Color Theorem." Math Horizons, vol. 10, no. 2, 2002, pp. 21–26. JSTOR, www.jstor.org/stable/25678395. Accessed 14 Oct. 2020.

[11] Robbins Algebras Are Boolean. Webové stránky dostupné na: https://calculemus.org/MathUniversalis/4/6robbins.html.

[12] Wikipedie – Kepler conjecture. Webové stránky dostupné na: https://en.wikipedia.org/wiki/Kepler_conjecture.

[13] AZFour: Connect Four Powered by the AlphaZero Algorithm. Webové strány dostupné na: https://medium.com/@sleepsonthefloor/azfour-a-connect-four-webapp-powered-by-the-alphazero-algorithm-d0c82d6f3ae9.

[14] Wikipedie – Connect four. Webové stránky dostupné na: https://en.wikipedia.org/wiki/Connect_Four.

[15] Harrison, John & Urban, Josef & Wiedijk, Freek. (2014). History of Interactive Theorem Proving. 10.1016/B978-0-444-51624-4.50004-6.

[16] Geuvers, H. Proof assistants: History, ideas and future. Sadhana 34, 3–25 (2009).

[17] Freek Wiedijk, osobní stránky, Automath. Webové stránky dostupné na: https://www.cs.ru.nl/%7Efreek/aut/index.html

[18] PRL Project. Dostupné na: http://www.nuprl.org/Intro/intro.html

[19] The Coq Proof Assistant. Dostupné na: https://coq.inria.fr/

[20] The Twelf Project. Dostupné na: http://twelf.org/wiki/Main_Page

[21] Wikipedie: Twelf. Dostupné na: https://en.wikipedia.org/wiki/Twelf.

[22] Ulf Norell and James Chapman. Dependently Typed Programming in Agda.

[23] Welcome to Agda's Documentation. Webové stránky dostupné na: https://agda.readthedocs.io/en/v2.6.1.1/

[24] The Agda Wiki. Webové stránky dostupné na: https://wiki.portal.chalmers.se/agda/pmwiki.php

[25] Gordon, Mike. (2000). From LCF to HOL: a short history.. 169-186.

[26] HOL, Interactive Theorem Prover. Dostupné na: https://hol-theorem-prover.org/.

[27] Isabelle, proof assistant. Webové stránky dostupné na: https://isabelle.in.tum.de/.

[28] Adam Naumowicz, Artur Kornilowicz, Adam Grabowski, Mizar Hands-on Tutorial, Institute of Informatics, University of Bialystok, Poland. CICM 2016, Bia lystok, July 29, 2016. Webové stránky dostupné na: http://mizar.org/cicm_tutorial/mizar.pdf

[29] The Lexical Context of a Mizar Article. Webové stránky dostupné na: http://mizar.org/language/lexicon.html

[30] Mizar Home Page. Webové stránky dostupné na: http://mizar.org

[31] W. McCune, "Prover9 and Mace4", https://www.cs.unm.edu/ mccune/prover9/, 2005-2010.

[32] Stephan Schulz, Simon Cruanes, Petar Vukmirovic: Faster, Higher, Stronger: E 2.3, Proceedings of the 27th Conference on Automated Deduction (CADE'19), Natal, Brazlil. Volume 11716 of LNAI. Springer, 2019.

[33] The E Theorem Prover. Webové stránky dostupné na: https://wwwlehre.dhbw-stuttgart.de/ sschulz/E/E.html

[34] Vampire's Home Page. Webové stránky dostupné na: http://www.vprover.org/index.cgi

[35] The Automated Reasoning System, Nqthm. Webové stránky dostupné na: http://www.computationallogic.com/software/nqthm/

[36] ACL2 Version 8.3. Webové stránky dostupné na: https://www.cs.utexas.edu/users/moore/acl2/

[37] Prototype Verification System. Webové stránky dostupné na: http://pvs.csl.sri.com

[38] Nawaz, M. Saqib & Malik, Moin & Li, Yi & Meng, Sun & Lali, Muhammad Ikram. (2019). A Survey on Theorem Provers in Formal Methods.

[39] The TPTP Problem Library for Automated Theorem Proving. Webové stránky dostupné na: http://www.tptp.org/

[40] The QED Project . Webové stránky dostupné na: http://mizar.org/qed

[41] QED manifesto. Webové stránky dostupné na: http://www.rbjones.com/rbjpub/logic/qedres01.htm#First

[42] The Hundred Greatest Theorems. Webové stránky dostupné na: http://pirate.shu.edu/%7Ekahlnath/Top100.html.

[43] Freek Wiedijk, osobní stránky, Formalizing 100 Theorems. Webové stránky dostupné na: https://www.cs.ru.nl/ freek/100/

[44] ACL2 Version 8.3. Webové stránky dostupné na: https://www.cs.utexas.edu/users/moore/publications/how-to-prove-thms/intro-to-acl2.pdf

[45] Wikipedie: Automated Reasoning. Webové stránky dostupné na: https://en.wikipedia.org/wiki/Automated_reasoning

[46] Dr Geoff Sutcliffe. What is Automated Theorem Proving? Webové stránky dostupné z: https://www.cs.miami.edu/home/geoff/Courses/CSC545-08S/Content/WhatIsATP.shtml

[47] Dr Geoff Sutcliffe. Automated Theorem Proving. Webové stránky dostupné na: https://www.cs.miami.edu/home/geoff/Courses/TPTPSYS/FirstOrder/