VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÝ THEOREM PROVER

AUTOMATIC THEOREM PROVER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN MAZÁNEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ONDŘEJ LENGÁL, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Mazánek Martin**
Program: Informační technologie
Název: **Automatický theorem prover**
An Automatic Theorem Prover
Kategorie: Teoretická informatika

Zadání:

1. Seznamte se s principy automatizovaného dokazování teorémů a základními logickými kalkuly.
2. Nastudujte základní techniky používané v automatických theorem proverech.
3. Navrhněte architekturu nástroje pro automatické rozhodování logické platnosti formulí výrokové logiky a predikátové logiky prvního řádu.
4. Nástroj navržený v předchozím bodě implementujte. Jako vstupní formát použijte TPTP.
5. Proveďte experimentální srovnání vytvořeného nástroje a state-of-the-art automatických theorem proverů.
6. Dosažené výsledky diskutujte.

Literatura:

- Huth, Ryan. Logic in Computer Science. Cambridge University Press.
- Robinson, Voronkov. Handbook of Automated Reasoning. The MIT Press.
- Harrison. Handbook of Practical Logic and Automated Reasoning. Cambridge University Press.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1-3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Lengál Ondřej, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 31. října 2019

Abstrakt

Tato bakalářská práce se zabývá implementací systému pro automatické dokazování vět výrokové a predikátové logiky používajícím rezoluci. Cílem této práce je vytvořit jednoduchý systém a zdokumentovat jeho vývoj, nikoliv tvorba konkurenceschopného systému. Dále je v práci představeno několik populárních systémů automatického dokazování.

Abstract

This thesis focuses on implementation of resolution-based automatic theorem prover for propositional and first-order logic. The goal of this thesis is to create simple prover and document the development. Making state-of-the-art prover is not the goal of this thesis. We also present some popular automated theorem provers.

Klíčová slova

Automatické dokazování, rezoluce, výroková logika, predikátová logika prvního řádu, TPTP

Keywords

Automated Theorem Proving, Resolution, Propositional logic, First-order logic, TPTP

Citace

MAZÁNEK, Martin. *Automatický Theorem Prover*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Ondřej Lengál, Ph.D.

Automatický Theorem Prover

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ondřeje Lengála. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Mazánek
4. června 2020

Poděkování

Rád bych poděkoval vedoucímu mé práce, Ing. Ondřeji Lengálovi PhD., za vysvětlení probírané látky, rychlou komunikaci a velice přínosnou odezvu k práci během jejího psaní.

Obsah

1 Úvod **3**

2 Matematická logika **4**
2.1 Výroková logika 4
2.1.1 Sémantika výrokových formulí 5
2.1.2 Sémantické dokazování pravdivosti výroku 7
2.2 Predikátová logika prvního řádu 8
2.2.1 Sémantika predikátové logiky 1. řádu 11
2.3 Vyšší řády logiky 11

3 Způsoby dokazování **12**
3.1 Normální formy 13
3.1.1 Negační normální forma 13
3.1.2 Prenexní normální forma 13
3.1.3 Skolemova normální forma 13
3.2 Substituce a unifikace 14
3.3 Hilbertovský výrokový kalkulus 14
3.4 Hilbertovský predikátový kalkulus 15
3.5 Přirozená dedukce 15
3.6 Rezoluce 15
3.6.1 Rezoluce ve výrokové logice 16
3.6.2 Rezoluce v predikátové logice 16
3.6.3 Faktorizace 16
3.6.4 Rezoluce v predikátové logice s rovností 16
3.7 Paramodulace 17

4 Návrh systému **18**
4.1 Formát TPTP 18
4.1.1 Formát formule predikátové logiky prvního řádu 18
4.2 Kalkulus důkazového systému 20
4.3 Datový model 20
4.3.1 Reprezentace klauzulí 22
4.4 Obecná architektura systému 23
4.4.1 Převod do CNF 23
4.4.2 Unifikace 26
4.4.3 Výběr klauzule ke zpracování 28
4.4.4 Zpracování vybrané klauzule 28
4.5 Zpracování nových klauzulí 29

4.6 Odvozovací pravidla . . . 29
4.6.1 Binární rezoluce . . . 29
4.6.2 Paramodulace . . . 30
4.6.3 Faktorizace . . . 31
4.6.4 E-rezoluce . . . 31
4.7 Srovnání se state-of-the-art systémy . . . 32
4.7.1 E . . . 32
4.7.2 OTTER . . . 32
4.7.3 Vampire . . . 32
4.7.4 Shrnutí představených systémů . . . 32

5 Implementace **33**
5.1 Optimalizace . . . 33
5.1.1 Odstranění literálů . . . 33
5.1.2 Odstranění redundantních klauzulí . . . 33
5.2 Profiling . . . 34
5.3 Použití . . . 34
5.3.1 Kompilace . . . 34
5.3.2 Konfigurace . . . 35

6 Experimenty **36**
6.1 Experimenty v průběhu vývoje . . . 36
6.2 Srovnání paramodulace a rezoluce s axiomy rovnosti . . . 36
6.3 Experimenty na problémech z TPTP . . . 37
6.4 Zhodnocení experimentů . . . 38

7 Závěr **39**

Literatura **40**

A Obsah média **41**

Kapitola 1

Úvod

S postupným technologickým vývojem lidstva se vyvíjí i obor umělé inteligence. Už se nám podařilo překonat fyzickou stránku člověka, ale z pohledu té duševní ještě technologie zaostává. Tento text pojednává o specifické části inteligentních systémů, kterou je automatické dokazování matematických vět. Systémy schopné takového dokazování se nazývají automatické provery, nebo automatické dokazovače vět a lze se s nimi setkat pod zkratkou ATP z anglického Automatic Theorem Prover. Funkce takových systémů spočívá ve stavbě důkazu dané domněnky ze zadaných matematických axiomů. Výsledkem běhu automatického proveru je důkaz, který v praxi může sloužit například v matematice k hledání či ověření teorémů nebo k verifikaci software a hardware. Tématem této bakalářské práce je vytvoření jednoho takového proveru a srovnání se state-of-the-art nástroji, jako například E, Vampire nebo Otter.

Prvně ale musíme myšlenku, kterou se snažíme dokázat, řádně zformulovat. Touto formulací se zabývá druhá kapitola, ve které popisujeme matematickou nebo-li formální logiku, způsob jejího zápisu a její formy. Je zde znázorněna výroková logika a predikátová logika prvního řádu, které jsou doménou našeho proveru.

Teď máme zformalizované myšlenky, ale ty nám samy o sobě nic neřeknou o naší domněnce. Ve třetí kapitole se zabýváme způsoby, jak lze domněnku dokázat. Procházíme několik populárních způsobů dokazování včetně rezoluce, kterou budeme používat v našem proveru.

Po výběru dokazovací metody se pouštíme do návrhu samotného proveru.

Kapitola 2

Matematická logika

Při normální lidské komunikaci často používáme výrazy, které mají vágní význam a jejich pravdivost je diskutabilní a závislá na kontextu. V této kapitole si popíšeme matematickou logiku, nebo-li formální logiku, což je exaktní věda zabývající se přesným opakem — zachycením myšlenky pomocí formálního zápisu, jehož význam je pevně daný a nepopiratelný. To dělá z matematické logiky silný nástroj, kterým lze formálně uvažovat nad fakty a domněnkami.

V následujících částech představujeme výrokovou logiku a predikátovou logiku prvního řádu a nastíníme další možná rozšíření logiky. Definice v této kapitole jsou převzaty z [1, 3, 13, 12, 6].

2.1 Výroková logika

Výroková logika je základní částí matematické logiky. Jak už název napovídá, skládá se z výrokových formulí — tvrzení, která jsou buď pravdivá, nebo nepravdivá. Tato tvrzení zapisujeme pomocí proměnných $x, y...$ a pomocných symbolů — *logických spojek* ($\neg, \wedge, \vee, \rightarrow, \equiv$) a závorek.

Definice 2.1 *Výroková formule je tvrzení, které můžeme označit buď jako pravdivé, nebo jako nepravdivé.*

Definice 2.2 *Výrokový atom je výroková formule, která neobsahuje žádnou logickou spojku.*

Výrokovým atomům se někdy říká *prvotní formule.*

Definice 2.3 *Množina všech výrokových formulí je nejmenší množina výrazů konečné délky splňující následující podmínky*

- *každá proměnná je výroková formule,*
- *je-li ϕ výroková formule, pak $\neg\phi$ a (ϕ) jsou výrokové formule,*
- *jsou-li ϕ a ψ výrokové formule, pak $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$ a $\phi \equiv \psi$ jsou výrokové formule.*

2.1.1 Sémantika výrokových formulí

Výroková logika tvoří formule spojováním výrokových *atomů* do složených výroků pomocí *logických spojek*. Například výrok *Venku svítí.* je *atom*, protože nelze dále dělit. Pojmenujme si tento výrok jako A. Tento výrok může nabývat dvou hodnot, může být buď pravdivý (značíme číslicí 1), nebo nepravdivý (značíme číslicí 0). Dalším příkladem atomického výroku je například *Venku je teplo.* Označme si ho jako B. Toto byly příklady výrokových *atomů*. Složitější výroky tvoříme spojováním *atomů* pomocí *logických spojek*.

- spojkou $\neg$ značíme *negaci* výroku. V přirozeném jazyce jí odpovídá spojení *Není pravda, že....* Formule $\neg A$ je ekvivalentní tvrzení "Venku nesvítí."
- spojkou $\wedge$ značíme *konjunkci* dvou výroků. V přirozeném jazyce jí odpovídá slovo *a*. Formule $A \wedge B$ je ekvivalentní tvrzení *Venku svítí a je tam teplo.*
- spojkou $\vee$ značíme *disjunkci* dvou výroků. V přirozeném jazyce jí odpovídá slovo *nebo*. Formule $A \vee B$ je ekvivalentní tvrzení *Venku svítí nebo je teplo.* Toto spojení není výlučné, proto by bylo pravdivé i kdyby zároveň svítilo a bylo teplo.
- spojkou $\rightarrow$ značíme *implikaci*. V přirozeném jazyce jí odpovídá spojení *Pokud ..., tak ....* Formule $A \rightarrow B$ je ekvivalentní tvrzení *Pokud venku svítí, tak je venku teplo.*
- spojkou $\equiv$ značíme *ekvivalenci*. V přirozeném jazyce jí odpovídá spojení *... právě tehdy, když ....* Formule $A \equiv B$ je ekvivalentní tvrzení *Venku svítí právě tehdy, když je tam teplo.*

Tabulka 2.1: Tabulka pravdivostních hodnot logických spojek.

X	Y	$\neg X$	$\neg Y$	$X \wedge Y$	$X \vee Y$	$X \rightarrow Y$	$X \equiv Y$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

Tabulka 2.2: Tabulka precedence logických spojek

	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\equiv$
precedence[1]	1	2	3	4	5

[1] Nižší hodnota znamená těsnější spojku.

Definice 2.4 *Pravdivostní ohodnocení (interpretace) je každá funkce υ z množiny všech výrokových formulí do množiny $\{0,1\}$, která pro libovolné formule ϕ a ψ splňuje podmínky*

- $\upsilon(\phi \wedge \psi) = 1$, *právě když* $\upsilon(\phi) = 1$ *a* $\upsilon(\psi) = 1$,
- $\upsilon(\phi \vee \psi) = 1$, *právě když* $\upsilon(\phi) = 1$ *nebo* $\upsilon(\psi) = 1$,
- $\upsilon(\phi \to \psi) = 1$, *právě když* $\upsilon(\phi) = 0$ *nebo* $\upsilon(\psi) = 1$,
- $\upsilon(\phi \equiv \psi) = 1$, *právě když* $\upsilon(\phi) = \upsilon(\psi)$,
- $\upsilon(\neg\phi) = 1$, *právě když* $\upsilon(\phi) = 0$.

Zápis $\upsilon(\phi) = 1$ *čteme* formule ϕ je splněna (pravdivostním) ohodnocením υ *nebo* (ohodnocení) υ splňuje formuli ϕ. *Místo* $\upsilon(\phi) = 1$ *se píše také* $\upsilon \models \phi$.

Definice 2.5 *Řekněme, že* výroková formule ϕ *je* splnitelná, *jestliže existuje pravdivostní ohodnocení* υ *takové, že* $\upsilon(\phi) = 1$. *Formule* ϕ *je* tautologie ($\top$), *jestliže* $\upsilon(\phi) = 1$ *pro každé pravdivostní ohodnocení* υ. *Množinu všech splnitelných výrokových formulí a množinu všech tautologií značíme* SAT, *resp.* TAUT.

Definice 2.6 *Formule* ϕ *je* kontradikce ($\bot$), *jestliže* $\upsilon(\phi) = 0$ *pro každé pravdivostní ohodnocení* υ.

Definice 2.7 *Řekněme, že výroková formule* ϕ *je (tautologickým)* důsledkem *množiny formulí* T *nebo že* ϕ *vyplývá z* T, *a píšeme* $T \models \phi$, *jestliže* ϕ *má pravdivostní hodnotu* 1 *při každém pravdivostním ohodnocení* υ, *které přiřazuje hodnotu* 1 *všem formulím v* T. *O množině* T *v této souvislosti mluvíme jako o množině* předpokladů *nebo o množině* axiomů. *Formule* ϕ *je důsledkem formule* ψ, *jestliže* $\psi \models \phi$. *Formule* ϕ *a* ψ *jsou* ekvivalentní, *jestliže* ϕ *je důsledkem* ψ *a zároveň* ψ *je důsledkem* ϕ.

Pro všechny formule výrokové logiky platí následující logické ekvivalence:
Zákon ekvivalence

$$A \equiv B \iff (A \to B) \wedge (B \to A)$$

Zákon implikace

$$A \to B \iff \neg A \vee B$$

Komutativní zákony

$$A \wedge B \iff B \wedge A$$

$$A \vee B \iff B \vee A$$

Asociativní zákony

$$A \vee (B \vee C) \iff (A \vee B) \vee C$$

$$A \wedge (B \wedge C) \iff (A \wedge B) \wedge C$$

Distributivní zákony

$$A \vee (B \wedge C) \iff (A \vee B) \wedge (A \vee C)$$

$$A \wedge (B \vee C) \iff (A \wedge B) \vee (A \wedge C)$$

Zákony zjednodušení disjunkce

$$A \vee \top \iff \top$$

$$A \vee \bot \iff A$$

$$A \vee A \iff A$$

$$A \vee (A \wedge B) \iff A$$

Zákony zjednodušení konjunkce

$$A \wedge \top \iff A$$

$$A \wedge \bot \iff \bot$$

$$A \wedge A \iff A$$

$$A \wedge (A \vee B) \iff A$$

Zákon o vyloučení třetího

$$A \vee \neg A \iff \top$$

Zákony sporu

$$\neg(A \vee \neg A) \iff \bot$$

$$A \wedge \neg A \iff \bot$$

De Morganovy zákony

$$\neg(A \wedge B) \iff \neg A \vee \neg B$$

$$\neg(A \vee B) \iff \neg A \wedge \neg B$$

Zákon dvojité negace

$$\neg\neg A \iff A$$

Eliminace konjunkce

$$A_1 \wedge A_2 \wedge ... \wedge A_n \to A_i$$

$$i \in \{1, ..., n\}$$

Zavedení disjunkce

$$A_i \to A_1 \vee A_2 \vee ... \vee A_n$$

$$i \in \{1, ..., n\}$$

2.1.2 Sémantické dokazování pravdivosti výroku

Definovali jsme si, co je to tautologie. Chceme-li určit, zda-li je nějaká formule tautologie, stačí probrat všechny funkce z F do $0, 1$, kde F je konečná množina všech výrokových atomů, které se vyskytují v dané formuli. Jako příklad si vezměme formuli $\phi = \neg(A \vee B \to A \wedge C) \to (C \to B)$. Z formule můžeme vyčíst, že $F = A, B, C$ – formule ϕ má tedy 3 výrokové atomy. Každému atomu náleží hodnota z $0, 1$, tudíž máme 2^3 možných kombinací. Obecný vzorec je 2^n kombinací. Vytvoříme si tabulku a můžeme vyhodnocovat jednotlivé *interpretace* formule ϕ.

Tabulka 2.3: Tabulková metoda důkazu tautologie

A	B	C	$A \vee B$	$A \wedge C$	$C \to B$	$A \vee B \to A \wedge C$	$\neg(A \vee B \to A \wedge C) \to (C \to B)$
0	0	0	0	0	1	1	1
0	0	1	0	0	0	1	1
0	1	0	1	0	1	0	1
0	1	1	1	0	1	0	1
1	0	0	1	0	1	0	1
1	0	1	1	1	0	1	1
1	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1

Z tabulky2.3 můžeme vidět, že všechna pravdivostní ohodnocení splňují formuli ϕ. Sémanticky jsme tak dokázali, že se jedná o *tautologii*.

Tato metoda se nazývá *tabulková metoda* a pomocí ní lze zjistit, zda-li je daná formule *splnitelná*, *tautologie*, nebo *kontradikce*.

Tato metoda pro většinu případů značně nepraktická, jelikož s počtem atomů stoupá její náročnost exponenciálně. V dalších kapitolách probereme, jak lze tento problém vyřešit použitím syntaktického dokazování.

2.2 Predikátová logika prvního řádu

Výroková logika je užitečný nástroj, ale pro komplexnější problémy, jako je například aritmetika není dostatečně expresivní. Aritmetický výraz, jako například $x > y$, není ani pravdivý, ani nepravdivý. Jeho pravdivost závisí na hodnotách x a y, či, více formálně, operátor $>$ je relace, která mapuje dvojici čísel na pravdivostní hodnotu. Pro práci s takovými výrazy tedy potřebujeme expresivnější jazyk.

Predikátová logika prvního řádu (dále jen predikátová logika) je rozšíření výrokové logiky o predikáty. V predikátové logice pracujeme s objekty, kterým pomocí *predikátových symbolů* přiřazujeme nějaké vlastnosti. Pokud bychom chtěli ve výrokové logice zapsat fakt, že *Bob* je otcem *Alice*, použili bychom nějaký atom q, který by měl hodnotu 1, pokud by opravdu *Bob* byl otcem *Alice*. Tato forma zápisu je poněkud neexpresivní, protože samotný zápis q nám o zaznamenaném faktu nic neříká. Predikátová logika nám umožňuje takové tvrzení zapsat mnohem expresivněji. Vezměme si binární predikátový symbol $otec(x, y)$, který nám udává, že y je otcem x. Dále mějme dvě konstanty a, b, reprezentující Alici a Boba. Potom formule $otec(a, b)$ znamená, že b je otcem a.

V predikátových formulích se mohou vyskytovat symboly různého druhu:

- *Logické spojky*: $\neg$, $\wedge$, $\vee$, $\to$, $\equiv$.
- *Kvantifikátory*: univerzální kvantifikátor $\forall$, existenciální kvantifikátor $\exists$.
- *Závorky*: ( ).
- *Proměnné*: $x, y, z, u, v, ..., x_0, x_1...$
- *Funkční symboly* pro označení operací s objekty. Každému funkčnímu symbolu F je přiřazeno přirozené číslo $n \geq 0$ zvané *četnost* symbolu F. Například + zpravidla označuje binární funkční symbol, tj. funkční symbol četnosti 2. Speciálním případem jsou funkční symboly s $n = 0$, které se nazývají *konstanty*.

- *Predikátové symboly* (*relační symboly*) pro označení vztahů mezi objekty. Každému predikátovému symbolu P je přiřazeno přirozené číslo $n \geq 0$ zvané *četnost* symbolu P. Například $\in$ zpravidla označuje binární predikátový symbol, tj. predikátový symbol četnosti 2. Speciálním případem jsou predikátové symboly s $n = 0$, které se nazývají *výrokové proměnné*.

Tabulka 2.4: Tabulka precedence predikátové logiky

	$\forall, \exists$	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\equiv$
precedence[2]	0	1	2	3	4	5

Množinu všech proměnných označíme Var a předpokládejme o ní, že je nekonečná spočetná. Logickým spojkám a kvantifikátorům se hromadně říká *logické symboly*. Funkční a predikátové symboly se dohromady nazývají *mimologické symboly*. Kdykoliv budeme mluvit o formulích, budeme předpokládat, že nejprve byla pevně zvolena nebo zadána množina L mimologických symbolů zvaná *jazyk*.

Definice 2.8 *Jazyk je tedy množina L mimologických symbolů spolu s údajem, který pro každý prvek množiny L určuje, zda je to funkční nebo predikátový symbol a jaká je jeho četnost.*

Důležitý je také symbol rovnosti $=$. Prozatím ho mějme za predikátový symbol.

Definice 2.9 *Množina všech* termů *jazyka L je nejmenší množina výrazů splňující podmínky:*

- *každá proměnná je term jazyka L,*
- *jsou-li $t_1, ..., t_n$ termy a $F \in L$ je funkční symbol četnosti n, pak $F(t_1, ..., t_n)$ je term jazyka L.*

Definice 2.10 *Atomická formule jazyka L je každý výraz tvaru $P(t_1, ..., t_n)$, kde $t_1, ..., t_n$ jsou* termy *jazyka L a $P \in L$ je* predikátový symbol *četnosti n.*

Definice 2.11 *Množina všech predikátových formulí jazyka L je nejmenší množina splňující podmínky:*

- *každá atomická formule je formule jazyka L,*
- *jsou-li ϕ a ψ formule jazyka L a x je proměnná, pak i výrazy $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \equiv \psi)$, $\neg\phi$, $\forall x\phi$ a $\exists x\phi$ jsou formule jazyka L.*

Definice 2.12 *Literál je atomická formule, nebo její negace.*

[2] Nižší hodnota znamená těsnější spojku.

Definice 2.13 *Interpretací I jazyka predikátové logiky se rozumí:*
neprázdná množina D_I (obor interpretace — univerzum — doména) a přiřazení α_I:

- *přiřazení pevně zvoleného prvku z D_I každé individuové konstantě,*
- *přiřazení libovolného prvku z D_I každé individuové proměnné,*
- *přiřazení n-ární operace každému n-místnému funkčnímu symbolu ($D_I^n \to D_I$),*
- *přiřazení n-ární relace každému n-místnému predikátovému symbolu ($D_I^n \to \{0,1\}$).*

Definice 2.14 *Každý výskyt libovolné proměnné v atomické formuli je* volný*. Každý* volný *(*vázaný*) výskyt proměnné x ve formuli ϕ a ve formuli ψ je zároveň* volným *(*vázaným*) výskytem ve formulích $(\phi \to \psi)$, $(\phi \equiv \psi)$, $(\phi \wedge \psi)$ a $(\phi \vee \psi)$. Každý* volný *(*vázaný*) výskyt proměnné x ve formuli ϕ je zároveň* volným *(*vázaným*) výskytem proměnné x ve formuli $\neg\phi$. Všechny výskyty proměnné x ve formulích $\forall x\phi$ a $\exists x\phi$ jsou* vázané*, žádný z nich není* volný*. Je-li proměnná y různá od proměnné x, pak každý* volný *(*vázaný*) výskyt proměnné y ve formuli ϕ je zároveň* volným *(*vázaným*) výskytem proměnné y ve formulích $\forall x\phi$ a $\exists x\phi$.*

Definice 2.15 Term *je* uzavřený*m jestliže neobsahuje žádné proměnné. Formule ϕ je* uzavřená formule *neboli* sentence*, jestliže ϕ neobsahuje* volné *výskyty proměnných. Formule ϕ je* otevřená*, jestliže neobsahuje kvantifikátory.*

Příklad 2.1 *Uvažujme formuli $\phi = \exists x(x < y \wedge \forall y(z + S(y) \neq x))$. Proměnná z má ve ϕ jediný výskyt, který je volným výskytem (nespadá pod žádný kvantifikátor pro z). Všechny tři výskyty proměnné x jsou vázané a ze tří výskytů proměnné y je první výskyt volný a zbývající dva jsou vázané. Formule ϕ není sentence, formule $\forall y\forall z\phi$ je sentence.*

Formule $otec(x, y)$ kde x, y jsou proměnné je otevřená, protože neobsahuje kvantifikátory. Formule $\forall x\forall y(otec(x, y)) \to rodic(x, y)$ kde $otec, rodic$ jsou predikátové symboly je sentence, protože neobsahuje volné proměnné. Formule $otec(Alice, Bob)$ kde $Alice, Bob$ jsou konstanty a $otec$ je predikátový symbol je otevřená sentence, protože neobsahuje kvantifikátory a neobsahuje volné výskyty proměnných.

Pro všechny formule predikátové logiky platí pravidla výrokové logiky, rozšířená o následující pravidla:

Zákon přesunu kvantifikátorů doleva (x se nevyskytuje v B!)

$$Qx(A) \vee B \iff Qx(A \vee B)$$
$$Qx(A) \wedge B \iff Qx(A \wedge B)$$

Zákon přesunu negace dovnitř

$$\neg(\forall x(A)) \iff \exists x(\neg A)$$
$$\neg(\exists x(A)) \iff \forall x(\neg A))$$

Distributivní zákony pro kvantifikátory

$$\forall x(A) \wedge \forall x(B) \iff \forall x(A \wedge B)$$
$$\exists x(A) \vee \exists x(B) \iff \exists x(A \vee B)$$

Zákony přejmenování vázaných proměnných

$$\forall x(A(x)) \vee \forall x(B(x)) \iff \forall x\forall y(A(x) \vee B(y))$$
$$\exists x(A(x)) \wedge \exists x(B(x)) \iff \exists x\exists y(A(x) \wedge B(y))$$

2.2.1 Sémantika predikátové logiky 1. řádu

Pro logické spojky negace, konjunkce, disjunkce, implikace a ekvivalence platí stejná pravidla, jako ve výrokové logice.

Pro kvantifikátory platí následující pravidla: Nechť $I \triangleleft \{x \mapsto v\}$ je interpretace vzniklá nahrazením mapování $x \mapsto ...$ za $x \mapsto v$ v interpretaci I

- $I \models \forall x\phi$ právě tehdy, když pro všechny v z domény I platí $I \triangleleft \{x \mapsto v\} \models \phi$
- $I \models \exists x\phi$ právě tehdy, když pro nějaké v z domény I platí $I \triangleleft \{x \mapsto v\} \models \phi$

Pro sémantiku termů si rekurzivně definujeme rovnost:

$$\alpha_I[f(t_1, ..., t_n)] \triangleq \alpha_I[f](\alpha_I[t_1], ..., \alpha_I[t_n])$$

Pak $I \models p(t_1, ..., t_n)$ právě tehdy, když $\alpha_I[p](\alpha_I[t_1], ..., \alpha_I[t_n])$

Definice 2.16 *Model formule predikátové logiky ϕ je interpretace I taková, že platí $I \models \phi$*

Definice 2.17 *Formule predikátové logiky ϕ je splnitelná, pokud má* model.

Definice 2.18 *Formule predikátové logiky ϕ je platná, pokud pro pokud je splnitelná ve všech interpretacích daného jazyka. Platnou formuli značíme $\models \phi$*

2.3 Vyšší řády logiky

Mimo výrokovou logiku (VL) a predikátovou logiku prvního řádu (PL^1) existují expresivnější logiky vyšších řádů (PL^n) kde $n \geq 2$. Takové logiky nám pak umožňují kvantifikaci nejen proměnných, ale i predikátů a funkcí. Tyto logiky více zkoumat nebudeme, ale určitě stojí za zmínku.

Kapitola 3

Způsoby dokazování

V předchozí kapitole jsme si u výrokové logiky uvedli příklad, jak sémanticky dokázat, že je daná formule tautologie. Použitou metodu jsme nazvali *tabulková metoda*. Případů, kdy lze takovým stylem ověřit formuli je ale hodně málo. V praxi se setkáváme s větším množstvím proměnných a množství kombinací díky exponenciálnímu růstu roste příliš rychle na to, aby se daná metoda dala aplikovat. Další omezení je fakt, že touto metodou můžeme dokazovat pouze formule výrokové logiky. V neposlední řadě tímto způsobem důkazu přicházíme o postup, jakými kroky dedukovat naši formuli. Tato kapitola čerpá z [13].

Tautologie lze ale dokázat i *syntakticky*, totiž jako formule, které lze odvodit mechanickou aplikací jistých *odvozovacích pravidel*. Tautologie je přesně ta formule, které lze formálně dokázat aplikací pravidel jistého důkazového systému neboli *kalkulu*.

Odvozovací pravidlo může vypadat například takto: $X1 : \psi \to \phi, \neg\psi \to \phi \ / \ \phi$.

Toto pravidlo umožňuje prohlásit za odvozenou (formálně dokázanou) formuli ϕ, kdykoliv se nám pro libovolnou formuli ψ podařilo (nezávisle n a sobě) dokázat formule $\psi \to \phi$ a $\neg\psi \to \phi$. Pravidlo X1 je pravidlo se dvěma předpoklady a je aplikovatelné teprve poté, kdy byly odvozeny alespoň dvě formule. Pravidla mohou mít 0 a více předpokladů. Pravidla bez předpokladů nazýváme *axiomy*.

Kalkulus tedy chápeme jako množinu odvozovacích pravidel. Kalkulus je *korektní*, jestliže neumožňuje dokázat žádnou formuli, která vzhledem k sémantice není pravdivá. Kalkulus je *úplný*, jestliže je korektní a každá tautologie v něm je dokazatelná.

Definice 3.1 *Důkaz je posloupnost formulí z předpokladů Δ, jestliže každý člen je prvkem množiny Δ, nebo je logickým axiomem kalkulu pro predikátovou logiku, nebo je odvozen z předchozích členů pomocí některého odvozovacího pravidla příslušného kalkulu. Fakt, že je formule ϕ dokazatelná z množiny předpokladů T, zapisujeme jako $T \vdash \phi$. Je-li $T = \emptyset$, píšeme jen $\vdash \phi$.*

Definice 3.2 *Kalkulus je korektní, pokud je každá dokazatelná formule pravdivá. [1].*

Definice 3.3 *Kalkulus je úplný, pokud je každá pravdivá formule dokazatelná. [1].*

3.1 Normální formy

Normální formy logických formulí jsou logické formule, které mají jistou syntaktickou strukturu. Formule zapsané takovým způsobem jsou užitečné pro jejich použití s různými kalkuly, které fungují za předpokladu, že jsou formule zapsány v normalizovaném stavu.

3.1.1 Negační normální forma

Definice 3.4 *Formule ϕ je v* negační normální formě *v případě, že se v ní nachází negace pouze před predikáty a že neobsahuje jiné logické spojky než* negace, konjunkce *a* disjunkce.

Formuli ϕ můžeme převést do negační normální formy pomocí zákonu ekvivalence, zákonu implikace, De Morganových zákonů a zákonu dvojí negace.

3.1.2 Prenexní normální forma

Definice 3.5 *Formule ϕ je v* prenexní normální formě *v případě, že má tvar*

$$\phi \equiv Q_1 x_1 Q_2 x_2 ... Q_n x_n (M)$$

, kde Q_n je buď univerzální nebo existenční kvantifikátor, x_n jsou navzájem různé proměnné a M je otevřená formule vzniklá z atomických formulí použitím logických spojek. Formuli M se pak říká jádro *(matice) a posloupnosti kvantifikátorů $Q_1 x_1 Q_2 x_2 ... Q_n x_n$ se říká* prefix *formule ϕ.*

Formuli ϕ můžeme převést do prenexní normální formy pomocí zákonů predikátové logiky (zákonů výrokové logiky doplněných o zákony pro práci s kvantifikátory). Postup převádění do prenexní normální formy:

1. eliminace logických spojek implikace a ekvivalence (zákon ekvivalence a zákon implikace)
2. přesunutí negací směrem k literálům pomocí zákonu dvojité negace a De Morganových zákonů
3. přesunutí kvantifikátorů na začátek formule pomocí zákonů o přesunutí kvantifikátorů doleva, distributivních kvantifikátorových zákonů a zákonů pro přejmenování vázaných proměnných
4. převedení matice do konjunktivní normální formy s použitím zbylých zákonů

3.1.3 Skolemova normální forma

Formule ϕ je ve *Skolemově normální formě* (konjuktivní normální formě bez kvantifikátorů), pokud je v prenexní konjunktivní normální formě a neobsahuje existenční kvantifikátory.

Převod formule do Skolemovy normální formy se nazývá *skolemizace*.

Postup skolemizace:

1. Nechť $\exists x_1$ je prvním kvantifikátorem v prefixu formule ϕ zleva. Pak se vybere libovolná konstanta c, která se v matici ještě nevyskytuje, takzvaná Skolemova konstanta, a touto konstantou se nahradí všechny výskyty proměnné x_1 v matici M a kvantifikátor $\exists x_1$ se z prefixu odstraní.

2. Nechť $\exists x_{m+1}$ je prvním existenčním kvantifikátorem v prefixu formule zleva a nechť se před ním zleva nachází m univerzálních kvantifikátorů. Pak se do jazyka přidá nová m-místná funkce $f(x_1, ..., x_m)$, takzvaná Skolemova funkce, která se dosud v matici M nevyskytuje, a její instanciancí s proměnnými $x_1, ..., x_m$ se nahradí všechny výskyty proměnné x_{m+1} v matici M a kvantifikátor $\exists x_{m+1}$ se z prefixu odstraní.

Zpřísněním syntaxe formule převedením do *Skolemovy normální formy* umožňujeme využít určitých odvozovacích pravidel pro dokazování.

3.2 Substituce a unifikace

Predikátová logika pracuje s proměnnými. Proces dosazení hodnoty do proměnné se nazývá *substituce*.

Definice 3.6 *Substituce σ je konečná množina dvojic $t_1/x_1, t_2/x_2, ..., t_n/x_n$, kde t_i jsou* termy *a x_i jsou* proměnné. Instancí *formule ϕ je formule ϕ_σ, která se získá nahrazením všech* volných *výskytů proměnných x_i termy t_i.*

Příklad 3.1 *Nechť je formule $\phi \triangleq P(x) \wedge \forall x(A(x) \vee B(y))$ a nechť je substituce $\sigma = f(a)/x$, pak instance formule $\phi_\sigma \triangleq P(f(a)) \wedge \forall x(A(x) \vee B(y))$.*

Definice 3.7 *Unifikátorem množiny literálů $\{L_1, L_2, ..., L_n\}$ je substituce σ, pro kterou platí $L_{1\sigma} = L_{2\sigma} = ... = L_{n\sigma}$. Jestliže unifikace existuje, pak se množina nazývá unifikovatelnou.*

Definice 3.8 *Unifikátor σ množiny literálů M je největší společný unifikátor M, pokud pro každý unifikátor λ nožiny literálů M existuje unifikátor τ takový, že $M\sigma\tau = M\lambda$.*

3.3 Hilbertovský výrokový kalkulus

Hilbertovský výrokový kalkulus patří mezi základní logické kalkuly. Je *korektní* a *úplný* [1]. Skládá se z jednoho odvozovacího pravidla zvaného *modus ponens* (eliminace implikace) a axiomatických schémat. Tato schémata mohou nabývat různých podob, podmínkou ale je, že z logických spojek může být použita jen *negace* $\neg$ a *implikace* $\rightarrow$ a musí být dodržena *korektnost* a *úplnost*. Uvedené příklady pochází z odborné literatury [1].

Definice 3.9 *Axiomatické schéma můžeme chápat jako nekonečnou množinu axiomů, které sdílejí jednotný tvar. Slouží k vygenerování nové formule (axiomu) z množiny libovolných formulí. Každá formule vygenerovaná pomocí axiomatického schématu je axiom.*

Odvozovací pravidlo:

- MP: ϕ, $\phi \rightarrow \psi$ / ψ

Axiomatická schémata:

- A1: $\phi \rightarrow (\psi \rightarrow \phi)$,
- A2: $(\phi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \chi))$,
- A3: $(\neg\psi \rightarrow \neg\phi) \rightarrow (\phi \rightarrow \psi)$,

Příklad 3.2 *Pojďme vytvořit důkaz pro libovolnou formuli A že $\vdash A \to A$.*

1. $\vdash A \to ((A \to A) \to A) \to ((A \to (A \to A)) \to (A \to A))$ *A2*
2. $\vdash A \to ((A \to A) \to A)$ *A1*
3. $\vdash (A \to (A \to A)) \to (A \to A)$ *MP 1,2*
4. $\vdash A \to (A \to A)$ *A1*
5. $\vdash A \to A$ *MP 3,4*

3.4 Hilbertovský predikátový kalkulus

Hilbertovský kalkulus pro predikátovou logiku prvního řádu získáme tak, že k výrokovému kalkulu přidáme dvě axiomatická schémata pro přidávání univerzálního kvantifikátoru a pravidlo *generalizace*.

- A4: $\forall x \phi(x) \to \phi(t)$,
- A5: $\forall x(\phi \to \psi(x)) \to (\phi \to \forall x \psi(x))$,
- Gen: $\phi \ / \ \forall x \phi$,

Kde v případě axiomatických schémat A4 a A5 je ϕ formule, ve které je term t substituovatelný za proměnnou x, a v případě pravidla generalizace je ϕ formule, která neobsahuje volné výskyty proměnné x.

3.5 Přirozená dedukce

Přirozená dedukce je v podstatě obrácený Hilbertovský kalkulus. Hilbertovský kalkulus se skládá z jednoho respektive dvou odvozovacích pravidel (modus ponens a generalizace) a spousty axiomatických schémat, přirozená dedukce využívá převážně odvozovací pravidla a málo či žádné axiomatické schéma. Tím si přirozená dedukce zasloužila slovo "přirozená"– když přirozeně uvažujeme, tak uvažujeme na základě faktů a uvažováním tato fakta kombinujeme v nové poznatky. Stejně tak fungují odvozovací pravidla. Důkaz přirozené dedukce je tak snadněji pochopitelný, i když jeho komplexita důvodem jednoduchých odvozovacích pravidel je pořád velká.

3.6 Rezoluce

Představili jsme si Hilbertovský kalkulus jako způsob, kterým lze dedukovat formule. Důkaz pomocí Hilbertovského kalkulu je ale značně složitý a nicneříkající. Zpravidla má takový důkaz hodně kroků a formule dosazované do axiomatických schémat nejdou intuitivně odvodit. Mechanizace dokazování je vzhledem k velkému kroku pro dokázání i jednoduchých formulí velice náročná, jelikož stavový prostor takového důkazu roste velmi rychle.

Rezoluce[5] patří mezi *refutační* metody, které vytvářejí důkaz sporem. V případě, že domněnka není důsledkem axiomů, může algoritmus rezoluce běžet nekonečně dlouho, jelikož nikdy nedojde k vyhodnocení kontradikce. Rezoluce používá jedno odvozovací pravidlo, pracující s formulemi v konjunktivní normální formě bez kvantifikátorů.

3.6.1 Rezoluce ve výrokové logice

Příklad 3.3 *Nechť existuje množina axiomů* $\Gamma = \{(A \to B) \vee C, \neg(A \wedge D)\}$ *a nechť existuje domněnka* $\phi = A \wedge B$*. Pokud platí, že* $\Gamma \models \phi$*, tak podle principu rezoluce platí* $\Gamma \cup \neg\phi \vdash_R \bot$*.*

3.6.2 Rezoluce v predikátové logice

Pro náš systém si představíme odvozovací pravidlo *binární rezoluce*. Nutno podotknout, že slovo binární v tomto případě neznačí, že používáme dvě klauzule, ale že spojujeme klauzule podle dvou literálů — z každé klauzule jeden. Při rezoluci předpokládáme, že jsou formule v klauzulární normální formě.

Rezoluční pravidlo má pro predikátovou logiku následující tvar:

$$\frac{\Gamma_1 \cup L_1 \qquad \Gamma_2 \cup L_2}{(\Gamma_1 \cup \Gamma_2)\sigma}$$

kde σ je nejobecnějším unifikátorem L_1 a $\neg L_2$.

3.6.3 Faktorizace

Pokud používáme binární rezoluci jako jediné odvozovací pravidlo, tak může nastat situace, kdy je dosažení kontradikce u pravdivé domněnky nemožné. To proto, že binární rezoluce spojuje dvě klauzule pomocí právě jednoho literálu z každé z nich, tudíž pokud tvoříme rezolventu z klauzulí o n a m literálech, tak má výsledná rezolventa právě $n + m - 2$ literálů. Pokud bychom používali rezoluci na klauzulích o dvou literálech, nikdy bychom nemohli odvodit prázdnou klauzuli aneb *kontradikci*, neboť by všechny rezolventy měly právě 2 literály. Z tohoto důvodu zavádíme odvozovací pravidlo *faktorizace*, které unifikuje dva literály v jedné klauzuli. Odvozovací systém s pravidly *rezoluce* a *faktorizace* je *úplný*[2].

Odvozovací pravidlo *faktorizace*:

$$\frac{C \vee L_1 \vee L_2}{(C \vee L)\sigma}$$

kde σ je nejobecnějším unifikátorem L_1 a L_2.

3.6.4 Rezoluce v predikátové logice s rovností

Do teď jsme předpokládali zpracování rovnosti = jako regulérního predikátového symbolu. Na rozdíl od ostatních predikátů má ale rovnost stejný význam, nezávisle na modelu. Sémantický význam formule $a = b$ je, že objekt značený a je ten stejný, co objekt b. V Našem dokazovacím systému tedy můžeme počítat s vlastnostmi rovnosti - reflexivitou, symetrií, tranzitivitou a monotónností.

Vlastnosti rovnosti můžeme aplikovat pomocí následujících axiomů [8]:

- Reflexivita: $x = x$
- Symetrie: $x = y \to y = x$
- Tranzitivita: $x = y \wedge y = z \to x = z$
- Monotónnost funkcí $x_1 = y_1 \wedge x_2 = y_2 \wedge ... \wedge x_n = y_n \to f(x_1, x_2, ..., x_n) = f(y_1, y_2, ..., y_n)$, kde f je funkční symbol

- Monotónnost predikátů $x_1 = y_1 \wedge x_2 = y_2 \wedge ... \wedge x_n = y_n \wedge P(x_1, x_2, ..., x_n) \rightarrow P(y_1, y_2, ..., y_n)$, kde P je predikátový symbol

Lze si povšimnout, že pravidla monotónnosti jsou axiomatická schémata, která generují jeden axiom pro každý funkční, respektive predikátový symbol. To může způsobit generování velkého množství axiomů a výrazně zkomplikovat řešení.

3.7 Paramodulace

Generování velkého množství často zbytečných axiomů použitím axiomů pro rovnost v predikátové logice vedlo k vymyšlení efektivnějších metod. Jednou z nich je *paramodulace* [8], která v sobě implicitně obsahuje axiomy rovnosti.

Odvozovací pravidlo *paramodulace*:

$$\frac{C \vee s = t \qquad D}{(C \vee D[t]_p)\sigma}$$

kde $D|p$ je subtermem D na pozici p, σ je nejobecnější unifikátor s a $D|p$ a $D[t]_p$ značí výsledek nahrazení subtermu D na pozici p termem t

Kapitola 4

Návrh systému

V této kapitole bude na základě vědomostí z předchozích kapitol prezentován návrh automatického theorem proveru. Jako hlavní dedukční systém jsme vybrali rezoluci, pro zefektivnění rozhodování s rovností je použita paramodulace. V prvé řadě je ale potřeba určit obecnou architekturu systému, tvary datových struktur a tvar rozhraní systému. Systém bude popsán postupně od nejvyššího stupně abstrakce po detailní zachycení jistých částí. Návrh systému probíhal po průzkumu existujících řešení a částečně je jimi inspirován. Největší inspirací byl Otter a jeho nástupce Prover9, zejména k vytvoření hlavní smyčky proveru. Zde je prezentována finální verze návrhu, která se v průběhu implementace vyvíjela z důvodu průběžných komplikací týkajících se optimalizace a celkového fungování systému.

4.1 Formát TPTP

Jako vstupní a výstupní formát systému je zvolen formát *TPTP* [11] . Jedná se o populární formát zápisu logických formulí pro automatické theorem provery, který vytvořil Geoff Sutcliffe. TPTP je zkratka pro *Thousands of Problems for Theorem Provers* (tisíce problémů pro theorem provery) a mimo definici společného formátu také obsahuje velkou knihovnu problémů různých syntaktických a sémantických významů. Problémy z této knihovny budou použity pro provádění experimentů s výsledným systémem. Formát *TPTP* podporuje mimo formulí logiky prvního řádu také formule logiky vyšších řádů a formule teorie typů.

Problémy knihovny *TPTP* obsahují také metadata o složitosti problému a typech a počtech formulí, která mohou být použita k lepší volbě strategie řešení. Tento systém tato metadata nevyužívá.

4.1.1 Formát formule predikátové logiky prvního řádu

$$typ(nazev, role, formule <, anotace >).$$

$$fof(pel55\_1, axiom, (?[X] : (lives(X) \& killed(X, agatha)))).$$

Formule mohou mít různé role: *axiom*, *hypothesis*, *definition*, *assumption*, *lemma*, *plain*, *theorem*, *corollary*, *conjecture*, *negated_conjecture*, *type*, *fi_domain*, *fi_functors*, *fi_predicates*, *unknown*.

Pro tento systém nás zajímají pouze některé role, a to:

- *axiom* — Formule reprezentující axiom
- *conjecture* — Formule reprezentující domněnku, kterou se snažíme dokázat
- *negated_conjecture* — Formule reprezentující negaci domněnky, kterou se snažíme dokázat
- *plain* — Formule nemá dle specifikace žádnou danou sémantiku, nicméně v našem případě používáme tuto roli jako roli formulí, odvozených z předchozích formulí.

Po roli následuje zápis samotné logické formule. Ten se skládá z:

- funkčních symbolů — slova začínající malým písmenem
- predikátových symbolů — slova začínající malým písmenem
- proměnných — slova začínající velkým písmenem
- logických spojek:
  - konjunkce &
  - disjunkce |
  - negace ~
- závorek
- kvantifikátorů:
  - univerzální kvantifikátor — ![...] :
  - existenční kvantifikátor — ?[...] :

  kde ... je seznam proměnných oddělených čárkou

Všechny proměnné v zápisu formule typu *fof* — *first-order formula* musí být explicitně kvantifikované. V zápisech formulí typu *cnf* — *clausal normal form* jsou naopak všechny proměnné implicitně univerzálně kvantifikované a kvantifikátory tak neobsahují.

4.2 Kalkulus důkazového systému

- Binární rezoluce

$$\frac{\Gamma_1 \cup L_1 \qquad \Gamma_2 \cup L_2}{(\Gamma_1 \cup \Gamma_2)\sigma}$$

  kde σ je nejobecnějším unifikátorem L_1 a $\neg L_2$.

- Faktorizace

$$\frac{C \vee L_1 \vee L_2}{(C \vee L)\sigma}$$

  kde σ je nejobecnějším unifikátorem L_1 a L_2.

- Paramodulace

$$\frac{C \vee s = t \qquad D}{(C \vee D[t]_p)\sigma}$$

  kde $D|p$ je subtermem D na pozici p, σ je nejobecnější unifikátor s a $D|p$ a $D[t]_p$ značí výsledek nahrazení subtermu D na pozici p termem t

- E-rezoluce

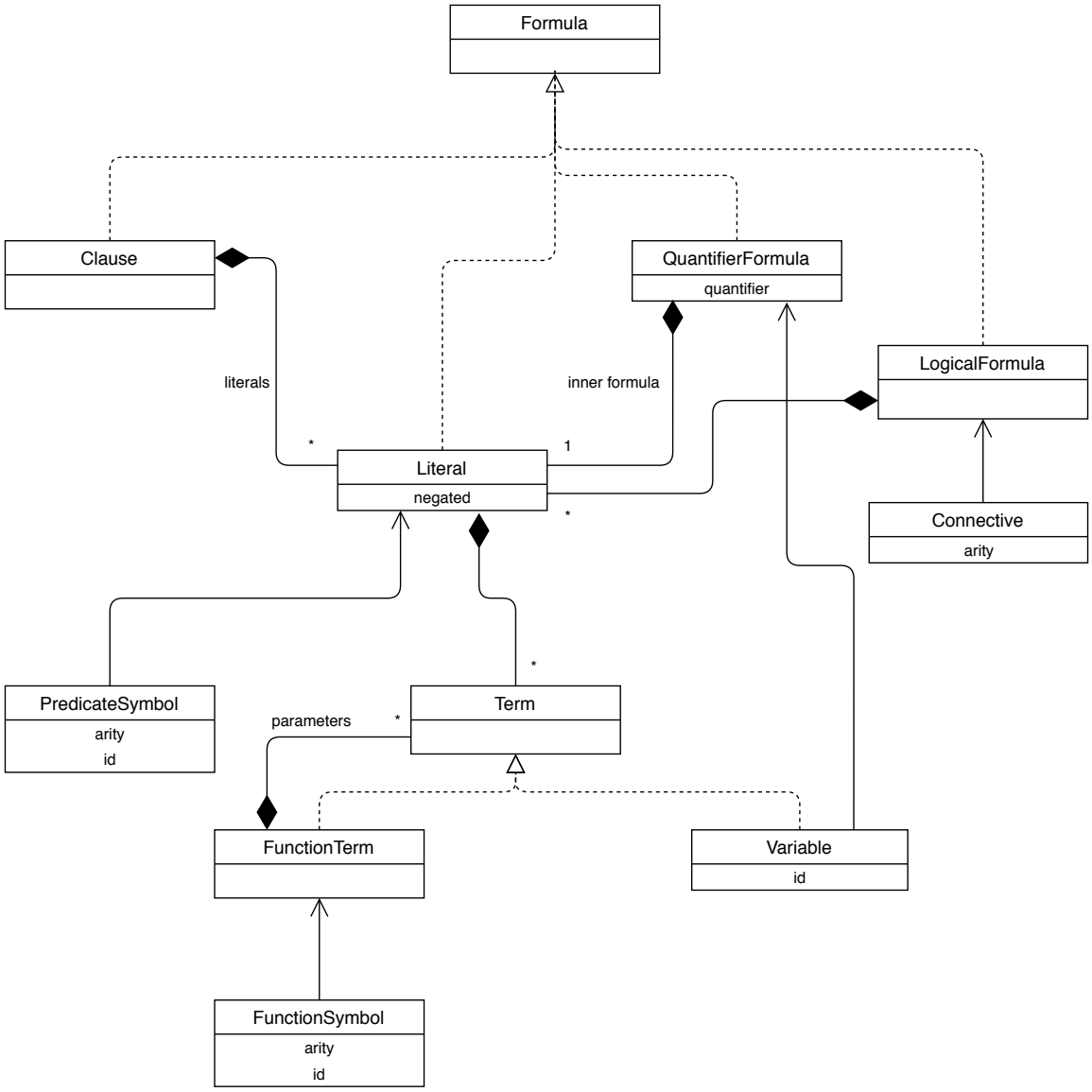
$$\frac{C \vee T_1 \neq T_2}{C\sigma}$$

  kde σ je nejobecnějším unifikátorem T_1 a T_2.

- Dopředné odstranění literálů

4.3 Datový model

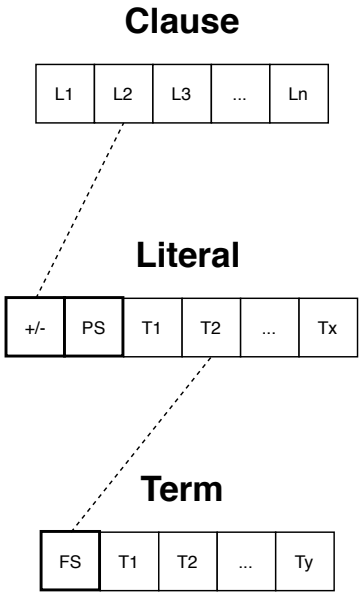
Po načtení vstupu jsou data předělána do interního formátu. Jednotlivé formule jsou reprezentovány jako stromové struktury (viz. obrázek 4.1).



Obrázek 4.1: Model formule

4.3.1 Reprezentace klauzulí

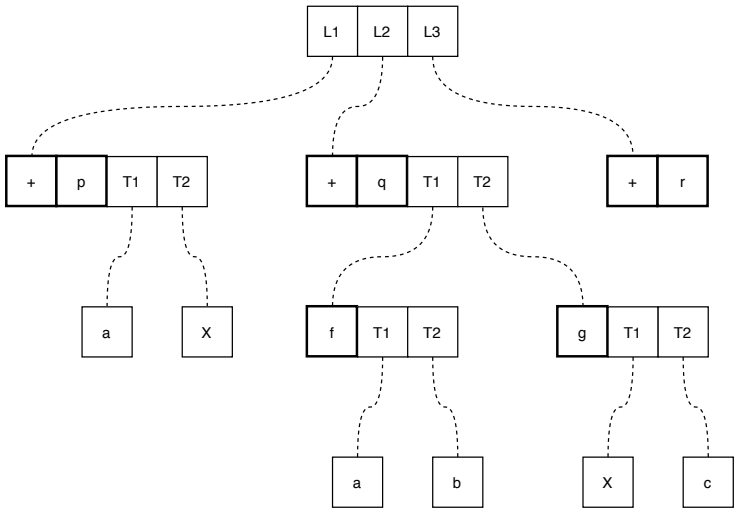
Klauzule jsou reprezentovány seznamem literálů (viz. obrázek 4.2).



Obrázek 4.2: Model klauzule

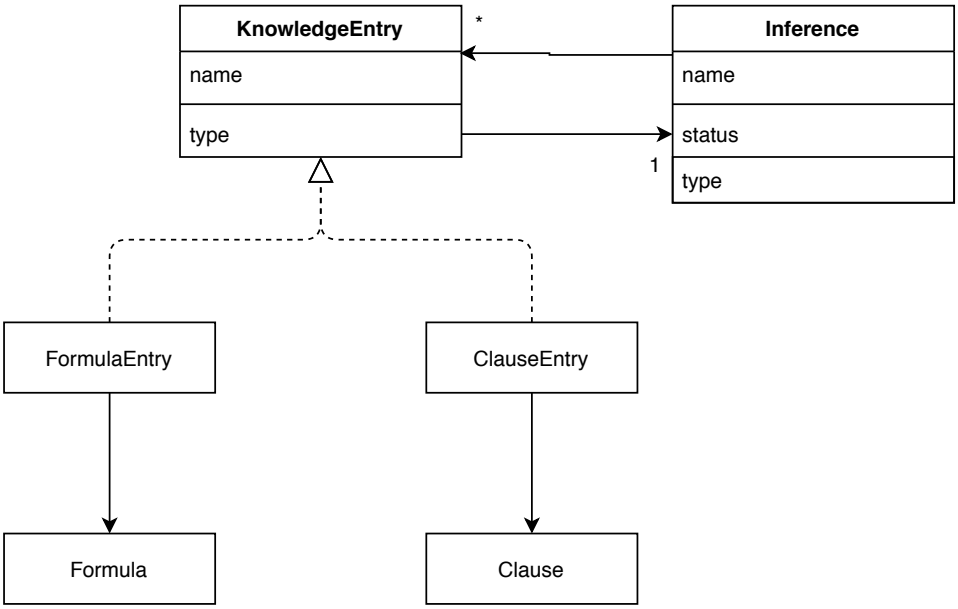
Klauzule tak může mít následující podobu:

p(a, X) v q(f(a, b), g(X, c)) v r



Obrázek 4.3: Příklad modelu klauzule

Při tvorbě důkazu ovšem nestačí pouhá reprezentace logické části klauzule (resp. formulí obecně), ale potřebujeme také znát původ této klauzule (reps. formule). Proto je každá zachovaná formule obalena záznamem *KnowledgeEntry*, který mimo referenci na logickou reprezentaci dané formule zaznamenává také její název, typ a původ dané formule.



Obrázek 4.4: Model záznamu

Prover při své funkci pracuje převážně s takto obalenými formulemi a při každém odvození nové formule tak vzniká i strom jejího původu.

4.4 Obecná architektura systému

Systém pracuje s formulemi logiky prvního řádu, což v případě *TPTP* znamené *fof* a *cnf*. Jelikož rezoluce vyžaduje formule v konjunktivní normální formě, tak jsou formule typu *fof* na začátku dokazování převedeny na odpovídající formule *cnf*. Automatická dedukce pak probíhá pouze nad formulemi typu *cnf*. Systém je založen na hlavní smyčce stylu *given-clause*. Skládá se ze dvou seznamů klauzulí *active* a *waiting*, kdy seznam *active* obsahuje zpracované klauzule, to znamená takové klauzule, které mezi sebou aplikovaly všechna odvozovací pravidla, a seznam *waiting* obsahuje klauzule, které čekají na zpracování. Vzhledem k tomu, že je systém stavěný zejména na práci s formulemi v konjunktivní normální formě, jsou *atomické formule* reprezentovány jako *literály*.

4.4.1 Převod do CNF

V první řadě je třeba převést formule do konjunktivní normální formy. Proces převodu obecné formule logiky prvního řádu do odpovídající reprezentace v konjunktivní normální formě je rozdělen na 3 části, a to převod do *negační normální formy*, převod do *Skolemovy normální formy* a *převod do konjunktivní normální formy*. Vzhledem k tomu, že jsou jednotlivé proměnné v rámci našeho proveru unikátně identifikovány, je možné vynechat převod z *negační normální formy* do *prenexní normální formy*, protože není třeba proměnné přejmenovávat, a tak se jedná o pouhý posun kvantifikátorů beze změny jejich pořadí.

Algoritmus 4.1 *Algoritmus pro převod obecné formule do negační normální formy*

```
Vstup: Obecná formule
Výstup: Formule v negační normální formě
function push_negations(Formula: formule, boolean: negate)
    if formule je negace then
        return push_negations(formule.0, ¬negate)
    else if formule je konjunkce then
        n0 ← push_negations(formule.0, negate)
        n1 ← push_negations(formule.1, negate)
        if negate then
            return disjunkce formulí n0 a n1
        else
            return konjunkce formulí n0 a n1
        end if
    else if formule je disjunkce then
        n0 ← push_negations(formule.0, negate)
        n1 ← push_negations(formule.1, negate)
        if negate then
            return konjunkce formulí n0 a n1
        else
            return disjunkce formulí n0 a n1
        end if
    else if formule je literál then
        if negate then
            return negace literálu formule
        else
            return literál formule
        end if
    else if formule je univerzální kvantifikace then
        if negate then
            return existenční kvantifikace push_negations(formule.0, true)
        else
            return univerzální kvantifikace push_negations(formule.0, false)
        end if
    else if formule je existenční kvantifikace then
        if negate then
            return univerzální kvantifikace push_negations(formule.0, true)
        else
            return existenční kvantifikace push_negations(formule.0, false)
        end if
    end if
end function
```

Algoritmus *push_negations* posouvá negace dovnitř formule pomocí *De Morganových zákonů*, *zákonu dvojí negace* a *zákonu o přesunu negace dovnitř kvantifikátoru*

Převod do Skolemovy normální formy

Algoritmus 4.2 *Převod formule v negační normální formě do Skolemovy normální formy*

`Vstup:` *Formule v negační normální formě*
`Výstup:` *Formule ve Skolemově normální formě*
function skolemize*(Formula:* formule*, List:* scopedVars*, List:* generatedSkolemFuncs*)*
  if formule *je literál* ***then***
    return $formule$
  else if formule *je logická spojka* ***then***
    nechť $formule2$ *je nová formule s logickou spojkou* $formule$
    for $n \leftarrow$ *počet podformulí* $formule$ ***do***
      $formule2.n \leftarrow skolemize(formule.n, scopedVars, generatedSkolemFuncs)$
    end for
    return $formule2$
  else if formule *je univerzální kvantifikace* ***then***
    přidej proměnné z $formule$ *do* $scopedVars$
    $formule2 \leftarrow$ skolemize(formule.1, scopedVars, generatedSkolemFuncs)
    odeber proměnné z $formule$ *ze* $scopedVars$
    return $formule2$
  else if formule *je existenční kvantifikace* ***then***
    $newFunc \leftarrow$ *nový funkční symbol s aritou velikosti* $scopedVars$
    přidej $newFunc$ *do* $generatedSkolemFuncs$
    $formule2 \leftarrow$ *nahrazení proměnné existenčního kvantifikátoru za* $newFunc$ *s parametry* $scopedVars$ *ve formuli* $formule$
    return $skolemize(formule2, scopedVars, generatedSkolemFuncs)$
  end if
end function

Algoritmus 4.3 *Převod formule ve Skolemově normální formě do konjunktivní normální formy*

`Vstup:` *Formule ve Skolemově normální formě*
`Výstup:` *Seznam klauzulí reprezentující vstupní formuli*
function clausify*(Formula:* formule*)*
  if formule *je klauzule* ***then***
    return *jednotkový seznam obsahující formule*
  else if formule *je literál* ***then***
    return *jednotkový seznam obsahující klauzuli z literálu*
  else if formule *je konjunkce* ***then***
    c0 $\leftarrow$ *clausify(*formule.0*)*
    c1 $\leftarrow$ *clausify(*formule.1*)*
    return *spojení seznamů* c1 *a* c2
  else if formule *je disjunkce* ***then***
    c0 $\leftarrow$ *clausify(formule.0)*
    c1 $\leftarrow$ *clausify(formule.1)*
    return *klauzule vzniklé spojením každé klauzule z* c0 *s každou klauzulí z* c1
  end if
end function

4.4.2 Unifikace

Algoritmus unifikace vychází z Robinsonova algoritmu[9]. Ten spočívá v procházení stromů formulí do hloubky a postupném tvoření substituce. V našem případě se jedná o unifikaci dvou literálů (unifikovat celé klauzule u binární rezoluce nemusíme), tudíž vrcholem porovnávaných stromů jsou literály. V případě jejich neshody unifikátor neexistuje, v případě shody můžeme porovnávat dál. Další porovnávání už se jedná termů — máme jednu společnou substituci, do které postupně přidáváme mapování jednotlivých proměnných. Tuto substituci pak před dalším procházením stromů aplikujeme na zbývající části stromů, tudíž se nám neprohledaná část procházených stromů při unifikaci mění tak, aby reflektovala dosavadní mapování proměnných. Dojde-li k neshodě funkčních symbolů, tak unifikátor neexistuje. Dojde-li ke stavu, kdy je současný uzel jednoho stromu proměnná a současný uzel druhého stromu je funkční symbol, jehož parametry obsahují danou proměnnou, unifikátor neexistuje.

Algoritmus 4.4 *Algoritmus pro výpočet největšího společného unifikátoru dvou literálů*

```
Vstup: Dva literály l1 a l2
Výstup: Substituce — největší společný unifikátor l1 a l2, nebo chyba
function mgu(Literal: l1, Literal: l2)
    if predikátový symbol l1 ≠ predikátový symbol l2 then
        return chyba
    end if
    s ← prázdná substituce
    for t1,t2 ← l1,l2 do
        t1s ← výsledek substituce s na t1
        t2s ← výsledek substituce s na t2
        s ← spojení substitucí s a mgu(t1s, t2s)
    end for
    return s
end function
```

Algoritmus 4.5 *Algoritmus pro výpočet největšího společného unifikátoru dvou termů*

```
Vstup: Dva termy t1 a t2
Výstup: Substituce — největší společný unifikátor t1 a t2, nebo chyba
function mgu(Term: t1, Term: t2, Substitution: substituce)
    if t1 je proměnná then
        if t2 je proměnná then
            if t1 = t2 then
                return substituce
            else
                return substituce ∪ (t1, t2)
            end if
        else if t2 je funkční symbol then
            if t1 má mapování v substituce then
                return chyba
            else
                return substituce ∪ (t1, t2)
            end if
        end if
    else if t1 je funkční symbol then
        if t2 je proměnná then
            if t2 má mapování v substituce then
                return chyba
            else
                return substituce ∪ (t2, t1)
            end if
        else if t2 je funkční symbol then
            if t1 ≠ t2 then
                return chyba
            else
                for (t1p, t2p) ← (t1, t2) do
                    substituce ← mgu(t1p, t2p)
                end for
                return substituce
            end if
        end if
    end if
end function
```

4.4.3 Výběr klauzule ke zpracování

Výběr klauzule ke zpracování je pravděpodobně nejdůležitější částí algoritmu *given-clause*[7], jelikož ovlivňuje postup vyhledávání. Pokud bychom vybírali nejnovější klauzuli v seznamu *waiting*, tak budeme prohledávat stavový prostor do hloubky a může dojít k zacyklení — vyhledávání by tak nebylo úplné. Naopak výběrem nejstarší klauzule docílíme vyhledávání do šířky — vyhledávání bude úplné, ale za cenu rychlého růstu stavového prostoru. Vybíral lze ale i podle jiných kritérií, jako je velikost klauzule, počet proměnných, druh klauzule a další.

V naší implementaci je možnost výběru klauzule podle dvou kritérií — podle stáří (výběr nejstarší klauzule pro vyhledávání do šířky) a výběrem nejmenší klauzule. Proměnné *selectAge* (resp. *selectSmallest*) představují počet nejstarších klauzulí k výběru (resp. počet nejmenších klauzulí k výběru) a jsou nastaveny při spuštění programu5.2.

Algoritmus 4.6 *Výběr klauzule ke zpracování*

remainingAge ← *selectAge*
remainingSmallest ← *selectSmallest*
while *True* ***do***
  if *remainingAge* = *0 a remainingSmallest* = *0* ***then***
    remainingAge ← *selectAge*
    remainingSmallest ← *selectSmallest*
  end if
  if *remainingAge* > *0* ***then***
    vyber nejstarší klauzuli z waiting
    dekrementuj remainingAge
  else if *remainingSmallest* > *0* ***then***
    vyber nejmenší klauzuli z waiting
    dekrementuj remainingSmallest
  end if
  přidej vybranou klauzuli do seznamu active
  zpracuj vybranou klauzuli
end while

4.4.4 Zpracování vybrané klauzule

Po výběru klauzule následuje aplikace binárních odvozovacích pravidel mezi vybranou klauzulí a každou klauzulí ze seznamu *active*. Po každé aplikaci odvozovacích pravidel nám vzniká množina nových klauzulí, které dále zpracováváme.

Algoritmus 4.7 *Zpracování vybrané klauzule*

for second ← active ***do***
  resolvents ← *výsledek resoluce* second *a vybrané klauzule*
  modulated ← *výsledek paramodulace* second *a vybrané klauzule*
  for *newClause* ← *resolvents* ∪ *modulated* ***do***
    zpracuj novou klauzuli newClause
  end for
end for

4.5 Zpracování nových klauzulí

Nyní jsme ve fázi po aplikování rezoluce (případně paramodulace) na vybranou klauzuli a klauzuli ze seznamu *active*. Máme tak k dispozici seznam nově vytvořených klauzulí, které přijdou do seznamu *waiting*. Tyto klauzule můžeme přidat do seznamu *waiting* rovnou, ale mohlo by tak dojít k duplicitě známých klauzulí, která by se postupem času prudce zvyšovala. Před přidáním klauzule do seznamu *waiting* tak prvně provádíme optimalizační opatření, aplikujeme unární odvozovací pravidla, jejichž výsledné klauzule zpracujeme, a až poté přidáváme danou klauzuli do seznamu *waiting*. Zamezujeme tím redundanci.

Testujeme, zdali se daná klauzule už nevyskytuje v seznamu *active* nebo *waiting*. Následně přidáváme danou klauzuli do seznamu *waiting* a provádíme na ní *faktorizaci*. Klauzule vzniklá faktorizací dané klauzule opět zpracováváme pomocí stejné procedury.

Algoritmus 4.8 *Algoritmus pro zpracování nově generované klauzule*

```
procedure addGenerated(Clause: clause)
    if clause je v seznamu active, nebo ji lze vytvořit substitucí klauzule v active then
        zahoď klauzuli clause
    end if
    if clause je v seznamu waiting, nebo ji lze vytvořit substitucí klauzule v waiting then
        zahoď klauzuli clause
    end if
    přidej clause do waiting
    factors ← faktorizace clause
    for currentClause ← factor do
        zpracuj novou klauzuli currentClause
    end for
    resolvents ← e-rezoluce clause
    for currentClause ← factor do
        zpracuj novou klauzuli currentClause
    end for
end procedure
```

Tato část našeho dokazovacího systému je dobrým místem pro přidání případných optimalizací. Mimo

4.6 Odvozovací pravidla

V kapitole 3 jsme rozebírali teoretické podklady dokazování a představili jsme odvozovací pravidla, která používáme. V této sekci tuto teorii převedeme do praktičtějšího pohledu a navrhneme algoritmy pro vykonávání těchto odvozovacích pravidel.

4.6.1 Binární rezoluce

Naše hlavní odvozovací pravidlo je binární rezoluce, které spojuje dvě klauzule pomocí unifikace dvou komplementárních literálů. Její průběh tedy začíná vyhledáním dvojice literálů se stejným predikátovým symbolem. U těchto literálů se pokoušíme unifikovat a pokud unifikátor existuje, tak vytváříme novou klauzuli — rezolventu — spojením zbývajících literálů obou klauzulí a následnou aplikací získaného unifikátoru.

Algoritmus 4.9 *Výpočet rezolvent z klauzulí* current *a* second

```
Vstup: Dvě klauzule — current a second
Výstup: Seznam všech rezolvent klauzulí current a second
resolvents ← prázný seznam
for currentLiteral ← current do
    for secondLiteral ← second do
        if currentLiteral a secondLiteral mají odlišná znaménka then
            s ← mgu(currentLiteral, secondLiteral)
            if existuje s then
                remainingCurrent ← literály z current kromě currentLiteral
                remainingSecond ← literály z second kromě currentLiteral
                newClause ← spojení remainingCurrent a remainingSecond
                aplikuj s na newClause a výsledek přidej do resolvents
            end if
        end if
    end for
end for
```

4.6.2 Paramodulace

Při paramodulaci pracujeme se dvěmi klauzulemi — v jedné hledáme predikáty ekvivalence a termy, které se rovnají, a ve druhé hledáme termy odpovídající termu jedné strany ekvivalence a nahrazujeme je druhou stranou.

Algoritmus 4.10 *Výpočet paramodulace z klauzulí* current *a* second

```
Vstup: Dvě klauzule — current a second
Výstup: Seznam všech klauzulí získaných paramodulací current na second
modulated ← prázný seznam
for currentLiteral ← current do
    if currentLiteral je rovnost a currentLiteral je positivní then
        leftTerm ← currentLiteral.0
        rightTerm ← currentLiteral.1
        otherLiterals ← seznam literálů current bez currentLiteral
        for secondLiteral ← second do
            for position ← find(second, leftTerm) do
                newClause ← replaceOrSubstitute(second, position, rightTerm)
                přidej do newClause literály otherLiterals po substituci position.substitution
                přidej newClause do seznamu modulated
            end for
            for position ← find(second, rightTerm) do
                newClause ← replaceOrSubstitute(second, position, leftTerm)
                přidej do newClause literály otherLiterals po substituci position.substitution
                přidej newClause do seznamu modulated
            end for
        end for
    end if
end for
```

4.6.3 Faktorizace

Faktorizace je implementačně podobná binární rezoluci. Na rozdíl od binární rezoluce ale pracujeme pouze s jednou klauzulí a hledáme unifikátory mezi literály, které jsou buď oba negované, nebo oba bez negace.

Algoritmus 4.11 *Faktorizace klauzule*

factors ← *prázný seznam*
for *currentLiteral* ← *current* ***do***
  for *secondLiteral* ← *zbývající část current* ***do***
    if *currentLiteral má stejný PS a znaménko jako secondLiteral* ***then***
      $s \leftarrow mgu(currentLiteral, secondLiteral)$
      if *existuje s* ***then***
        newClause ← *klauzule current bez currentLiteral*
        přidej výsledek substituce s na newClause do factors
      end if
    end if
  end for
end for

4.6.4 E-rezoluce

Pro řešení problémů s rovností jsme si představili paramodulaci, ale ta neřeší nerovnost. Proto přidáváme *E-rezoluci*, která pracuje s nerovností. Při *E-rezoluci* pracujeme s jednou klauzulí a snažíme se najít unifikátor mezi dvěmi stranami nerovnosti. Pokud unifikujeme obě strany nerovnosti, tak nám z daného literálu vzniká kontradikce a můžeme ho z klauzule odstranit — unifikací obou stran nerovnosti zajistíme, aby se dané termy rovnaly — nemohou se nerovnat.

Algoritmus 4.12 *E-rezoluce klauzule*

resolvents ← *prázný seznam*
for currentLiteral ← current ***do***
  if *currentLiteral je rovnost a currentLiteral je negativní* ***then***
    $leftTerm \leftarrow currentLiteral.0$
    $rightTerm \leftarrow currentLiteral.1$
    $s \leftarrow mgu(leftTerm, rightTerm)$
    if *existuje s* ***then***
      newClause ← *klauzule current bez currentLiteral*
      přidej výsledek substituce s na newClause do resolvents
    end if
  end if
end for

4.7 Srovnání se state-of-the-art systémy

V této sekci se podíváme na použitých odvozovacích pravidel několika vybraných satte-of-the-art theorem proverů.

4.7.1 E

E[1] je přes 20 let starý theorem prover vyvinutý Stephanem Schulzem na Technické univerzitě v Mnichově. Byl vyvinut jako automatický theorem prover pro predikátovou logiku prvního řádu, ale jeho doména se postupně rozšířila i na některé druhy logiky vyššího řádu. Využívá kalkulus superpozice, což je optimalizovaná variace paramodulace, která využívá řazení termů pro snížení redundance.

4.7.2 OTTER

OTTER[7][2] (Organized Techniques for Theorem-proving and Effective Research) je theorem prover pro predikátovou logiku prvního řádu, který je nyní již nahrazený jeho novější podobou *Prover9*[3], ale přesto stále používaný. Používá několik druhů rezoluce a paramodulace.

4.7.3 Vampire

Vampire[4][4] je velmi rychlý prover, který vyhrál nejvíce prvních míst na soutěži automatických theorem proverů CASC[10]. Vampire je automatický theorem prover pro prefikátovou logiku prvního řádu. Používá rezoluci a superpozici. Jako hlavní smyčku má k dispozici tři variace given-clause algoritmu, z nichž je zajímavá *limited-resource-strategy*, která je stavěná pro využití s omezenými zdroji (paměť, čas) a promazává z *passive* seznamu (v naší implementaci značen jako waiting) klauzule, které by byly pravděpodobně vybrány až po vyčerpání zdrojů. Porušuje se tím úplnost systému, ale v některých případech je toto zrychlení výhodnější, než zachování úplnosti.

4.7.4 Shrnutí představených systémů

Ve výše uvedených systémech můžeme vidět značné podobnosti — rezoluci[5] a superpozici. U state-of-the-art automatických theorem proverech je tato kombinace velice častá, ovšem použitá odvozovací pravidla jsou jen malou částí úspěchu. Velkou otázkou je na jakých klauzulích/formulích tato pravidla aplikovat.

[1] http://www.eprover.org/
[2] https://www.cs.unm.edu/~mccune/otter/
[3] https://www.cs.unm.edu/~mccune/prover9/
[4] http://www.vprover.org/
[5] Rezoluce je v tomto případě myšlena obecně — ne pouze binární rezoluci

Kapitola 5

Implementace

Prover je implementován pomocí programovacího jazyka *Java* s pomocí frameworku *Spring Boot* pro tvorbu spustitelného *jar* souboru a nástroje *Maven* pro build. Výběr těchto technologií je čistě z osobní preference.

5.1 Optimalizace

V průběhu vývoje se brzy a výrazně začaly projevovat nedostatky spojené s prohledáváním do šířky, a to zahlcení paměti. Zatímco jednoduché problémy šly vyřešit během několika milisekund, problémy s delšími důkazy byly prakticky nesplnitelné, ať už z důvodu nedostatku paměti, nebo z důvodu překročení časového limitu.

5.1.1 Odstranění literálů

Při dedukci můžeme narazit na klauzule, skládající se z právě jednoho literálu. Pokud při dalším odvozování narazíme na klauzuli obsahující negaci takového literálu, můžeme ji z klauzule odstranit a udělat tím prakticky jeden krok rezoluce.

5.1.2 Odstranění redundantních klauzulí

Při dedukci často nacházíme klauzule, které už známe, nebo které mohou vzniknout substitucí známé klauzule. Pokud bychom neodstraňovali, tak by v seznamech *active* a *waiting* vznikali duplikáty, které by způsobovaly duplicitní aplikaci odvozovacích pravidel a zbytečně tak snižovaly rychlost proveru. Při každém přidání nové klauzule tedy prvně zjišťujeme, zdali přidávaná klauzule nejde vytvořit substitucí existující klauzule. Pokud ano, tak přidávanou klauzuli zahodíme. Tato optimalizační technika lze použít i zpětně — pokud existující klauzule lze vytvořit substitucí nově přidávané, tak existující klauzuli můžeme odstranit. Zpětné odstranění ovšem v našem proveru implementováno není.

5.2 Profiling

Při vývoji proveru byly dva nejčastější problémy — nedostatek paměti a příliš pomalé odvozování nových klauzulí. Pro snadnější lokalizaci problémů jsou do proveru zakomponovány jednoduché profilovací čítače.

Tabulka 5.1: Údaje vypsané při zapnutém profilingu

clausesGenerated
clausesDeduces
clausesFind
clausesReplaceOrSubstitute
literalsFind
literalsDeduces
literalsMgu
literalsReplaceOrSubstitute

5.3 Použití

Prover je implementován v jazyce Java a s tím souvisí potřeba nainstalovaného Java Runtime Enviroment[1] ke spuštění. Prover se spouští v konzoli jako *executable jar*. Prover nedisponuje žádným grafickým uživatelským rozhraním.

java −jar jatp.jar ... argumenty ... problemFile.p

Pro spuštění je třeba mít odkaz na složku s binárnímy soubory Javy v systémové proměnné *PATH*, případně nahradit příkaz java plnou cestou k odpovídajícímu binárnímu souboru Java Runtime Enviroment.

5.3.1 Kompilace

Prover je sestaven pomocí nástroje *Maven*[2].

mvn clean install

Výsledky sestavení programu včetně spustitelného souboru jar jsou uloženy do adresáře *target*.

[1] https://www.java.com/en/download/
[2] https://maven.apache.org/

5.3.2 Konfigurace

Tabulka 5.2: Argumenty konfigurace

argument	význam
—basedir DIR	nastavení adresáře se soubory logických problémů
—includes DIR	nastavení adresáře pro načítání *include* souborů
—debug	zapnout výpis ladících informací
—verbose	bohatší výpis programu
—maxtime N	nastavení maximálního času běhu programu na N sekund
—maxAge N	počet aktivních klauzulí vybíraných podle stáří (bfs)
—maxSize N	počet aktivních klauzulí vybíraných podle počtu literálů
—maxClauseSize N	nastaví maximální velikost nově generovaných klauzulí na N
—maxClauseVariables N	maximální počet proměnných v nově generovaných klauzulích
—dumpActive	výpis akvitních klauzulí na konci běhu programu
—dumpUnist	výpis jednotkových klauzulí na konci běhu programu
—dumpRewrite	výpis pravidel paramodulace na konci běhu programu

Nastavení maximální paměti

Při hledání důkazu vzniká velké množství klauzulí, které rychle zaplňují paměť. Je proto důležité nastavit proveru vhodné množství paměti, aby měl dostatečný prostor pro dosažení požadovaného výsledku. Množství přidělené paměti můžeme nastavit pomocí argumentu JVM při spouštění proveru argumentem *-Xmx*

$$java\ -Xmx8g\ -jar\ jatp.jar\ \ldots\ argumenty\ \ldots\ problemFile.p$$

Nastavení maximální doby běhu

Rezoluce je založena na hledání kontradikce, která nemusí vždy existovat (=domněnka není logickým důsledkem axiomů). V našem proveru se taková situace ve většině případů nedá odhalit, a tak hledání důkazu nedokazatelné domněnky může běžet do té doby, než dojdou systémové zdroje—paměť. Abychom kvůli tomu nenechávali prover běžet bezvýznamně dlouho zavádíme omezení běhu programu. To nastavujeme argumentem *-maxtime N*, kde *N* značí maximální dobu běhu v sekundách. Implicitní hodnota tohoto nastavení je 60 sekund.

$$java\ -jar\ jatp.jar\ -maxtime\ 300\ problemFile.p$$

Kapitola 6

Experimenty

6.1 Experimenty v průběhu vývoje

Pro experimentování s vyvíjeným proverem bylo vytvořeno několik jednodušších problémů, které testují různé logické zákony a jednotlivé funkce proveru. Tvoří jakousi roli unit testů. Tyto experimenty jsou díky své složitosti (nebo spíše jednoduchosti) v rámci srovnávání theorem proverů prakticky nicneříkající, ale ve fázi vývoje jsou silným nástrojem pro lokalizaci nedostatků a pro ladění proveru.

6.2 Srovnání paramodulace a rezoluce s axiomy rovnosti

V teoretické části této práce popisujeme, jak lze dokazovat formule s rovností za použití *rezoluce* a *axiomů rovnosti* bez použití *paramodulace*. Abychom zjistili, zda v našem případě měla implementace paramodulace smysl, bylo u několika problémů vytvořeny podobné, s nahrazením rovnosti za binární predikát *eq* a přidáním axiomů rovnosti. Můžeme tak porovnávat řešení problémů s použitím a bez použití paramodulace.

Tabulka 6.1: Srovnání paramodulace a axiomů rovnosti

problém	průměr s pm	active s pm	průměr bez pm	active bez pm
1[1]	23ms	6	28ms	10
2	33ms	10	45ms	16
3	173ms	10	timeout	879

Na námi řešených problémech k porovnání paramodulace a axiomů rovnosti (viz. tabulka 6.1) lze vidět, že použití axiomů rovnosti je pomalejší a v jednom případě jsme se ani nedokázali v čase 60 sekund dostat k důkazu. Pokud se zaměříme na počty klauzulí v seznamu *active*, tak zjistíme, že verze s paramodulací dosáhla výsledku při použití menšího množství aktivních klauzulí. Z toho lze vyčíst že při paramodulaci postupujeme ke kontradikci efektivnějšími kroky a nezahlcujeme prohledávaný prostor axiomy rovnosti. Nutno podotknout že ze strany rezoluce je implementována pouze ta nejzákladnější verze — *binární rezoluce*. Tak jako existuje paramodulace pro zlepšení dokazování rovnosti existují také další variace rezoluce pro zlepšení dokazování problémů s různou syntaktickou strukturou klauzulí.

[1] Problém 1 odpovídá souboru p4.p, problém 2 souboru p5.p a problém 3 souboru p7.p ve složce problems. Varianty bez paramodulace mají v názvu slovo *axioms*

6.3 Experimenty na problémech z TPTP

Knihovna problémů TPTP[11] dělí problémy do několika oborů, které kategorizují problémy. Jsou jimi *Logika*, *Matematika*, *Počítačová věda*, *Věda a inženýrství*, *Sociální vědy*, *Umění a humanitní obory* a *Ostatní*. Tyto obory se dále dělí do subkategorií—domén, které mají přiřazeny unikátní adresáře v knihovně *TPTP* a prefixy. Například *teorie čísel* — *NUM*.

Součástí komentářů s souborech problémů TPTP je *Rating* — číslo z intervalu $< 0, 1 >$ značící poměr počtu proverů registrovaných v TPTP, které daný problém nevyřešily, vůči počtu proverů, které se o to pokoušely. To znamená že problém s ratingem 0 dokáží vyřešit všechny provery registrované v TPTP a problém s ratingem 1 žádný prover registrovaný v TPTP vyřešit nedokázal. Vzhledem k tomu, že jsou v TPTP registrovány převážně state-of-the-art provery, lze očekávat, že náš prover bude schopen řešit pouze problémy s nulovým ratingem.

Pro experimenty bylo vybráno 20 problémů ze subkategorie NUM a 20 problémů ze subkategorie PUZ. Experimenty probíhaly s implicitně nastaveným výběrem klauzulí (střídavě jedna nejstarší a jedna s nejnižším počtem literálů) a implicitním časovým omezením 60 sekund.

Tabulka 6.2: Experimenty na problémech TPTP subkategorie NUM

problém	čas [ms]	active	waiting	rating
NUM001-1	timeout	145	21616	0
NUM002-1	timeout	145	20730	0
NUM003-1	timeout	145	20775	0
NUM004-1	timeout	145	21735	0
NUM005-1	timeout	176	20731	0.94
NUM006-1	timeout	91	12613	1
NUM007-1	timeout	169	21721	0.76
NUM008-1	timeout	92	13480	1
NUM009-1	timeout	95	13869	0.06
NUM010-1	timeout	96	13895	0.94
NUM011-1	timeout	96	13679	0.53
NUM012-1	timeout	96	14001	1
NUM013-1	timeout	99	13796	0
NUM014-1	74	94	178	0
NUM015-1	476	60	1144	0
NUM016-1	3327	118	4157	0
NUM017-1	timeout	235	27760	0.33
NUM018-1	timeout	95	13867	1
NUM019-1	2056	120	3947	0
NUM020-1	1491	112	3063	0

Mezi prvními 20 problémy ze subkategorie NUM (viz. tabulka 6.2) se nachází větší množství složitějších problémů, a tak úspěšnost nalezení důkazu nebyla vysoká. U neúspěšných problémů si můžeme všimnout vysokého počtu čekajících klauzulí, způsobujícího zpomalení v hledání důkazu.

Tabulka 6.3: Experimenty na problémech TPTP subkategorie NUM

problém	čas [ms]	active	waiting	rating
PUZ001-1	81	36	65	0
PUZ002-1	75	28	17	0
PUZ003-1	231	44	678	0
PUZ004-1	64	26	9	0
PUZ005-1	timeout	461	30400	0
PUZ006-1	timeout	196	3927	0.06
PUZ007-1	timeout	193	4770	0.06
PUZ008-1	77	34	128	0
PUZ009-1	62	28	36	0
PUZ010-1	timeout	735	28346	0
PUZ011-1	124	268	167	0
PUZ012-1	504	66	1512	0
PUZ013-1	87	52	134	0
PUZ014-1	432	102	397	0
PUZ015-1	timeout	150	4048	0.73
PUZ016-1	timeout	149	3933	0
PUZ017-1	timeout	379	34999	0
PUZ018-1	timeout	131	21925	0
PUZ019-1	timeout	431	20425	0
PUZ020-1	426	60	204	0

Při experimentování na subkategorii PUZ (viz. tabulka 6.3) je úspěšnost vyšší, než u subkategorie NUM, čemuž přispívá vyšší zastoupení problémů s ratingem 0.

V obou testovaných subkategoriích se nám podle očekávání nepodařilo dokázat problém s ratingem vyšším, než 0. Zároveň jsme nebyli schopni dokázat část problémů s ratingem 0, které jsou pro state-of-the-art provery jednoduché. Velká část těchto problémů byla složená pouze z Hornových klauzulí a pro takové problémy existují efektivní strategie, které běžně používané provery obsahují. V naší implementaci však taková strategie není, a proto bylo řešení těchto problémů z velké části neúspěšné.

Stejné příklady jsme pro porovnání vyzkoušeli s výběrem pouze nejstarší klauzule, oproti defaultnímu střídání nejstarší a nejmenší. Výsledkem byl pouze jeden úspěšný důkaz z problémů subkategorie NUM a 6 úspěšných důkazů ze subkategorie PUZ. Toto zhoršení je způsobeno častějším výběrem větších klauzulí, které jsou pomalejší na zpracování a při aplikaci odvozovacích pravidel generují více nových klauzulí, vedoucích k rychlejšímu zahlcení stavového prostoru.

6.4 Zhodnocení experimentů

S jednoduchými problémy si náš prover poradí, ale při stoupající složitosti a zvyšujícím se počtu vstupních formulí—axiomů problému už brzy nestačí.

Kapitola 7

Závěr

V této práci jsme se zabývali návrhem a implementací automatického theorem proveru založeném na rezoluci a paramodulaci. Dále jsme představili formát a knihovnu problémů TPTP a pokoušeli jsme se některé problémy z této knihovny řešit pomocí našeho automatického theorem proveru.

Podařilo se vytvořit automatický theorem prover, schopný řešit jednodušší problémy, a demonstrovat na něm zefektivnění řešení problémů predikátové logiky prvního řádu s ekvivalencí. Dále byly implementovány jednoduché optimalizační prvky pro další zefektivnění hledání důkazu. Stručně jsme popsali několik vybraných sate-of-the-art theorem proverů.

Konkurovat state-of-the-art proverům bylo ale prakticky nemožné. Na počátku práce jsem o existenci automatických theorem proverů skoro nic nevěděl a většinu věcí jsem se učil za pochodu. Pokud bych měl prover dělat znovu, tak bych určitě návrh více zaměřil na řazení termů, které by umožnilo použití superpozice a efektivnějších optimalizačních pravidel. Z krátkodobého pohledu vylepšení současné implementace by nejvíce pomohla implementace zpětného odstranění klauzulí.

Nicméně tvorbou této práce jsem si vytvořil přehled v automatických theorem proverech, jakožto zajímavé a mnou do této doby neznámé oblasti umělé inteligence.

Literatura

[1] Ben Ari, M. *Mathematical Logic for Computer Science*. 2. vyd. Springer, 2001. ISBN 1-85233-319-7.

[2] Fitting, M. *First-Order Logic and Automated Theorem Proving*. 2. vyd. Springer, 1996. ISBN 978-1-4612-7515-2.

[3] Huth, M. a Ryan, M. *Logic in Computer Science*. 2. vyd. Cambridge University Press, 2004. ISBN 0-511-26401-1.

[4] Kovács, L. a Voronkov, A. *First-Order Theorem Proving and Vampire*. Červenec 2013. Dostupné z: http://dx.doi.org/10.1007/978-3-642-39799-8_1.

[5] Leitsch, A. *The Resolution Calculus*. 1. vyd. Springer, 1997. ISBN 3-540-61882-1.

[6] Lengál, O. *IAM — First-Order Logic*. 2020.

[7] McCune, W. *Otter 3.3 Reference Manual*. Listopad 2003. Dostupné z: http://dx.doi.org/10.2172/822573.

[8] Nieuwenhuis, R. a Rubio, A. *Paramodulation-Based Theorem Proving*. Srpen 2001. Dostupné z: http://dx.doi.org/10.1016/B978-044450813-3/50009-6.

[9] Robinson, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*. New York, NY, USA: Association for Computing Machinery. leden 1965, roč. 12, č. 1, s. 23–41. Dostupné z: https://doi.org/10.1145/321250.321253. ISSN 0004-5411.

[10] Sutcliffe, G. The CADE ATP System Competition - CASC. *AI Magazine*. 2016, roč. 37, č. 2, s. 99–101.

[11] Sutcliffe, G. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*. 2017, roč. 59, č. 4, s. 483–502.

[12] Zbořil, F. a Zbořil, F. *Základy umělé inteligence IZU studijní opora*. 2012.

[13] Švejdar, V. *Logika neúplnost, složitost a nutnost*. 1. vyd. Academia, 2002. ISBN 80-200-1005-X.

Příloha A

Obsah média

Kořenová složka datového média obsahuje následující:

jatp-1.0-Final.jar — zkompilovaný prover ve formátu jar

experimenty.odt — tabulka s výsledky experimentů

Axioms — složka obsahující axiomy k testovaným problémům z TPTP

problems — složka obsahující soubory problémů z TPTP a soubory problémů pro testování při vývoji

program — složka obsahující zdrojové soubory, které jsou zároveň k nalezení na `https://github.com/nekdozjam/jatp`

zprava — složka obsahující zdrojové soubory tohoto dokumentu

xmazan09-automaticky-theorem-prover.pdf — tento dokument v barevném provedení

xmazan09-automaticky-theorem-prover-tisk.pdf — tento dokument ve formě pro tisk