View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace at VSB Technical University of Ostrava

VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Návrh algoritmu pro rozhodování vybraných podtříd predikátové logiky 1. řádu

Algoritm Deciding Delected Subclasses of First-order Predicate Logic Formulas

2016

Bc. Daniel Šudřich

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Daniel Šudřich**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Návrh algoritmu pro rozhodování vybraných podtříd predikátové logiky 1. řádu
Algoritm Deciding Delected Subclasses of First-order Predicate Logic Formulas

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je návrh algoritmu pro rozhodování vybraných podtříd formulí predikátové logiky 1. řádu. Práce bude navazovat na diplomovou práci Radima Vašíčka, obhájenou v r. 2012 s názvem "Rozhodnutelné podtřídy formulí predikátové logiky 1. řádu". V této obhájené práci uvedl absolvent několik rozhodnutelných podtříd. Student tedy navrhne algoritmy pro dokazování ve vybraných rozhodnutelných podtřídách.
Práce bude obsahovat:
1. Přehled rozhodnutelných podtříd predikátové logiky 1. řádu.
2. Výběr důkazového kalkulu a zdůvodnění tohoto výběru.
3. Algoritmus implementace důkazových metod pro tento vybraný kalkul a vybrané podtřídy.

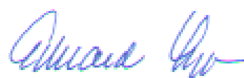
Seznam doporučené odborné literatury:

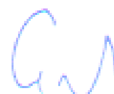
[1] Švejdar, V.: Logika - neúplnost, složitost a nutnost. Academia Praha 2002
[2] ochor, A.: Klasická matematická logika. Karolinum Praha 2001.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. RNDr. Marie Duží, CSc.**

Datum zadání: 01.09.2013
Datum odevzdání: 29.04.2016

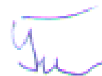




doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

„Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal."

V Ostravě dne 29.6.2016  ....

Poděkování

Chtěl bych tímto poděkovat paní prof. RNDr. Marie Duží, CSc, bez které by tato práce nevznikla. Obzvláště si vážím jejího přístupu a její podpory, která mě motivovala.

Abstrakt

Problémem v predikátové logice je, že logická pravdivost formulí je nerozhodnutelná. To znamená, že neexistuje všeobecný algoritmus, který by vždy s absolutní jistotou rozhodnul, zda je daná formule predikátové logiky tautologie, kontradikce nebo jen splnitelná. Avšak existují velké oblasti, kde tento problém je úspěšně zvládnutý a těmi jsou rozhodnutelné podtřídy. Tyto podtřídy byly diplomovou prací Radima Vašíčka z roku 2012 s názvem "Rozhodnutelné podtřídy formulí predikátové logiky 1. řádu" představeny a dokázány. Tato práce postupně tyto podtřídy představuje. Navrhne a implementuje algoritmus pro rozhodování. Výsledkem je program, který na vstupu přijme formuli predikátové logiky, ověří, zda vyhovuje některé z podtříd a za pomocí implementace důkazového kalkulu, tuto podtřídu rozhodne.

Klíčová slova**:** Predikátová logika 1. řádu, rozhodnutelnost, rozhodnutelné podtřídy, algoritmus, rezoluční metoda

Abstract

The problem in predicate logic is that the logical truthfulness of formulas is undecidable. This means that there is no general algorithm that should always be decided with absolute certainty whether a given formula of predicate logic tautology, contradiction or only achievable. However, there are large areas where the problem is successfully mastered and those are decidable subclasses. These subclasses have been introduced in the diploma thesis of Radim Vasicek from 2012 called "Decidable subclass of predicate logic formulas of the first order." This thesis presents gradually these subclasses. Designs and implements an algorithm for decision making. The result is a program that accepts input formula of predicate logic verifies that meets some of the subclasses and by implementing proving calculus, this subclass decides.

Keywords**:** First-order predicate logic, decidability, decidable subclasses, algorithm, resolution method

Obsah

1. Úvod ... 4
1.1 Cíl práce ... 4
1.2 Postup řešení ... 4
1.3 Implementační technologie ... 4
2. Rozhodnutelné podtřídy predikátové logiky 1. řádu ... 5
2.1 Predikátová logika ... 5
2.1.1 Jazyk predikátové logiky 1. řádu ... 5
2.1.2 Volné a vázané proměnné ... 6
2.1.3 Rozhodnutelnost ... 7
2.2 Prenexní tvar formule ... 7
2.3 Rozhodnutelné podtřídy ... 8
2.2.1 Löb-Gurevichova třída ... 9
2.3.2 Gödel-Kalmár-Schütteova třída ... 9
2.3.3 Bernays-Schönfinkelova třída ... 9
2.3.4 Gurevich-Maslov-Orevkovova třída ... 10
2.3.5 Gurevichova třída ... 10
2.3.6 Rabinova třída ... 10
2.3.7 Shelahova třída ... 11
3. Důkazový kalkul ... 12
3.1 Skolemova forma ... 12
3.2 Rezoluční metoda ... 14
3.3 Robinsonův unifikační algoritmus ... 15
4. Datová struktura ... 16
5. Návrh algoritmu ... 17
5.1 Algoritmus ... 17
5.2 Algoritmus pro rozhodování ve vybraných podtřídách PL1 ... 17
6. Implementace algoritmů rozhodnutelných podtříd ... 19
6.1 Čtení vstupní formule ... 19
6.2 Konstrukce syntaktického stromu ... 20
6.3 Převod do prenexního tvaru ... 21
6.4 Návrh a implementace algoritmu pro vybrané podtřídy ... 28
6.4.1 Algoritmus pro Löb-Gurevichovu třídu ... 28
6.4.2 Algoritmus pro Gödel-Kalmár-Schütteovu třídu ... 29
6.4.3 Algoritmus pro Bernays-Schönfinkelovu třídu ... 31

6.4.4 Algoritmus pro Gurevich-Maslov-Orevkovovu třídu ... 32
6.4.5 Algoritmus pro Gurevichovu třídu ... 32
6.4.6 Algoritmus pro Rabinovu třídu ... 34
6.4.7 Algoritmus pro Shelahovu třídu ... 34
7. Implementace důkazového kalkulu ... 36
7.1 Skolemova forma ... 36
7.2 Rezoluční metoda ... 42
7.3 Robinsonův unifikační algoritmus ... 44
8. Závěr ... 46
Literatura ... 47
Seznam výpisu zdrojového kódu ... 48
Seznam obrázků ... 49

1. Úvod

1.1 Cíl práce

Cílem práce je navrhnout algoritmus pro rozhodování ve vybraných podtřídách formulí predikátové logiky 1. řádu. Obecně je známo, že predikátová logika 1. řádu je nerozhodnutelná a proto bude práce navazovat na obhájenou diplomovou práci Radima Vašíčka z roku 2012 s názvem "Rozhodnutelné podtřídy formulí predikátové logiky 1. řádu" [1]. V této práci student uvedl několik podtříd predikátové logiky 1. řádu, ve kterých je problém logické pravdivosti rozhodnutelný. Práce tedy bude obsahovat návrh a implementaci algoritmů pro jednotlivé podtřídy a to včetně výběru a implementace důkazového kalkulu. Výstupem práce bude program, který na vstupu přijme uživatelem zadanou formuli, ověří, zda formule patří do některé z rozhodnutelných podtříd, a následně určí, zda se jedná o formuli rozhodnutelnou nebo nikoliv.

1.2 Postup řešení

Jelikož tato práce navazuje na předešlou obhájenou práci, nebude se práce zabývat důkazy v matematické logice, protože vše potřebné již bylo dokázáno v předešlé práci. Práce se ovšem neobejde bez úvodu do problematiky a základních definic. Proto tato práce v prvních kapitolách poskytne úvod do terminologie a definic, které budou v průběhu práce vyžadovány.

Nejprve si stručně popíšeme, co je to predikátová logika a její rozhodnutelnost. Následovat bude souhrn jednotlivých rozhodnutelných podtříd predikátové logiky, kterým se budeme v této práci věnovat.

Pro rozhodování budeme potřebovat také vybrat některý z důkazových kalkulů. Tento důkazový kalkul tedy zvolíme a popíšeme si jej.

Výsledný program se neobejde bez datové struktury, se kterou bude program pracovat, proto si potřebnou datovou strukturu definujeme.

Dále se dostaneme k návrhu zadaného algoritmu, kde si specifikujeme požadavky na algoritmus a dle těchto požadavků si patřičný algoritmus sestavíme.

Následovat bude implementace algoritmů pro rozhodnutelné podtřídy a implementace důkazového kalkulu pro rozhodování.

Poslední částí bude ukázka programu a závěr práce.

1.3 Implementační technologie

Tato práce pro implementaci algoritmů využije objektově orientovaného programovacího jazyka C#. Výstupem práce bude webová aplikace postavena na frameworku ASP.NET. Tento framework je open source řešením pro tvorbu dynamických webových stránek od společnosti Microsoft a hlavním jazykem pro vývoj je právě C#.

2. Rozhodnutelné podtřídy predikátové logiky 1. řádu

V této kapitole práce čerpá z *Logiky pro informatiky* [2] a z předchozí diplomové práce [1]. Účelem této kapitoly je shrnout základní principy predikátové logiky 1. řádu (PL1) a její rozhodnutelné podtřídy v takovém rozsahu, který budeme potřebovat pro návrh požadovaného rozhodovacího algoritmu.

2.1 Predikátová logika

V této práci se budeme zabývat pouze predikátovou logikou 1. řádu, která formalizuje úsudky o vlastnostech předmětů a vztazích mezi předměty dané předmětné oblasti (univerza). Predikátová logika 1. řádu je zobecněním výrokové logiky, která umožnuje analyzovat věty pouze do úrovně elementárních výroku, jejichž strukturu již dále nezkoumá. Výrokovou logiku můžeme považovat za predikátovou logiku nultého řádu. PL1 však umožnuje navíc analyzovat elementární výroky do úrovně vlastností jednotlivých objektů zájmu (tzv. individuí – prvků univerza diskurzu) a jejich vztahů. Dále pak existují predikátové logiky vyšších řádů, které umožňují navíc analyzovat výroky do úrovně vlastnosti vlastností, vlastnosti funkcí, atd.

2.1.1 Jazyk predikátové logiky 1. řádu

Návrh našeho algoritmu se neobejde bez specifikace jazyka PL1 a proto si jej v této podkapitole definujeme. Jazyk PL1 obsahuje jazyk výrokové logiky, avšak navíc potřebujeme označovat individua z daného universa diskurzu, k tomu nám slouží tzv. termy (konstanty, proměnné a funkční termy) a dále vlastnosti individuí a vztahy mezi individui, k tomu nám slouží tzv. predikátové symboly jako P, Q, atd. Abychom pak mohli mluvit o všech individuích nebo o některých individuích, zavedeme také tzv. kvantifikátory, a to všeobecný $\forall$ (pro všechna) a existenční (pro některá).

Jazyk predikátové logiky:

1) ***Abeceda predikátové logiky*** je tvořena následujícími skupinami symbolů:
   a) Logické symboly
      i) předmětové (individuové) proměnné: $x, y, z, \ldots$ (příp. s indexy)
      ii) symboly pro spojky: $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$
      iii) symboly pro kvantifikátory: $\forall, \exists$
      iv) případně binární predikátový symbol = (predikátová logika s rovností)
   b) *Speciální symboly* (určují specifiku jazyka)
      i) predikátové symboly: $P, Q, R, \ldots$ (příp. s indexy)
      ii) funkční symboly: $f, g, h, \ldots$ (příp. s indexy)
         Ke každému funkčnímu a predikátovému symbolu je přiřazeno nezáporné číslo n ($n \geq 0$), tzv. ***arita***, udávající počet individuových proměnných, které jsou argumenty funkčního symbolu nebo predikátu.
2) ***Gramatika***, která udává, jak tvořit:
   a) ***termy***:
      i) každý symbol proměnné je atomický *term*
      ii) jsou-li $t_1, \ldots, t_n$ ($n \geq 0$) termy a je-li f n-ární funkční symbol, pak výraz $f(t_1, \ldots, t_n)$ je (funkční) *term*; pro $n = 0$ se jedná o nulární funkční symbol, neboli individuovou *konstantu* (značíme $a, b, c \ldots$); pro $n > 0$ se jedná o *složený term.*

iii) jen výrazy dle i) a ii) jsou termy.

b) ***atomické formule***:

i) je-li P n-ární predikátový symbol a jsou-li $t_1, \ldots, t_n$ termy, pak výraz $P(t_1\ldots t_n)$ je *atomická formule*.

ii) jsou-li t_1 a t_2 termy, pak výraz $(t_1 = t_2)$ je *atomická formule*

c) **(*složené*) *formule***:

i) každá atomická formule je *formule*

ii) je-li výraz A formule, pak $\neg A$ je *formule*

iii) jsou-li výrazy A a B formule, pak výrazy $(A \vee B), (A \wedge B), (A \supset B), (A \equiv B)$ jsou *formule.*

iv) je-li x proměnná a A formule, pak výrazy $\forall x\ A$ a $\exists x\ A$ jsou *formule*

v) jen výrazy dle i) – iv) jsou *formule*

Poznámky k jazyku PL1:

1. Jazyk predikátové logiky – jak byl vymezen výše – je jazyk logiky 1. řádu, pro něhož je charakteristické to, že jediný přípustný typ proměnných jsou individuové proměnné. Pouze individuové proměnné lze vázat kvantifikátory. (V logice 2. řádu jsou povoleny i predikátové proměnné.)

2. Definice jazyka umožňuje formulaci speciálního jazyka (určité teorie) konkrétní volbou prvků (predikátových a funkčních konstant) dle bodu 1) b) definice. Pro takový konkrétní jazyk budou platit obecné principy logické a mimo to – v závislosti na specifických vlastnostech (interpretacích) těchto prvků – i principy mimologické, které zadá tvůrce tohoto speciálního jazyka pomocí speciálních axiómů (dané teorie). Je-li arita funkčního symbolu $n = 0$, pak se jedná o individuovou konstantu (značíme a, b, …), která však není pravou (logickou) konstantou, neboť podléhá (jako každý funkční symbol) interpretaci.

3. Zápis formulí můžeme zjednodušit na základě následujících konvencí o vynechávání závorek:

- Elementární formule a formuli nejvyššího řádu netřeba závorkovat (vnější závorky vynecháváme).
- Závorky je možné vynechávat v souladu s následující prioritní stupnicí funktorů: $(\forall, \exists)$, $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$. Každý funktor vlevo od vybraného funktoru váže silněji, než vybraný funktor.
- V případě, že o prioritě vyhodnocení nerozhodnou ani závorky ani prioritní stupnice, vyhodnocujeme formuli zleva doprava.
- Speciálně vzhledem k asociativitě konjunkce a disjunkce, netřeba při zápisu vícečlenných konjunkcí a disjunkcí užívat žádné závorky.
- Vedle závorek (,) lze využívat i závorky [,],{,}.

2.1.2 Volné a vázané proměnné

Při rozhodování se neobejdeme bez substituce, která bude popsána později, proto je potřeba si definovat volné a vázané proměnné dané formule a jejich pravidla:

Výskyt proměnné x ve formuli A je vázaný, jestliže je součástí nějaké podformule $\forall x B(x)$ nebo $\exists x B(x)$ formule A.

Proměnná x je vázaná ve formuli A, má-li v A vázaný výskyt. Výskyt proměnné x ve formuli A, který není vázaný, nazýváme *volný*.

Proměnná x je volná ve formuli A, má-li v A volný výskyt.

Formule, v níž každá proměnná má buď všechny výskyty volné, nebo všechny výskyty vázané, se nazývá *formulí s čistými proměnnými.*

2.1.3 Rozhodnutelnost

O libovolné formuli můžeme říct, že je rozhodnutelná pouze tehdy, existuje-li algoritmus, který dokáže vždy rozhodnout, zda je daná formule tautologie, kontradikce či jen splnitelná. V opačném případě je formule nerozhodnutelná.

Problémem v predikátové logice je, že logická pravdivost formulí je nerozhodnutelná. To znamená, že neexistuje všeobecný algoritmus, který by vždy s absolutní jistotou rozhodnul, zda je daná formule predikátové logiky tautologie, kontradikce nebo jen splnitelná.

Všeobecně proto můžeme říct, že predikátová logika je nerozhodnutelná. Avšak existují velké oblasti, kde tento problém je úspěšně zvládnutelný a těmi jsou dokázané rozhodnutelné podtřídy.

2.2 Prenexní tvar formule

Jelikož většina rozhodnutelných podtříd specifikuje podmínky nad prenexním tvarem formule, je potřeba si tento tvar definovat dříve, než přikročíme ke specifikaci podtříd. Definujeme si, co je to prenexní tvar formule a algoritmus pro převod do tohoto tvaru.

Formule A predikátové logiky je v *prenexním* tvaru, má-li tvar $Q_1 x_1\, Q_2 x_2 \dots Q_n x_n\ \ B$, kde $n \geq 0$ a pro každé i = 1, 2, …, n je Q_i buď všeobecný kvantifikátor $\forall$ nebo existenční $\exists$, $x_1, x_2, \dots . x_n$ jsou navzájem různé individuové proměnné, B je formule utvořená z elementárních formulí pouze užitím výrokových spojek $\neg$, $\wedge$, $\vee$. Výraz $Q_1 x_1\, Q_2 x_2 \dots\ Q_n x_n$ se nazývá *prefix* a B *otevřeným* jádrem formule A v prenexním tvaru.

Každou formuli lze ekvivalentně přepsat do prenexního tvaru, tj. ke každé formuli predikátové logiky A existuje formule A* v prenexním tvaru, která je s formulí A ekvivalentní, tj. $A \Leftrightarrow A^*$.

Algoritmus pro převod do prenexního tvaru:

1) Eliminace spojek $\supset$ a $\equiv$. Použijeme ekvivalence:
$$A \supset B \Leftrightarrow \neg A \vee B,$$
$$A \equiv B \Leftrightarrow (A \supset B) \wedge (B \supset A) \Leftrightarrow (\neg A \vee B) \wedge (\neg B \vee A).$$
2) Převedení formule na tvar s čistými proměnnými
   a) Požijeme následující ekvivalence (náhrady jejich levé strany pravou stranou):
$$(\forall x A \wedge \forall x B) \Leftrightarrow \forall x (A \wedge B),$$
$$(\exists x A \vee \exists x B) \Leftrightarrow \exists x\, (A \vee B).$$
   b) Přejmenování vázaných proměnných tak, aby žádná proměnná nebyla ve formuli současně volná i vázaná a tak, aby všechny proměnné vázané různými kvantifikátory byly navzájem různé. To platí nejenom pro celou formuli, ale i pro každou její podformuli.
3) Vypuštění nadbytečných kvantifikátorů, tj. kvantifikátorů, jejichž dosah působnosti neobsahuje žádný výskyt kvantifikované proměnné.

4) Přenesení všech výskytů spojky negace bezprostředně před elementární formule. Toho lze dosáhnout opakovaným užitím následujících ekvivalencí (náhrady jejich levé strany pravou stranou):

$$\neg\neg A \Leftrightarrow A,$$
$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B,$$
$$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B,$$
$$\neg\forall x A(x) \Leftrightarrow \exists x \neg A(x),$$
$$\neg\exists x A(x) \Leftrightarrow \forall x \neg A(x).$$

5) Přenesení všech kvantifikátorů na začátek formule. Toto lze dosáhnout opakovaným užitím následujících ekvivalencí:

$\forall x A \wedge B \Leftrightarrow \forall x(A \wedge B)$	$\exists x A \vee B \Leftrightarrow \exists x(A \vee B)$	B *neobsahuje volnou* x
$A \wedge \forall x B \Leftrightarrow \forall x(A \wedge B)$	$A \vee \exists x B \Leftrightarrow \exists x\,(A \vee B)$	A *neobsahuje volnou* x
$\exists x A \wedge B \Leftrightarrow \exists x(A \wedge B)$	$\forall x A \vee B \Leftrightarrow \forall x(A \vee B)$	B *neobsahuje volnou* x
$A \wedge \exists x B \Leftrightarrow \exists x(A \wedge B)$	$A \vee \forall x B \Leftrightarrow \forall x(A \vee B)$	A *neobsahuje volnou* x

Aplikaci algoritmu si ukážeme na následujícím příkladu:

1)	$\forall x \left[P(x) \wedge \forall y \exists x \left(\neg Q(x,y) \supset \forall z\, R(a,x,y)\right)\right]$	výchozí formule
2)	$\forall x \left[P(x) \wedge \forall y \exists x \left(Q(x,y) \vee \forall z\, R(a,x,y)\right)\right]$	eliminace $\supset$
3)	$\forall x \left[P(x) \wedge \forall y \exists x_1 \left(Q(x_1,y) \vee \forall z\, R(a,x_1,y)\right)\right]$	přejmenování proměnné
4)	$\forall x \left[P(x) \wedge \forall y \exists x_1 \left(Q(x_1,y) \vee R(a,x_1,y)\right)\right]$	vypuštění nadbytečných kvantifikátorů
5)	$\forall x \forall y \left[P(x) \wedge \exists x_1 \left(Q(x_1,y) \vee R(a,x_1,y)\right)\right]$	přesun kvantifikátorů doleva
6)	$\forall x \forall y \exists x_1 \left[P(x) \wedge \left(Q(x_1,y) \vee R(a,x_1,y)\right)\right]$	přesun kvantifikátorů doleva

Pozn.: Prenexní tvar formule není určen jednoznačně. Konečná podoba prenexní formule závisí na pořadí provádění úprav a na způsobu přejmenování vázaných proměnných. Všechny prenexní tvary jsou však ekvivalentní.

2.3 Rozhodnutelné podtřídy

Tato podkapitola má za úkol shrnout podtřídy, které byly v předešlé práci [1] dokázány. Soustředíme se pouze na specifikaci podmínek těchto tříd, které budou stěžejní pro následný návrh algoritmů.

Základními rozhodnutelnými třídami jsou tzv. *esenciálně konečné*. Jejich formule jsou sestaveny z konečné kolekce atomických formulí. Pro tyto třídy nebudeme navrhovat algoritmus, protože náš program umožní uživateli rozhodnout formuli, přestože nebude vyhovovat níže popsaným rozhodnutelným podtřídám.

V předešlé práci bylo dokázáno celkem sedm podtříd, ve kterých je predikátová logika 1. řádu rozhodnutelná. U každé z těchto tříd si pro lepší pochopení uvedeme příklad, kdy formule vyhovuje podmínkám třídy a příklad, kdy formule nevyhovuje podmínkám. Pokud se podmínka třídy bude týkat formule v prenexním tvaru, budeme příklady uvádět v tomto tvaru.

2.2.1 Löb-Gurevichova třída

Löb-Gurevichova třída je třída všech těch formulí, které obsahují pouze monadické predikátové symboly. Jedná se tak tedy o monadickou predikátovou logiku, u které je problém logické pravdivosti formulí rozhodnutelný.

Příklad formule splňující podmínku:

- $\forall x \left[P(x) \wedge \forall y \left(\neg Q(y) \supset \forall z\, R(z)\right)\right]$

Příklad formule nesplňujících podmínku:

- $\forall x \left[P(x) \wedge \forall y \exists x \left(\neg Q(x, f(z)) \supset \forall z\, R(a, x, y)\right)\right]$

2.3.2 Gödel-Kalmár-Schütteova třída

Gödel-Kalmár-Schütteova třída je třída všech těch formulí, které v prenexní normální formě mají prefix ve tvaru $\exists^* \forall^2 \exists^*$ a žádné funkční symboly.

Prefixový tvar třídy:

$$\exists x_1 \dots \exists x_n \forall y_1 \forall y_2 \exists z_1 \dots \exists z_m \varphi$$

Příklad formule splňující podmínky:

- $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \exists z_1 \exists z_2 \exists z_3 \left[\left(P(x_1, x_2) \wedge \left(Q(y_1, x_1) \vee R(y_2, z_1)\right)\right) \vee S(z_2, z_3)\right]$

Příklad formule nesplňující podmínky:

- $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \exists z_1 \exists z_2 \forall z_3 \left[\left(P(x_1, x_2) \wedge \left(Q(f(y_1), x_1) \vee R(y_2, z_1)\right)\right) \vee S(z_2, z_3)\right]$

2.3.3 Bernays-Schönfinkelova třída

Jedná se o třídu všech formulí, které v prenexní normální formě mají prefix ve tvaru $\exists^* \forall^*$. Dále formule neobsahuje žádné funkční symboly.

Prefixový tvar třídy:

$$\exists x_1 \dots \exists x_n \forall y_1 \dots \forall y_m \varphi$$

Příklad formule splňující podmínky:

- $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \forall y_3 \left[\left(P(x_1, x_2) \wedge \left(Q(y_1, y_2) \vee R(y_2, y_3)\right)\right) \vee S(x_1, y_2)\right]$

Příklad formule nesplňující podmínky:

- $\exists x_1 \exists x_2 \forall y_1 \exists y_2 \left[P(x_1, x_2) \wedge \left(Q(y_1, y_2) \vee R(g(x_1))\right)\right]$

2.3.4 Gurevich-Maslov-Orevkovova třída

Gurevich-Maslov-Orevkovova třída je třída všech těch formulí predikátové logiky prvního řádu bez rovnosti, jejichž prefix má v prenexní normální formě tvar $\exists^*\forall\exists^*$.

Prefixový tvar třídy:

$$\exists x_1 \dots \exists x_n \forall y \exists z_1 \dots \exists z_m \varphi$$

Příklady formulí splňující podmínky:

- $\exists x_1 \exists x_2 \forall y_1 \exists y_2 \exists y_3 \left[\left(P(x_1, x_2) \wedge \left(Q(y_1, y_2) \vee R(y_2, y_3)\right)\right) \vee S(x_1, f(y_2, x_2))\right]$

Příklady formulí nesplňující podmínky:

- $\exists x_1 \exists x_2 \forall y_1 \forall y_2 \exists y_3 \left[\left(P(x_1, x_2) \wedge \left(Q(y_1, y_2) \vee R(y_2, y_3)\right)\right) \vee S(x_1, f(y_2, x_2))\right]$

2.3.5 Gurevichova třída

Gurevichova třída je třída všech těch formulí predikátové logiky s rovností, které v prenexní normální formě mají prefix ve tvaru $\exists^*$. Formule dále musí obsahovat buďto dva funkční symboly nebo funkci s aritou větší než jedna.

Prefixový tvar třídy:

$$\exists x_1 \dots \exists x_n \varphi$$

Příklad formule splňující podmínky:

- $\exists x_1 \exists x_2 \left[P(f(x_1, x_2)) = z \wedge Q(x_3))\right]$

Příklad formule nesplňující podmínky:

- $\exists x_1 \forall x_2 \left[P(f(x_1, x_2)) = z \wedge Q(x_3))\right]$

2.3.6 Rabinova třída

Rabinova třída patří mezi třídy formulí predikátové logiky s rovností, která obsahuje jednu unární funkci a pouze monadické predikáty. Zároveň u této třídy neplatí žádné omezení, co se týče prefixu formule.

Příklad formule splňující podmínky:

- $\exists x_1 \forall x_2 \left[P(f(x_1)) = z \wedge Q(x_2))\right]$

Příklad formule nesplňující podmínky:

- $\forall x_1 \exists x_2 \exists x_3 \left[P(f(x_1), x_2) = z \wedge Q(x_3))\right]$

2.3.7 Shelahova třída

Shelahova třída je standardní fragment predikátové logiky prvního řádu s rovností. Jedná se omezení, kdy prefix formule φ v prenexní formě má tvar $\exists^*\forall\exists^*$. Formule neobsahuje žádné funkční symboly s aritou n ≥ 2. Formule dále obsahuje nejvýše jeden unární funkční symbol.

Prefixový tvar třídy:

$$\exists x_1 \dots \exists x_n \forall y_1 \exists z_1 \dots \exists z_m \varphi$$

Příklad formule splňující podmínky:

- $\exists x_1 \forall x_2 \, [P(x_1) = z \wedge Q(x_2))]$

Příklad formule nesplňující podmínky:

- $\exists x_1 \exists x_2 \, [P(x_1) = z \wedge Q(x_2))]$

3. Důkazový kalkul

Stěžejní částí této práce je implementace důkazového kalkulu, proto musíme nějaký kalkul zvolit. Volíme obecnou rezoluční metodu, protože je jednou z velice efektivních metod a její vlastnosti jsou výbornými pro implementaci v programovacím jazyce. Používá se v logickém programování a je základem pro programovací jazyk Prolog.

Rezoluční metoda je zobecnění stejnojmenné metody ve výrokové logice, ale díky bohatší vnitřní struktuře formulí predikátové logiky je složitější. Rezoluční metoda parciálně rozhoduje, zda daná formule PL1 je nesplnitelná. Pro předloženou formuli A, která nesplnitelná je, procedura to zjistí a po konečném počtu kroků vydá kladnou odpověď. V případě, že A je splnitelná, proces nemusí nikdy skončit.

Rezoluční metodu lze aplikovat pouze na formuli v tzv. *klauzulární (Skolemově) formě*, proto si nejprve ukážeme algoritmus pro převedení do této formy. Jelikož je uplatnění rezolučního pravidla v predikátové logice komplikováno tím, že se v literálech (atomická formule nebo její negace) vyskytují obecně termy různého tvaru, popíšeme si unifikační algoritmus, který umožní tyto termy unifikovat. Následně shrneme celou rezoluční metodu.

3.1 Skolemova forma

Skolemova klauzulární forma dané formule je formule v konjunktivní normální formě, která má v prefixu pouze všeobecné kvantifikátory a matice formule je konjunkce *klauzulí*, kde klauzule je disjunkcí literálů.

Algoritmus pro převod do Skolemovy klauzulární formy:

1) Utvoření existenčního uzávěru formule A. (Krok zachovává splnitelnost)
2) Eliminace nadbytečných kvantifikátorů. (ekvivalentní krok)
   Z formule A vypustíme všechny kvantifikátory $\forall x_i$, $\exists x_i$ v jejichž dosahu se nevyskytuje proměnná x_i.
3) Přejmenování proměnných tak, aby každý kvantifikátor kvantifikoval přes jinou proměnnou. (ekvivalentní krok)
   Přejmenujeme tedy všechny proměnné, které jsou v A kvantifikovány více než jednou tak, aby všechny kvantifikátory měly ve svém dosahu navzájem různé proměnné.
4) Eliminace spojek ⊃ a ≡ podle těchto vztahů (ekvivalentní krok):
   $$(A \supset B) \Leftrightarrow (\neg A \vee B), (A \equiv B) \Leftrightarrow (A \supset B) \wedge (B \supset A) \Leftrightarrow (\neg A \vee B) \wedge (\neg B \vee A).$$
5) Přesun spojek ¬ dovnitř dle de Morganových zákonů (ekvivalentní krok)
6) Přesun kvantifikátorů doprava. (ekvivalentní krok)
   Provádíme náhrady podle těchto ekvivalencí (Q je kvantifikátor ∀ nebo ∃; © je symbol ∧ nebo ∨; A, B neobsahují volnou proměnnou x):
   $$Qx\,(A © B(x)) \Leftrightarrow A © QxB(x), Qx(A(x) © B) \Leftrightarrow Qx\,A(x) © B$$
7) Eliminace existenčních kvantifikátorů (krok zachovává splnitelnost)
   Provádíme postupně Skolemizaci podformulí $QxB(x), QxA(x)$, které jsme obdrželi v předchozím kroku 6, tedy náhradu existenčně kvantifikovaných formulí formulemi bez existenčního kvantifikátoru)

8) Přesun všeobecných kvantifikátorů doleva (Ekvivalentní krok, neboť jsme již provedli krok 3. a platí ekvivalence dle 6.)
9) Použití distributivních zákonů. (ekvivalentní krok)

$$(A \wedge B) \vee C \Leftrightarrow (A \vee C) \wedge (B \vee C), A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C),$$

Věta (Skolem):

Každá formule A může být převedena na formuli A^S v klauzurální (Skolemově) formě takovou, že A je splnitelná, právě když A^S je splnitelná.

Algoritmus demonstrujeme na příkladu:

$$A = \forall x \left\{P(x) \supset \exists z\{\neg\forall y[Q(x,y) \supset P(f(x_1))] \wedge \forall y[Q(x,y) \supset P(x)]\}\right\}$$

Kroky 1. a 2. Utvoření existenčního uzávěru a eliminace $\exists z$:

$$\exists x_1 \forall x \left\{P(x) \supset \{\neg\forall y[Q(x,y) \supset P(f(x_1))] \wedge \forall y[Q(x,y) \supset P(x)]\}\right\}$$

Krok 3. Přejmenování proměnné y:

$$\exists x_1 \forall x \left\{P(x) \supset \{\neg\forall y[Q(x,y) \supset P(f(x_1))] \wedge \forall z[Q(x,z) \supset P(x)]\}\right\}$$

Krok 4. Eliminace $\supset$:

$$\exists x_1 \forall x \left\{\neg P(x) \vee \{\neg\forall y[\neg Q(x,y) \vee P(f(x_1))] \wedge \forall z[\neg Q(x,z) \vee P(x)]\}\right\}$$

Krok 5. Přesun negace dovnitř:

$$\exists x_1 \forall x \left\{\neg P(x) \vee \{\exists y[Q(x,y) \wedge \neg P(f(x_1))] \wedge \forall z[\neg Q(x,z) \vee P(x)]\}\right\}$$

Krok 6. Přesun kvantifikátorů $\exists y$ a $\forall z$ doprava:

$$\exists x_1 \forall x \left\{\neg P(x) \vee \{[\exists y Q(x,y) \wedge \neg P(f(x_1))] \wedge [\forall z \neg Q(x,z) \vee P(x)]\}\right\}$$

Krok 7. Eliminace existenčních kvantifikátorů:

$$\forall x \left\{\neg P(x) \vee \{[Q(x,h(x)) \wedge \neg P(f(a))] \wedge [\forall z \neg Q(x,z) \vee P(x)]\}\right\}$$

Krok 8. Přesun $\forall z$ doleva:

$$\forall x \forall z \left\{\neg P(x) \vee \{[Q(x,h(x)) \wedge \neg P(f(a))] \wedge [\neg Q(x,z) \vee P(x)]\}\right\}$$

Krok 9. Použití distribučního zákona:

$$\forall x \forall z\{[\neg P(x) \vee Q(x,h(x))] \wedge [\neg P(x) \wedge \neg P(f(a))] \wedge [\neg P(x) \vee \neg Q(x,z) \vee P(x)]\}$$

Formuli můžeme ještě upravit:

1) Vypustíme třetí klauzuli, protože je to tautologie
2) Odstraníme kvantifikátor $\forall z$ (stal se zbytečným)
3) Ve druhé klausuli odstraníme $\neg P(x)$, neovlivníme tím splnitelnost

Výsledná Skolemova klauzulární forma je:

$$A^S = \forall x\{[\neg P(x) \vee Q(x,h(x))] \wedge \neg P(f(a))\}$$

3.2 Rezoluční metoda

Rezoluční metodu nejlépe demonstrujeme přímo na příkladu, kdy uvažujeme formuli A ve Skolemově klauzulární formě:

$$\forall x \forall y \forall z \forall v [P(x, f(x)) \wedge Q(y, h(y)) \wedge (\neg P(a, z) \vee \neg Q(z, v)]$$

Dokážeme, že je tato formule nesplnitelná. Vypíšeme jednotlivé klausule pod sebe a pokusíme se uplatňovat pravidlo rezoluce:

1. $P(x, f(x))$
2. $Q(y, h(y))$
3. $\neg P(a, z) \vee \neg Q(z, v)$

Klausule 1 a 3 obsahují literály s opačným znaménkem, avšak uplatnění rezoluce brání to, že $P(x, f(x)) \neq P(a, z)$. Uvědomíme-li si však, že všechny proměnné jsou univerzálně kvantifikovány a že platí zákon konkretizace (je-li term *t* substituovatelný za proměnnou *x* ve formuli A(*x*), pak $\forall x \mathrm{A} x \vdash \mathrm{A}(x / t)$, „co platí pro všechny, platí i pro *t*“), můžeme se pokusit najít vhodnou substituci termů za proměnné tak, abychom dostali shodné, tj. *unifikované* literály. V našem případě taková substituce existuje:

$$x/a,\ z/f(a)$$

Po provedení této substituce dostaneme klausule:

1. $P(a, f(a))$
2. $Q(y, h(y))$
3. $\neg P(a, f(a)) \vee \neg Q(f(a), v)$
   kde na 1. a 3. již lze uplatnit pravidlo rezoluce:
4. $\neg Q(f(a), v)$
   Abychom nyní mohli rezolvovat klausule 2 a 4, zvolíme opět substituci:

$$y/f(a),\ v/h(f(a))$$

Dostaneme:

5. $Q(f(a), h(f(a))$
6. $\neg Q(f(a), h(f(a))$

Rezolucí těchto klausulí obdržíme prázdnou klausuli #. Tedy formule A je nesplnitelná.

V tomto příkladu jsme se opřeli o zákon konkretizace, tedy postup byl korektní. Substituce jsme však hledali intuitivně. Pro automatizaci existují tzv. *unifikační algoritmy*. V naší práci budeme využívat Robinsonův unifikační algoritmus.

3.3 Robinsonův unifikační algoritmus

Formulace zcela obecného algoritmu je poměrně složitá (patří do výpočetních metod umělé inteligence) a jeho „ruční“ simulace značně nepřehledná. Omezíme se proto pouze na případ, kdy unifikované elementární formule nemají na obou místech stejnolehlých argumentů současně nějaké složené termy (v tomto případě by bylo třeba rekurzivním algoritmem postupně tyto struktury rozkrývat).

Předpokládejme tedy

$$\mathrm{A} = P(t_1, t_2, \dots\ t_n), \mathrm{B} = P(s_1, s_2, \dots s_n),$$

kde $t_1, t_2, \dots\ t_n$, $s_1, s_2, \dots s_n$ jsou termy takové, že t_i, s_i nejsou současně složené termy, tedy alespoň jeden z těchto termů je proměnná. Potom nejobecnější unifikaci získáme takto:

1. Pro i = 1,2…,n prováděj:
   - Je-li $t_i = s_i$, pak polož $\sigma_i = \varepsilon$
   - Není-li $t_i = s_i$, pak zjisti, zda jeden z termů t_i, s_i představuje nějakou individuovou proměnnou x a druhý nějaký term r, který proměnnou x neobsahuje
     - Jestliže ano, pak polož $\sigma_i = \{x/r\}$.
     - Jestliže ne, pak ukonči práci s tím, že formule A, B nejsou unifikovatelné
2. Po řádném dokončení cyklu urči $\sigma = \sigma_1\sigma_2 \dots \sigma_n$. Substituce σ je nejobecnější unifikací formulí A, B.

Příklad:

$\mathrm{A} = P(x, f(x), u)$, $\mathrm{B} = P\big(y, z, g(x, y)\big)$

- $\sigma_1 = \{x/y\},\ A\sigma_1 = P(y, f(y), u), B\sigma_1 = P\big(y, z, g(y, y)\big)$
- $\sigma_2 = \{z/f(y)\},\ A\sigma_1\sigma_2 = P(y, f(y), u), B\sigma_1\sigma_2 = P\big(y, f(y), g(y, y)\big)$
- $\sigma_3 = \{u/g(y, y)\},\ A\sigma_1\sigma_2\sigma_3 = P(y, f(y), g(y, y)), B\sigma_1\sigma_2\sigma_3 = P\big(y, f(y), g(y, y)\big)$

Složená substituce $\sigma = \sigma_1\sigma_2\sigma_3$ je unifikací formulí A, B ($\mathrm{A}\sigma = P\big(y, f(y), g(y, y)\big) = \mathrm{B}\sigma$), a to nejobecnější unifikací.

4. Datová struktura

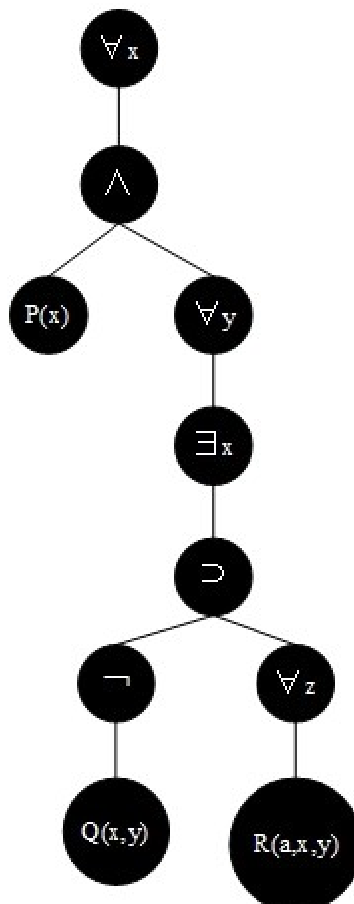
Abychom mohli se vstupní formulí pracovat, musíme ji formulovat jako určitou datovou strukturu. Jelikož každá výroková formule je reprezentována pomocí grafického útvaru nazývaného syntaktický strom, lze také predikátovou formuli reprezentovat syntaktickým stromem rozšířeným o konstrukty, kterými predikátová logika rozšiřuje logiku výrokovou.

V této podkapitole popíšeme, co je to syntaktický strom a jak takový strom vypadá. Následně specifikujeme algoritmus pro převod vstupní formule na syntaktický strom.

Stromem rozumíme souvislý graf, který neobsahuje kružnice. Strom je v informatice často využívanou datovou strukturou, jejímž základním konceptem je hierarchie. Stromy se využívají v případech, kdy potřebujeme rychle vyhledávat a reprezentovat strukturovaná data. Strom je složen z uzlů, které jsou vzájemně spojeny hranami. Rozlišujeme tři typy uzlů, kterými jsou kořen, vnitřní uzel a koncový uzel (tzv. list stromu).

Syntaktickým stromem v našem případě budeme rozumět strom, kde všechny jeho vnitřní uzly budou obsahovat podformule a listy stromu budou pouze atomické formule. Jako příklad je na následujícím obrázku zobrazen syntaktický strom formule:

$$\forall x \left[P(x) \land \forall y \exists x \left(\neg Q(x, y) \supset \forall z\, R(a, x, y) \right) \right].$$



Obrázek č.1: Syntaktický strom formule

5. Návrh algoritmu

V této kapitole se již dostáváme k návrhu algoritmu, který je cílem této práce. Nejprve popíšeme, co to algoritmus vlastně je. Poté specifikujeme požadavky na tento algoritmus a podle těchto požadavků algoritmus navrhneme.

5.1 Algoritmus

Algoritmus chápeme jako předpis pro řešení nějakého problému. Jako příklad lze uvést předpis pro konstrukci trojúhelníka pomocí kružítka a pravítka ze tří daných prvků. Pokud rozebereme řešení této úlohy podrobněji, musí obsahovat tři věci [6]:

1. hodnoty vstupních dat (tři prvky trojúhelníka)
2. předpis pro řešení
3. požadovaný výsledek, tj. výstupní data (výsledný trojúhelník)

Algoritmus je tedy konečná posloupnost přesně definovaných instrukcí, které ze vstupu vedou k požadovanému výstupu [6].

Po navrhnutí algoritmu je potřeba tento algoritmus zapsat takovým způsobem, aby mu ten, kdo jej bude provádět, rozuměl. Algoritmus může být zapsán přirozeným jazykem, strukturovaným jazykem, formální logickou specifikací, programovacím jazykem a grafickým zobrazením.

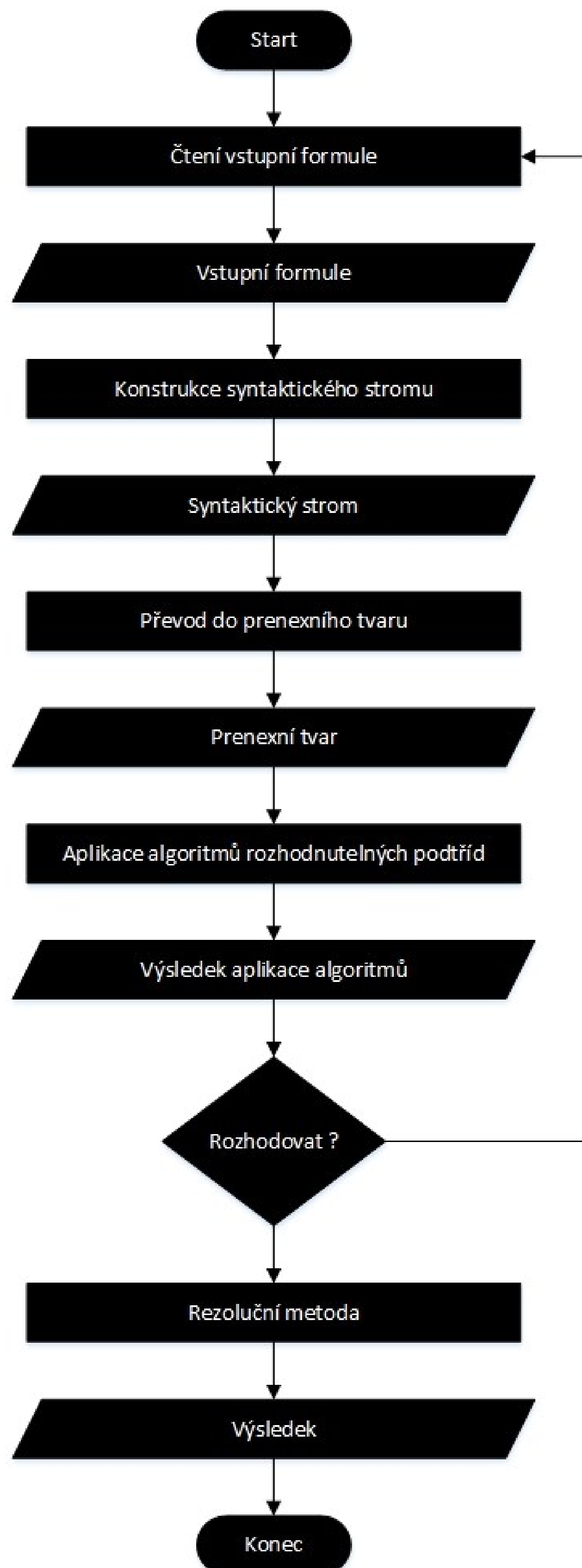
5.2 Algoritmus pro rozhodování ve vybraných podtřídách PL1

Cílem našeho algoritmu je na základě vstupní formule rozhodnout, do které rozhodnutelné podtřídy formule patří (pokud vůbec nějaké) a poté tuto formuli za pomoci důkazového kalkulu rozhodnout.

Na základě předešlých definic a zadání můžeme náš algoritmus popsat takto:

1. Umožní uživateli zadat vstupní formuli.
2. Vstupní formuli přečte a uloží do datové struktury (syntaktického stromu) pro zpracování.
3. Jelikož z předešlých definic pro rozhodnutelné podtřídy víme, že většina těchto podtříd specifikuje své podmínky nad formulí v prenexním tvaru, bude potřeba syntaktický strom převést do tohoto tvaru.
4. Aplikuje algoritmy rozhodnutelných podtříd a rozhodne, zda do některé patří nebo ne.
5. Zobrazí uživateli seznam rozhodnutelných podtříd, do kterých formule patří.
6. Po zobrazení seznamu umožní uživateli tuto formuli rozhodnout pomocí důkazového kalkulu. Pokud formule nepatří do žádné z podtříd, zobrazí uživateli výstražnou hlášku, že program může skončit chybou (algoritmus se zacyklí).

Algoritmus jsme si popsali slovně a na následujícím obrázku je algoritmus popsán v grafické podobě.



Obrázek č. 2: Algoritmus pro rozhodování ve vybraných podtřídách PL1

6. Implementace algoritmů rozhodnutelných podtříd

V minulé kapitole jsme navrhli algoritmus a v této kapitole se již dostaneme k jeho implementaci. Postupně si projdeme jednotlivé části algoritmu, popíšeme je podrobněji a následně implementujeme.

Jak již bylo zmíněno v úvodu, pro implementaci využijeme objektově orientovaný programovací jazyk C#. Jelikož v následujících kapitolách se budeme věnovat implementaci a popisy algoritmů budou právě výpisem z kódu, popíšeme si nyní, jak vypadají objekty, se kterými v implementační části pracujeme.

Prvním objektem je *Term*, který má tyto vlastnosti:

- **Symbol** – obsahuje symbol proměnné, funkce nebo konstanty (a, b, c, f, g, h, x, y, z)
- **Index** – reprezentuje index termu
- **TermType** – enum s typem termu (konstanta, proměnná nebo funkce)
- **Terms** – jedná se o kolekci termů, využíváme u funkčních termů

Dalším objektem je *AtomicFormula*, který má tyto vlastnosti:

- **Symbol** – obsahuje predikátový symbol (P, Q, R)
- **Index** – reprezentuje index predikátu
- **IsNegated** – příznak, jestli je formule negovaná
- **Terms** – jedná se o kolekci termů predikátu

Nejdůležitějším objektem je *Node* (uzel), který má tyto vlastnosti:

- **Operator** – jedná se o *enum*, který reprezentuje obsah vnitřních uzlů, může obsahovat logickou spojku, kvantifikátor nebo hodnotu nazvanou *Leaf*, která nám určuje, že se jedná o list stromu, tedy atomickou formuli.
- **Left** – jedná se o objekt typu *Node*, který reprezentuje levého potomka uzlu
- **Right** – také objekt typu *Node*, který reprezentuje pravého potomka uzlu
- **Formula** – objekt typu *AtomicFormula*, reprezentující atomickou formuli uzlu
- **Term** – typu Term, který reprezentuje proměnnou, kterou váže kvantifikátor (nabývá-li *Operator* hodnoty *ForAll* nebo *Exists*)

6.1 Čtení vstupní formule

Vstupní formulí pro algoritmus je formule predikátové logiky, reprezentována znakovým řetězcem. Pro její načtení v naší aplikaci bude sloužit klasický HTML textový input.

Příklad vstupní formule:

$$\forall x \left[P(x) \land \forall y \exists x \left(\neg Q(x,y) \supset \forall z\, R(a,x,y) \right) \right]$$

Pro náš program si zjednodušíme jazyk predikátové logiky takto:

- Předmětové proměnné budeme uvažovat pouze: x, y, z (případně s indexy)
- Predikátové symboly budeme uvažovat pouze: P, Q, R (případně s indexy)
- Funkční symboly budeme uvažovat pouze: f, g, h (případně s indexy)

Tato drobná úprava nám zjednoduší algoritmus pro čtení vstupní formule a konstrukci stromu. Díky indexaci se nejedná se o významné omezení.

6.2 Konstrukce syntaktického stromu

Dříve jsme si definovali datovou strukturu, kterou bude v našem případě syntaktický strom, a nyní si ukážeme, jak syntaktický strom sestavit. Pro sestavení stromu nám bude sloužit algoritmus, jehož funkcionalitu si nejprve představíme na příkladu.

Příkladem nám bude formule z předešlého příkladu:

$$\forall x\,\left[P(x)\ \land\ \forall y \exists x\,\left(\neg Q(x,y)\ \supset\ \forall z\, R(a,x,y)\right)\right]$$

1. Přečteme první znak, kterým je všeobecný kvantifikátor následovaný proměnnou x, a protože dále následuje závorka, hledáme její uzavírající závorku. Uzavírající závorka je zároveň posledním znakem ve formuli, proto volíme do kořenového uzlu stromu všeobecný kvantifikátor s danou proměnnou x a vytvoříme zárodek syntaktického stromu.

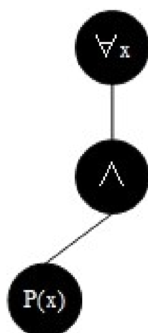
Obrázek č. 3: kořen stromu

2. Dále se rekurzivně snažíme vytvořit syntaktické stromy pro potomky, které kořen váže. Jelikož se jedná o kvantifikátor, bude mít pouze jednoho potomka, kterým je syntaktický strom vycházející z podformule $P(x)\ \land\ \forall y \exists x\,\left(\neg Q(x,y)\ \supset\ \forall z\, R(a,x,y)\right)$.

3. Přečteme první znak podformule potomka, zjistíme, že se jedná o predikátový symbol. V tomto případě hledáme následující logickou spojku, která tento predikát váže. Nalezneme spojku $\land$ a touto spojkou osadíme uzel.

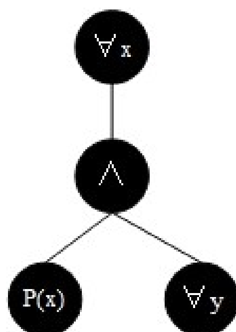
Obrázek č. 4: konstrukce stromu po 3. kroku

4. Dále pokračujeme rekurzivně levou stranou podformule $P(x)$. Jedná se o atomickou formuli, jde o list stromu.



Obrázek č.5: konstrukce stromu po 4. kroku

6. Následně se rekurzí vrátíme k pravému potomkovi a konstruujeme syntaktický strom pro formuli $\forall y \exists x \left(\neg Q(x,y) \ \supset \ \forall z \, R(a,x,y) \right)$.



Obrázek č. 6: konstrukce stromu po 6. kroku

Narazili jsme opět na kvantifikátor a postup je tedy stejný jako v prvním bodě. Algoritmus se zastavuje až úspěšně sestrojeným syntaktickým stromem, který již byl popsán na obrázku číslo 1.

6.3 Převod do prenexního tvaru

Nyní je potřeba sestavený strom převézt do prenexního tvaru. Postupně projdeme všechny kroky algoritmu pro převod do prenexního tvaru a ukážeme si jejich konkrétní implementace.

1. Eliminace spojek ⊃ a ≡ :

V tomto kroku budeme procházet uzly stromu a nahrazovat podstromy užitím těchto ekvivalencí:

$$A \supset B \iff \neg A \vee B,$$
$$A \equiv B \iff (A \supset B) \wedge (B \supset A) \iff (\neg A \vee B) \wedge (\neg B \vee A).$$

Implementace tohoto kroku vypadá následovně:

```
public Node PrenexStep1(Node node)
{
  if (node.IsLeaf())
     return node;

  Node prenexLeft = null, prenexRight = null;
  if (node.Left != null)
      prenexLeft = PrenexStep1(node.Left);
  if (node.Right != null)
      prenexRight = PrenexStep1(node.Right);

  if (node.Operator == Operator.Implication)
  {
     node.Operator = Operator.Or;
     prenexLeft = Node.CreateNot(prenexLeft);
  }

  if (node.Operator == Operator. Biconditional)
  {
     node.Operator = Operator.And;
     prenexLeft = Node.CreateNode(Operator.Or, Node.CreateNot(prenexLeft), prenexRight);
     prenexRight = Node.CreateNode(Operator.Or, prenexLeft, Node.CreateNot(prenexRight));
  }

  node.Left = prenexLeft;
  node.Right = prenexRight;
  return node;
}
```

Výpis č. 1 – Eliminace spojek ⊃ a ≡

Uzly stromu nahrazujeme od jeho listů, proto se v první části implementace zanoříme rekurzivně nejhlouběji a následně při jejím postupném vynořování kontrolujeme spojky. Při implikaci nejprve nastavíme symbol na disjunkci, vytvoříme nový uzel s negací, jehož potomkem bude levý potomek původního uzlu. V případě ekvivalence nastavíme funktor na konjunkci a vytvoříme nové potomky. Oba potomci budou mít funktor konjunkci. Nakonec přidáme uzly s negací nad potomky dle patřičné ekvivalence.

2. Převedení formule na tvar s čistými proměnnými:

V první části druhého aplikujeme tyto ekvivalence:

$$(\forall x A \ \wedge \ \forall x B) \Leftrightarrow \ \forall x(A \ \wedge B),$$
$$(\exists x A \ \vee \ \exists x B) \Leftrightarrow \ \exists x \ (A \ \vee B).$$

V tomto případě budeme postupovat obdobně jako v předchozím kroku. Rekurzí se zanoříme nejhlouběji do stromu a při její opuštění kontrolujeme uzly. V případě první ekvivalence kontrolujeme, zda se jedná o funktor konjunkce a zároveň oba jeho potomci jsou všeobecnými kvantifikátory se stejnými termy. Pokud ano, přesuneme kvantifikátor z potomků nad aktuální uzel. Obdobně postupujeme také v případě druhé ekvivalence.

```
public Node PrenexStep2A(Node node)
```

```
{
   if (node.IsLeaf())
      return node;

   Node prenexLeft = null, prenexRight = null;
   if (node.Left != null) prenexLeft = PrenexStep2A(node.Left);
   if (node.Right != null) prenexRight = PrenexStep2A(node.Right);

   if (node.Operator == Operator.And &&
      node.Left.Operator == Operator.ForAll && node.Right.Operator == Operator.ForAll &&
      node.Left.Term.Equals(node.Right.Term))
   {
      node.Operator = Operator.ForAll;
      node.Term = node.Left.Term;
      node.Left.Operator = Operator.And;
      node.Left.Left = node.Left.Left;
      node.Left.Right = node.Right.Left;

   }

   if (node.Operator == Operator.Or &&
      node.Left.Operator == Operator.Exists && node.Right.Operator == Operator.Exists &&
      node.Left.Term.Equals(node.Right.Term))
   {
      node.Operator = Operator.Exists;
      node.Term = node.Left.Term;
      node.Left.Operator = Operator.Or;
      node.Left.Left = node.Left.Left;
      node.Left.Right = node.Right.Left;
   }
   return node;
}
```

Výpis č. 2 – Aplikace ekvivalencí druhého kroku prenexního algoritmu

Ve druhé části tohoto kroku musíme implementovat algoritmus pro:

Přejmenování vázaných proměnných tak, aby žádná proměnná nebyla ve formuli současně volná i vázaná a tak, aby všechny proměnné vázané různými kvantifikátory byly navzájem různé. To platí nejenom pro celou formuli, ale i pro každou její podformuli.

Implementaci si nejprve ukážeme a následně popíšeme:

```
public Node PrenexStep2B(Node node)
{
   if (quantifiredTerms == null) quantifiredTerms = new List<Term>();
   if (exchangeTerms == null) exchangeTerms = new Dictionary<string, int>();

   if (node.IsLeaf())
   {
      foreach (var item in node.Formula.Terms)
      {
         if (exchangeTerms.ContainsKey(item.ToString()))
         {
            item.Index = exchangeTerms[item.ToString()];
```

```
            }
        }

        return node;
    }

    if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
    {
        if (!quantifiredTerms.Contains(node.Term))
        {
            quantifiredTerms.Add(node.Term);
        }
        else
        {
            var newTerm = new Term(node.Term);
            while(quantifiredTerms.Contains(newTerm))
            {
                newTerm.Index++;
            }

            node.Term = newTerm;
            exchangeTerms.Add(node.Term.ToString(), newTerm.Index);
        }
    }

    if (node.Left != null)       node.Left = PrenexStep2B(node.Left);
    if (node.Right != null)     node.Right = PrenexStep2B(node.Right);
    }

    if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
    {
        if (quantifiredTerms.Contains(node.Term))
            quantifiredTerms.Remove(node.Term);
        if(exchangeTerms.ContainsKey(node.Term.ToString())
          exchangeTerms.Remove(node.Term.ToString();
    }

    return node;
}
```

Výpis č. 3 – Převod formule na tvar s čistými proměnnými

V metodě využíváme kolekci pro již kvantifikované termy nazvanou *quantifiredTerms*. Dále využíváme slovník *exchangeTerms* pro nahrazování proměnných. Metoda prochází strom od kořene, pokud narazíme na kvantifikátor, přidáme jej do kolekce kvantifikovaných proměnných, pokud už v kolekci je obsažena, vytvoříme novou proměnnou a umístíme ji do slovníku pro nahrazení. V listech stromu poté kontrolujeme, jestli je proměnná ve slovníku, pokud ano, nahradíme ji novou proměnnou. Při výstupu z rekurze nesmíme zapomenout z kolekce a slovníku odebrat termy, protože náhrady platí pouze v podformulích.

3. Vypuštění nadbytečných kvantifikátorů:

```
private List<Term> usedTerms = new List<Term>();

public Node PrenexStep3(Node node)
{
   if (node.IsLeaf())
   {
      FillListOfTerms(node);
      return node;
   }

   Node prenexLeft = null, prenexRight = null;
   if (node.Left != null)
      prenexLeft = PrenexStep3(node.Left);
   if (node.Right != null)
      prenexRight = PrenexStep3(node.Right);

   if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
   {
      if (!usedTerms.Contains(node.Term))
      {
         return prenexLeft;
      }
   }

   node.Left = prenexLeft;
   node.Right = prenexRight;
   return node;
}
```

Výpis č. 4 – Vypuštění nadbytečných kvantifikátorů

Jak je možno vidět na výpisu výše, opět se zanoříme nejhlouběji do stromu a při vynořování hledáme kvantifikátory. Rozdílem v tomto případě je, že při dosažení listu stromu si plníme kolekci nazvanou *usedTerms* termy, které obsahuje list (atomická formule). K naplnění slouží pomocná metoda *FillListOfTerms*, která přijme list, projde všechny termy a přidá je do kolekce. Při vynořování z rekurze známe termy, které jsou obsaženy v atomických formulích. Při nalezení kvantifikátoru testujeme jeho term, pokud kolekce neobsahuje term, tento kvantifikátor vypouštíme.

4. Přenesení všech výskytů spojky negace bezprostředně před elementární formule:

Tentokrát budeme postupovat odlišně, než v předešlých případech. Negace se má postupně propadat stromem až před listy a proto budeme uzly procházet od kořene, tudíž před zanořením do rekurze. Pro každý nalezený uzel negace aplikujeme na jeho potomka tyto ekvivalence:

$$\neg\neg A \Leftrightarrow A,$$
$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B, \quad (de\ Morgan)$$
$$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B, \quad (de\ Morgan)$$
$$\neg\forall x A(x) \Leftrightarrow \exists x \neg A(x), \quad (de\ Morgan)$$
$$\neg\exists x A(x) \Leftrightarrow \forall x \neg A(x). \quad (de\ Morgan)$$

```
public Node PrenexStep4(Node node)
{
```

```
  if (node.IsLeaf())
    return node;

  if (node.Operator == Operator.Not)
  {

    switch (node.Left.Operator)
    {
      case Operator.And:
        node.Operator = Operator.Or;
        node.Right = Node.CreateNot(node.Left.Right);
        node.Left = Node.CreateNot(node.Left.Left);
        break;
      case Operator.Or:
        node.Operator = Operator.And;
        node.Right = Node.CreateNot(node.Left.Right);
        node.Left = Node.CreateNot(node.Left.Left);
        break;
      case Operator.Not:
        node = node.Left.Left;
        break;
      case Operator.ForAll:
        node.Operator = Operator.Exists;
        node.Term = node.Left.Term;
        node.Left = Node.CreateNot(node.Left.Left);
        break;
      case Operator.Exists:
        node.Operator = Operator.ForAll;
        node.Term = node.Left.Term;
        node.Left = Node.CreateNot(node.Left.Left);
        break;
      default:
        throw new Exception("Unexpected operator");
    }
  }

  if (node.Left != null)
    node.Left = PrenexStep4(node.Left);
  if (node.Right != null)
    node.Right = PrenexStep4(node.Right);

  return node;
}
```

Výpis č. 5 – Přenesení negace bezprostředně před elementární formule

5. Přenesení všech kvantifikátorů na začátek formule. Toto lze dosáhnout opakovaným užitím následujících ekvivalencí:

$\forall x A \wedge B \Leftrightarrow \forall x(A \wedge B)$	$\exists x A \vee B \Leftrightarrow \exists x(A \vee B)$	B *neobsahuje volnou* x
$A \wedge \forall x B \Leftrightarrow \forall x(A \wedge B)$	$A \vee \exists x B \Leftrightarrow \exists x\ (A \vee B)$	A *neobsahuje volnou* x
$\exists x A \wedge B \Leftrightarrow \exists x(A \wedge B)$	$\forall x A \vee B \Leftrightarrow \forall x(A \vee B)$	B *neobsahuje volnou* x
$A \wedge \exists x B \Leftrightarrow \exists x(A \wedge B)$	$A \vee \forall x B \Leftrightarrow \forall x(A \vee B)$	A *neobsahuje volnou* x

Díky těmto ekvivalencím považujeme implementaci za prohledávání stromu od kořene k listům a při nalezení kvantifikátoru jej převedeme na počátek stromu. Nesmíme však zapomenout dodržet pořadí kvantifikátorů.

Implementace bude vypadat takto:

```
private List<Node> quantifiers = new List<Node>();

public Node PrenexStep5(Node node)
{
   node = FindAndCutQuantifiers(node);
   for (int i = quantifiers.Count - 1; i >= 0; i--)
   {
      var item = quantifiers[i];
      item.Left = node;
      node = item;
   }

   return node;
}

private Node FindAndCutQuantifiers(Node node)
{
   if (node.IsLeaf())
      return node;

   if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
   {
      quantifiers.Add(Node.CreateQuantifier(node.Operator, null, node.Term));
   }

   node.Left = FindAndCutQuantifiers(node.Left);
   if (node.Right != null)
      node.Right = FindAndCutQuantifiers(node.Right);

   if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
      return node.Left;

   return node;
}
```

Výpis č. 6 - Přenesení všech kvantifikátorů na začátek formule

Nejprve pomocí metody *FindAndCutQuantifiers* najdeme všechny kvantifikátory a přidáme si je do listu *quantifiers*. Formuli procházíme od kořene, proto testujeme uzly před zanořením do rekurze. Jak nám název metody napovídá, musíme uzly s kvantifikátory při vynořování z rekurze ze stromu odebrat, protože se jedná o přesun. Po vykonání metody *FindAndCutQuantifiers* máme k dispozici strom bez kvantifikátorů a list se seřazenými kvantifikátory. Výsledný strom sestavíme v cyklu, který procházíme od posledního kvantifikátoru z listu. Tomuto kvantifikátoru nastavíme jako potomka vrácený strom metodou a v dalších průchodech cyklem postupně dosazujeme zbylé kvantifikátory nad kořen stromu.

6.4 Návrh a implementace algoritmu pro vybrané podtřídy

V této kapitole projdeme algoritmy pro jednotlivé podtřídy. Obdobně jako v předchozí podkapitole si nejprve zopakujeme podmínky podtřídy, pak popíšeme algoritmus a ukážeme jeho implementaci.

6.4.1 Algoritmus pro Löb-Gurevichovu třídu

Algoritmus této třídy má tedy za úkol projít všechny listy stromu a otestovat jejich atomické formule, jestli obsahují pouze monadické predikáty.

```
// Löb-Gurevichova třída
public bool SubClass1(Node tree)
{
  FillFormulas(tree);

  foreach (var formula in formulas)
  {
    if (formula.Terms.Count > 1 ||
      formula.Terms[0].Type == TermType.Function)
    {
      return false;
    }
  }

  return true;
}
```

Výpis č. 7 - Löb-Gurevichova třída

Prvním krokem algoritmu je metoda *FillFormulas*. Tato metoda má vstupní parametr syntaktický strom v podobě objektu *Node*. Tento strom projde a naplní list *formulas* všemi atomickými formulemi z listu stromu. Implementaci této metody je zobrazena výpisem č.7. Tuto metodu, respektive tento list, bude využívat více tříd, proto jsme ji implementovali samotnou pro opětovné využití.

Po naplnění listu všemi atomickými formulemi procházíme tyto atomické formule v cyklu. Testujeme, jestli mají více než jeden term nebo Term je funkční term. Pokud je alespoň jedna z podmínek splněna, vrací metoda *false*, tedy zadaná formule nesplňuje podmínky Löb-Gurevichovy třídy. V jakémkoliv jiném případě vrací metoda *true*, tedy zadaná formule splňuje podmínky Löb-Gurevichovy třídy.

```
public List<AtomicFormula> formulas;

private void FillFormulas(Node node)
{
  if (formulas == null)  formulas = new List<AtomicFormula>();

  if (node.IsLeaf())
  {
```

```
        formulas.Add(node.Formula);
    }

    if (node.Left != null)
    {
        FillFormulas(node.Left);
    }
    if (node.Right != null)
    {
        FillFormulas(node.Right);
    }
}
```

Výpis č. 8 – Naplnění listu atomickými formulemi

Pokud při zavolání metody nebyl ještě list *formulas* inicializován, provede metoda jeho inicializaci.

6.4.2 Algoritmus pro Gödel-Kalmár-Schütteovu třídu

V tomto případě budeme testovat syntaktický strom v prenexní normální formě. Prefix stromu musí být ve tvaru $\exists^*\forall^2\exists^*$. Dále budeme testovat strom na přítomnost funkčního termu obdobně, jako jsme testovali v předešlé podtřídě.

Implementace algoritmu této třídy tedy obsahuje opět metodu *FillFormulas* pro naplnění kolekce *formulas*. Nově bude implementace obsahovat třídu *FillQuantifiers*, která prochází strom rekurzivně od kořene a postupně plní kolekci *quantifiers* nalezenými kvantifikátory.

```
private void FillQuantifiers(Node node)
{
    if (quantifiers == null) quantifiers = new List<Node>();


    if (node.IsLeaf())
    {
        return;
    }

    if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
    {
        quantifiers.Add(Node.CreateQuantifier(node.Operator, null, node.Term));
    }

    FillQuantifiers(node.Left);
    if (node.Right != null)
    {
        FillQuantifiers(node.Right);
    }
}
```

Výpis č. 9 – Naplnění listu kvantifikátory

Po naplnění kolekce *formulas* a *quantifiers* si nejprve otestujeme přítomnost funkčních symbolů, k tomuto testu opět využijeme list *formulas*, který jsme si opět pomocí metody *FillFormulas* naplnili. Poté testujeme pořadí kvantifikátorů. První podmínkou je, že kolekce musí obsahovat minimálně dva kvantifikátory, což je dáno podmínkou tvaru prefixu této třídy $\exists^*\forall^2\exists^*$. Druhou podmínkou již je cyklus, kde procházíme kvantifikátory a testujeme jejich pořadí. V případě, že narazíme na všeobecný kvantifikátor, testujeme tyto podmínky:

1) Kvantifikátor nesmí být posledním v kolekci.
2) Následující kvantifikátor musí být taky všeobecný.
3) Pokud už byly jednou podmínky 1) a 2) splněny, nesmí kolekce obsahovat další všeobecný kvantifikátor.

Implementace algoritmu vypadá takto:

```
// Gödel-Kalmár-Schütteova třída
public bool SubClass2(Node tree)
{
   FillFormulas(tree);
   FillQuantifiers(tree);

   foreach (var formula in formulas)
   {
      foreach (var term in formula.Terms)
      {
         if (term.Type == TermType.Function)
            return false;
      }
   }

   if (quantifiers.Count < 2)
   {
      return false;
    }

   bool forAllChecked = false;
   for (int i = 0; i < quantifiers.Count; i++)
   {
      if (quantifiers[i].Operator == Operator.ForAll)
      {
         if (forAllChecked  i < quantifiers.Count - 1 ||
            quantifiers[i + 1].Operator != Operator.ForAll)
         {
            return false;
         }
         forAllChecked = true;
         i++;
      }
   }

   return forAllChecked ;
}
```

Výpis č. 10 - Gödel-Kalmár-Schütteova třída

V implementaci podmínky pro všeobecné kvantifikátory jsme využili proměnné *forAllChecked*. Tato proměnná nám zajistí, že pokud by se po otestovaném prvním všeobecném kvantifikátoru narazilo na další, pak nebude splněna podmínka podtřídy. Dále nám tato metoda zajistí, že metoda nevrátí *true* v případě, kdy by formule obsahovala pouze existenční kvantifikátory.

6.4.3 Algoritmus pro Bernays-Schönfinkelovu třídu

Tato třída se od předchozí Gödel-Kalmár-Schütteovy třídy liší pouze ve tvaru prefixu $\exists^*\forall^*$.

```
// Bernays-Schönfinkelova třída
public bool SubClass3(Node tree)
{
   FillFormulas(tree);
   FillQuantifiers(tree);

   foreach (var formula in formulas)
   {
      foreach (var term in formula.Terms)
      {
         if (term.Type == TermType.Function)
            return false;
      }
   }

   bool existsChecked = false;
   for (int i = 0; i < quantifiers.Count; i++)
   {
      if (quantifiers[i].Operator == Operator.Exists)
      {
         if (existsChecked)
         {
            return false;
         }
      }
      else if (quantifiers[i].Operator == Operator.ForAll)
      {
         existsChecked = true;

      }
   }

   return true;
}
```

Výpis č. 11 - Bernays-Schönfinkelova třída

Algoritmem popsaným výpisem výše ověřujeme v prvním kroku, že strom neobsahuje žádné funkční symboly, jako tomu bylo v předchozích třídách. Následně v cyklu ověříme pořadí kvantifikátorů a to tak, že pokud narazíme na existenční kvantifikátor a proměnná *existsChecked* je nastavena na *true*, metoda vrací *false*, tedy formule nevyhovuje Bernays-Schönfinkelově třídě.

6.4.4 Algoritmus pro Gurevich-Maslov-Orevkovovu třídu

Gurevich-Maslov-Orevkovova třída, je třída všech těch formulí predikátové logiky prvního řádu bez rovnosti, jejichž prefix má v prenexní normální formě tvar $\exists^*\forall\exists^*$. Díky těmto podmínkám je evidentní, že implementace algoritmu této třídy se bude skládat z testu stromu na symbol rovnosti a přítomnost právě jednoho všeobecného kvantifikátoru.

```
 // Gurevich-Maslov-Orevkovova třída
public bool SubClass4(Node tree)
{
  if (FindEuqality(tree))
    return false;

  FillQuantifiers(tree);

  if (!quantifiers.Any())
    return false;

  bool forAllChecked = false;
  for (int i = 0; i < quantifiers.Count; i++)
  {
    if (quantifiers[i].Operator == Operator.ForAll)
    {
      if (forAllChecked)
      {
        return false;
      }
      forAllChecked = true;
    }
  }

  return forAllChecked ;
}
```

Výpis č. 12 - Gurevich-Maslov-Orevkovova třída

V implementaci výše se setkáváme s metodou *FindEuqality*, tato metoda prochází strom a vrátí *true*, pokud nalezne symbol =. Metoda tedy ověřuje, zda se jedná o predikátovou formuli s rovností. Dále testujeme, jestli kolekce *quantifiers* obsahuje alespoň jeden záznam. V následném cyklu ověřujeme, že kolekce obsahuje právě jeden všeobecný kvantifikátor. Metoda v tomto případě vrací proměnnou *forAllChecked*, která je *true*, pokud kolekce obsahovala alespoň jeden všeobecný kvantifikátor.

6.4.5 Algoritmus pro Gurevichovu třídu

Gurevichova třída je třída všech těch formulí predikátové logiky s rovností, které v prenexní normální formě mají prefix ve tvaru $\exists^*$. Formule dále musí obsahovat buďto dva funkční symboly nebo funkci s aritou větší než jedna.

Implementace algoritmu vypadá takto:

```
// Gurevichova třída
public bool SubClass5(Node tree)
{
   if (!FindEuqality(tree))
      return false;

   FillQuantifiers(tree);

   foreach (var item in quantifiers)
   {
      if(item.Operator == Operator.ForAll)
      {
         return false;
      }
   }


   FillFormulas(tree);

   int functionCount = 0;
   foreach (var formula in formulas)
   {
      foreach (var term in formula.Terms)
      {
         if(term.Type == TermType.Function)
         {
            functionCount ++;

            if(term.Terms.Count > 1)
            {
               return true;
            }
         }
      }
   }

   return (functionCount == 2);
}
```

Výpis č. 13 - Gurevichova třída

Nejprve využijeme metodu *FindEuqality*, pro ověření, zda se jedná o predikátovou formuli s rovností. Poté v cyklu ověříme, zda strom neobsahuje všeobecný kvantifikátor. Dalším cyklem procházíme všechny termy atomických formulí a testujeme, zda existují alespoň dva funkční symboly nebo funkční symbol s aritou větší než jedna. V tomto cyklu využíváme proměnnou *functionCount*, která nám eviduje počet funkčních symbolů. Na konci metoda vrací *true*, pouze pokud počet funkčních symbolů jsou právě dva.

6.4.6 Algoritmus pro Rabinovu třídu

Rabinova třída patří mezi třídy formulí predikátové logiky s rovností, která obsahuje jednu unární funkci a pouze monadické predikáty. Zároveň u této třídy neplatí žádné omezení, co se týče prefixu formule. Implementace algoritmu se bude podobat algoritmu pro Löb-Gurevichovu třídu.

```
// Rabinova třída
public bool SubClass6(Node tree)
{
   if (!FindEuqality(tree))
      return false;

   FillFormulas(tree);

   int functionCount = 0;
   foreach (var formula in formulas)
   {
      if (formula.Terms.Count > 1)
      {
         return false;
      }

      foreach (var term in formula.Terms)
      {
         if (term.Type == TermType.Function &&
            term.Terms.Count == 1)
         {
            functionCount++;
         }
         else
         {
            return false;
         }
      }
   }

   return (functionCount == 1);
}
```

Výpis č. 14 - Rabinova třída

V prvním kroku ověříme, že se jedná o predikátovou formuli s rovnosti, jako v předešlém případě. Dále v cyklu procházíme všechny atomické formule a jejich termy. Obdobně jako v předchozím algoritmu využíváme proměnnou *functionCount*, kterou inkrementujeme při nalezení unární funkce. Na konci metoda vrací *true*, pouze pokud formule obsahuje právě jednu unární funkci.

6.4.7 Algoritmus pro Shelahovu třídu

Shelahova třída je standardní fragment predikátové logiky prvního řádu s rovností. Prefix formule v prenexní formě má tvar $\exists^*\forall\exists^*$. Formule neobsahuje žádné funkční symboly s aritou $n \geq 2$. Formule

dále obsahuje nejvýše jeden unární funkční symbol. Dle podmínky pro prefix, můžeme využít část implementace algoritmu pro Gurevich-Maslov-Orevkovovu třídu, která má stejný prefix. Implementace podmínky funkčních symbolů je obdobná jako v předchozí Rabinova třídě. Na konci bude metoda vracet *true*, pokud budou obě podmínky pravdivé. Nesmíme ovšem zapomenout na test pro predikátovou logiku s rovností.

```
// Shelahova třída
public bool SubClass7(Node tree)
{
   if (!FindEuqality(tree))
      return false;

   FillQuantifiers(tree);

   bool forAllChecked = false;
   for (int i = 0; i < quantifiers.Count; i++)
   {
      if (quantifiers[i].Operator == Operator.ForAll)
      {
         if (forAllChecked)
         {
            return false;
         }
         forAllChecked = true;
      }
   }

   FillFormulas(tree);

   int functionCount = 0;
   foreach (var formula in formulas)
   {
      if (formula.Terms.Count > 1)
      {
         return false;
      }

      foreach (var term in formula.Terms)
      {
         if (term.Type == TermType.Function &&
            term.Terms.Count == 1)
         {
            functionCount++;
         }
         else
         {
            return false;
         }
      }
   }

   return forAllChecked && (functionCount < 2);
}
```

Výpis č. 15 - Shelahova třída

7. Implementace důkazového kalkulu

V této kapitole si popíšeme implementaci vybraného důkazového kalkulu, kterým je rezoluční metoda. V jednotlivých podkapitolách si nejprve specifikujeme požadavky pro implementaci a následně si implementaci ukážeme a popíšeme tak, jako jsme to udělali v předchozí kapitole.

7.1 Skolemova forma

Postupně si projdeme implementaci pro všechny body algoritmu pro převod do skolemovy klauzulární formy. Již s předstihem víme, že některé kroky jsme již implementovali a popisovali při převodu do prenexního tvaru, což nám v této podkapitole značně ulehčí práci. Pro tyt

Krok 1. Utvoření existenčního uzávěru formule:

V tomto kroku procházíme strom od kořene a ověřujeme, jestli některá z atomických formulí obsahuje term, který není vázán žádným kvantifikátorem.

```
public Node SkolemStep1(Node node)
{
   if (quantifiredTerms == null) quantifiredTerms = new List<Term>();

   if (node.IsLeaf())
   {
      foreach (var item in node.Formula.Terms)
      {
         if (item.Type == TermType.Variable &&
            !quantifiredTerms.Contains(item))
         {
            return Node.CreateQuantifier(Operator.Exists, node, item);
         }
      }

      return node;
   }

   if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
   {
      if (!quantifiredTerms.Contains(node.Term))
      {
         quantifiredTerms.Add(node.Term);
      }
   }

   if (node.Left != null)
   {
      node.Left = SkolemStep1(node.Left);
   }
   if (node.Right != null)
   {
```

```
        node.Right = SkolemStep1(node.Right);
    }

    if (node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
    {
        if (quantifiredTerms.Contains(node.Term))
        {
            quantifiredTerms.Remove(node.Term);
        }
    }

    return node;
}
```

Výpis č. 16 - Utvoření existenčního uzávěru formule

Algoritmus tohoto kroku je možné vidět ve výpisu výše. Nejprve procházíme strom od jeho kořene a kdykoliv narazíme na kvantifikátor, přidáme ho do kolekce *quantifiredTerms*, která nám slouží jako seznam právě vázaných proměnných v aktuální podformuli. Pokud narazíme na list, projdeme všechny termy atomické formule a zkontrolujeme, jestli jsou obsaženy v kolekci. Ty termy, které nebudou obsaženy v kolekci, jsou volnými a proto jako jejich rodiče vytvoříme uzel s existenčním kvantifikátorem. Při výstupu z rekurze, musíme přidaný kvantifikátor z kolekce zase odebrat.

Následující kroky algoritmu pro převod do Skolemovy formy jsme již jednou implementovali v převodu do prenexního tvaru, proto znovu využijeme těchto implementovaných metod takto:

- Krok 2. – metoda vypuštění nadbytečných kvantifikátorů (výpis č. 4)
- Krok 3. – metoda
- Krok 4. – metoda eliminace funktorů ⊃ a ≡ (výpis č. 1)
- Krok 5. – metoda přenesení negace bezprostředně před elementární formule (výpis č. 5)

Krok 6. Přesun kvantifikátorů doprava

V tomto kroku má algoritmus za úkol procházet strom od kořene a v případě, že narazí na uzel s kvantifikátorem, bude uzel posouvat hlouběji stromem, dokud nebudou oba potomci obsahovat proměnnou vázanou tímto kvantifikátorem nebo dojde-li k listu stromu.

```
public Node SkolemStep6(Node node)
{
    if(node.IsLeaf())
    {
        return node;
    }

    if(node.Operator == Operator.ForAll || node.Operator == Operator.Exists)
    {
        bool isInLeft = SearchForTerm(node.Left.Left, node.Term);
        bool isInRight = SearchForTerm(node.Left.Right, node.Term);

        if(isInLeft && !isInRight)
        {
            node.Left.Left = Node.CreateQuantifier(node.Operator, node.Left.Left, node.Term);
            node = node.Left.Left;
```

```
        }
        if(!isInLeft && isInRight)
        {
            node.Left.Right = Node.CreateQuantifier(node.Operator, node.Left.Right, node.Term);
            node = node.Left.Right;
        }
    }

    if (node.Left != null)
    {
        node.Left = SkolemStep1(node.Left);
    }
    if (node.Right != null)
    {
        node.Right = SkolemStep1(node.Right);
    }

    return node;
}
```

Výpis č. 17 - Přesun kvantifikátorů doprava

V implementaci algoritmu je možné vidět, že rekurzivně procházíme stromem, dokud nenarazíme na kvantifikátor. Pokud narazíme na kvantifikátor, využijeme pomocné metody *SearchForTerm*, která rekurzivně prochází aktuální podstrom a testuje, zda se se v něm vyskytuje zadaný term. Tuto metodu musíme zavolat jako pro levého potomka, tak pro pravého potomka kvantifikátoru. Pokud v levém podstromu se nevyskytuje a v pravém ano, přesuneme aktuální uzel s kvantifikátorem nad pravého potomka. Obdobně to provádíme, pokud levý obsahuje a pravý ne. Pokud oba potomci obsahují kvantifikátor, kvantifikátor dosáhla nejhloupějšího místa a metoda pokračuje pro další kvantifikátory.

Krok 7. Eliminace existenčních kvantifikátorů (Skolemizace)

Algoritmus v tomto kroku prochází strom od kořene. V tomto algoritmu opět využijeme kolekce, do které si ukládáme proměnné vázané všeobecnými kvantifikátory. V přídě, že narazíme na existenční kvantifikátor, musíme dle pravidel provést Skolemizaci. Tedy budeme vytvářet Skolemovy konstanty v závislosti na kolekci všeobecně vázaných termů. Opět si nejprve ukážeme implementaci a následně popíšeme.

```
private List<Term> usedConstants;
private List<Term> usedFunctions;
private Dictionary<string, Term> exchangeTerms;
public Node SkolemStep7(Node node)
{
    if (usedConstants == null) usedConstants = new List<Term>();
    if (usedFunctions == null) usedFunctions = new List<Term>();
    if (quantifiredTerms == null) quantifiredTerms = new List<Term>();
    if (exchangeTerms == null) exchangeTerms = new Dictionary<string, Term>();

    if (node.IsLeaf())
    {
        for (int i = 0; i < node.Formula.Terms.Count; i++)
        {
            var term = node.Formula.Terms[i];
```

```
        switch (term.Type)
        {
            case TermType.Constant:
                if (!usedConstants.Contains(term)) usedConstants.Add(term);
                break;
            case TermType.Function:
                if (!usedFunctions.Contains(term)) usedFunctions.Add(term);
                break;
            default:
                break;
        }

        for (int j = 0; j < term.Terms.Count; j++)
        {
            var funcTerm = term.Terms[j];
            switch (funcTerm.Type)
            {
                case TermType.Constant:
                    if (!usedConstants.Contains(term)) usedConstants.Add(term);
                    break;
                case TermType.Function:
                    if (!usedFunctions.Contains(term)) usedFunctions.Add(term);
                    break;
                default:
                    break;
            }

            if (exchangeTerms.ContainsKey(funcTerm.ToString()))
                funcTerm = exchangeTerms[term.ToString()];
        }

        if (exchangeTerms.ContainsKey(term.ToString()))
            term = exchangeTerms[term.ToString()];
    }

    return node;
}

if (node.Operator == Operator.ForAll)
{
    if (!quantifiredTerms.Contains(node.Term))
        quantifiredTerms.Add(node.Term);
}

if (node.Operator == Operator.Exists)
{
    if(quantifiredTerms.Count == 0)
    {
        var constantTerm = new Term();
        constantTerm.Symbol = 'c';
        while(usedConstants.Contains(constantTerm))
        {
            constantTerm.Index++;
        }

        exchangeTerms.Add(node.Term.ToString(), constantTerm);
    }
    else
```

```
        {
          var functionTerm = new Term();
          functionTerm.Symbol = 'f';
          while(usedFunctions.Contains(functionTerm))
          {
            functionTerm.Index++;
          }
          foreach (var item in quantifiredTerms)
          {
            functionTerm.Terms.Add(item);
          }
          exchangeTerms.Add(node.Term.ToString(), functionTerm);
        }
      }

      if (node.Left != null) node.Left = SkolemStep7(node.Left);
      if (node.Right != null) node.Right = SkolemStep7(node.Right);

      if (node.Operator == Operator.ForAll)
      {
        if (quantifiredTerms.Contains(node.Term))
          quantifiredTerms.Remove(node.Term);
      }
      if(node.Operator == Operator.Exists)
      {
        if(exchangeTerms.ContainsKey(node.Term.ToString()))
          exchangeTerms.Remove(node.Term.ToString());
      }
      return node;
    }
```

Výpis č. 18 - Eliminace existenčních kvantifikátorů (Skolemizace)

Jak je možné z výpisu vidět, jde zatím o naši nejkomplexnější metodu. Strom opět procházíme rekurzivně od kořene. V případě, že narazíme na list je potřeba si naplnit kolekci pro již použité konstanty (*usedConstants*) a kolekci pro použité funkční symboly (*usedFunctions*). Obě tyto kolekce využíváme v metodě pro vytvoření nové *Skolemovy konstanty*, respektive *Skolemovy funkční konstanty*. Pokud již symbol pro konstantu byl použit, inkrementujeme cyklem její index, dokud nezískáme dosud nepoužitou konstantu.

Narazíme-li v metodě na všeobecný kvantifikátor, uložíme si jeho term do kolekce udržující proměnné vázané všeobecným kvantifikátorem (*quantifiredTerms*). V případě, že narazíme na existenční kvantifikátor, vytvoříme novou *Skolemovu konstantu* dle pravidel Skolemizace. Jestliže kolekce *quantifiredTerms* neobsahuje žádné proměnné vytváříme *Skolemovu konstantu*, pokud obsahuje proměnné, pak vytváříme *Skolemovu funkční konstantu* se všemi termy z kolekce *quantifiredTerms*.

Tyto vytvořené konstanty uložíme do slovníku *exchangeTerms*, kde klíč bude textový řetězec nového termu a hodnota bude samotný term. Tento slovník dále využíváme k nahrazení termů v podformuli.

V poslední řadě nesmíme zapomenout při opuštění rekurze, tedy podformule odebrat z kolekce *quantifiredTerms* proměnnou v případě, že opouštíme uzel se všeobecným kvantifikátorem a ze slovníku *exchangeTerms* term.

Krok 8. Přesun všeobecných kvantifikátorů doleva.

Pro tento krok využijme metodu Přenesení všech kvantifikátorů na začátek formule (výpis č. 6). Tato metoda nám přesouvá také existenční kvantifikátory, což nám ovšem nevadí, protože jsme je v kroku 7 eliminovali.

Krok 9. Použití distributivních zákonů

Algoritmus pro tento krok je mnohem jednodušší než předchozí krok. Procházíme strom rekurzivně od kořene, pokud narazíme na uzel s konjunkcí, nahrazujeme tento uzel dle distributivního zákonu.

```
public Node SkolemStep9(Node node)
{
  if (node.IsLeaf())
    return node;

  if (node.Operator == Operator.Or)
  {
    if(node.Left.Operator == Operator.And && node.Right.IsLeaf())
    {
      var newLeft = Node.CreateNode(Operator.Or, node.Left.Left, node.Right);
      var newRight = Node.CreateNode(Operator.Or, node.Left.Right, node.Right);

      node.Operator = Operator.And;
      node.Left = newLeft;
      node.Right = newRight;
    }
    if(node.Right.Operator == Operator.And && node.Left.IsLeaf())
    {
      var newLeft = Node.CreateNode(Operator.Or, node.Right.Left, node.Left);
      var newRight = Node.CreateNode(Operator.Or, node.Right.Right, node.Left);

      node.Operator = Operator.And;
      node.Left = newLeft;
      node.Right = newRight;
    }
  }

  if (node.Left != null) node.Left = SkolemStep7(node.Left);
  if (node.Right != null) node.Right = SkolemStep7(node.Right);

  return node;
}
```

Výpis č. 19 - Použití distributivních zákonů

7.2 Rezoluční metoda

Abychom mohli uplatňovat pravidlo rezoluce, je potřeba si náš syntaktický strom ve Skolemově formě převést na přívětivější datovou strukturu. Pro rezoluční metodu využíváme objekt typu *Clause*, který nám reprezentuje klauzuli literálů. Tento objekt má vlastnosti:

- **Id** – jedná se o identifikátor klauzule
- **LeftId** a **RightId** – jedná se o identifikátory předchozích klauzulí
- **Formulas** – jedná se o kolekci literálů (atomických formulí)
- **IsUsed** – identifikátor, jestli již byla klauzule využita pro rezoluci

Nejprve si ukážeme implementaci algoritmu pro rezoluční metodu a popíšeme si ji:

```
public List<Clause> DoResolution(List<Clause> clauses)
{
  for (int i = 0; i < clauses.Count; i++)
  {
    var clause1 = clauses[i];
    if (clause1.IsUsed)
      continue;

    for (int j = 0; j < clauses.Count; j++)
    {
      var clause2 = clauses[j];
      if (clause1.Id == clause2.Id || clause2.IsUsed)
        continue;

      var result = ResolveClauses(clause1.Formulas, clause2.Formulas);

      if (result != null)
      {
        clauses.Add(new Clause
        {
          Id = clauses.Count + 1,
          LeftId = clause1.Id,
          RightId = clause2.Id,
          IsUsed = false,
          Formulas = result
        }
        );

        clause1.IsUsed = true;
        clause2.IsUsed = true;

        i = 0;
        j = 0;
        break;
      }
    }
  }
  return clauses;
}
```

Výpis č. 20 – Rezoluční metoda

Metoda *DoResolution* přijímá jako vstupní parametr kolekci objektů *Clause*. Tato kolekce byla vytvořena ze syntaktického stromu ve Skolemově formě. V této metodě procházíme cyklicky klauzule a snažíme se pomocí volání metody *ResolveClauses* (výpis č. 21) uplatnit rezoluci. Metoda *ResolveClauses* nám vrací kolekci literálů po uplatnění rezoluce, pokud nevrátí nic, nebylo možné rezoluci nad literály uplatnit a pokračuje dalšími klauzulemi. Vrátí-li novou kolekci literálů, vytvoříme z nich novou klauzuli a přidáme do původní kolekce. Po tomto přidání je důležité nastavit proměnné *IsUsed* hodnotou *true*, aby nemohly být použity pro rezoluci podruhé. Dále nastavím proměnné pro iteraci na hodnotu nula, abychom v cyklu procházeli nově upravenou kolekci od začátku. Metoda končí, prošli jsme již všechny klauzule.

```
private List<AtomicFormula> ResolveClauses(List<AtomicFormula> clause1, List<AtomicFormula> claus
e2)
{
  var formulas1 = new List<AtomicFormula>();
  var formulas2 = new List<AtomicFormula>();
  formulas1.AddRange(clause1);
  formulas2.AddRange(clause2);

  foreach (var item1 in clause1)
  {
    foreach (var item2 in clause2)
    {
      if (item1.Symbol == item2.Symbol && item1.Index == item2.Index &&
        item1.Terms == item2.Terms && item1.IsNegated != item2.IsNegated)
      {
        formulas1.Remove(item1);
        formulas2.Remove(item2);
      }
    }
  }

  if (formulas1.Count != clause1.Count)
  {
    formulas1.AddRange(formulas2);

    return formulas1;
  }

  return null;
}
```

Výpis č. 21 – Rezoluce klauzulí

Tato implementace se týká rezoluční metody, která ovšem neumí řešit substituci. Proto si v následující podkapitole rozšíříme toto řešení o implementaci Robinsonova unifikačního algoritmu.

7.3 Robinsonův unifikační algoritmus

V této podkapitole rozšíříme rezoluční metodu o Robinsonův unifikační algoritmus. Implementace rozšiřuje metodu *ResolveClauses* (výpis č. 21). Robinsonův algoritmus aplikujeme při každém pokusu o rezoluci. Implementace vypadá takto:

```
private List<AtomicFormula> ResolveClauses(List<AtomicFormula> clause1, List<AtomicFormula> claus
e2)
{
   var formulas1 = new List<AtomicFormula>();
   var formulas2 = new List<AtomicFormula>();
   formulas1.AddRange(clause1);
   formulas2.AddRange(clause2);

   for (int i = 0; i < clause1.Count; i++)
   {
      var item1 = ApplySubstitute(clause1[i]);
      for (int j = 0; j < clause2.Count; j++)
      {
         var item2 = ApplySubstitute(clause2[j]);
         if (FindSubstitute(item1, item2))
         {
            formulas1.Remove(item1);
            formulas2.Remove(item2);
         }
      }
   }

   if (formulas1.Count != clause1.Count)
   {
      formulas1.AddRange(formulas2);

      return formulas1;
   }

   return null;
}
```

Výpis č. 22 – Rozšíření rezoluční metody o Robinsonův unifikační algoritmus

Jak je možné vidět, implementace rozšířila metodu o aplikaci metody *ApplySubstitute*, která je pouze pomocnou metodou a jediné co provádí je, jak je již z názvu patrné, aplikuje substituční pravidla ze slovníku *substitutes*, který obsahuje všechny aktuální substituce. Tento slovník plní metoda *FindSubstitute*, která nahradila implementaci pro rezoluční pravidlo s využitím unifikace.

Implementaci metody *FindSubstitute*:

```
private Dictionary<string, Term> substitutes = new Dictionary<string, Term>();
public bool FindSubstitute(AtomicFormula formula1, AtomicFormula formula2)
{
```

```
    var tempDictionary = new Dictionary<string, Term>();
    if (formula1.Symbol != formula2.Symbol ||
        formula1.Index != formula2.Index ||
        formula1.Terms.Count != formula2.Terms.Count ||
        (formula1.IsNegated && !formula2.IsNegated) ||
        (!formula1.IsNegated && formula2.IsNegated))
    {
        return false;
    }

    foreach (var term1 in formula1.Terms)
    {
        foreach (var term2 in formula2.Terms)
        {
            if (term1 == term2)
                continue;

            if (term1.Type == TermType.Variable)
            {
                tempDictionary.Add(term1.ToString(), term2);
            }
            else if (term2.Type == TermType.Variable)
            {
                tempDictionary.Add(term2.ToString(), term1);
            }
            else
            {
                return false;
            }
        }
    }

    foreach (var item in tempDictionary)
    {
        substitutes.Add(item.Key, item.Value);
    }

    return true;
}
```

Výpis 23 – Užití Robinsonova algoritmu

Metoda nám v prvním kroku ověří základní pravidla pro využití rezolučního pravidla, pokud jsou pravidla splněny, aplikuje Robinsonů algoritmus. Nejprve otestujeme, jestli jsou termy shodné, pokud ano pokračujeme dále. Pokud shodné nejsou, otestujeme, zda je alespoň jeden term proměnná, pokud ano, pak substituujeme druhý term za tuto proměnnou. Tuto naši novou substituci přidáme do pouze dočasného slovníku, ze kterého přesypeme pravidla do hlavního slovníku až poté, co máme jistotu, že unifikaci je možné provést.

8. Závěr

Cílem této práce byl návrh a implementace algoritmu pro rozhodování predikátové logiky 1. řádu v jejích rozhodnutelných podtřídách, které byly dokázány v předešlé diplomové práci, na kterou tato práce navazuje. Tato práce byla vedena jako implementační s důrazem na výsledný program, kterým je webová aplikace umístěna na adrese http://algoritmus-rozhodnutelnych-podtrid.cz/.

Zadání práce bylo splněno ve všech bodech a výsledný program také splňuje všechny požadavky. Práce začala plnit body zadání úvodem do predikátové logiky, kde jsme se nesoustředili na podrobné definice a dokazování, ale základní úvod, potřebný ke zhotovení algoritmu. Dále jsme podle zadaní shrnuli rozhodnutelné podtřídy, kde jsme hlavně uvedli jejich podmínky. Následoval výběr důkazového kalkulu a jeho zdůvodnění. Poté jsme se již dostali k praktičtější části práce, kde jsme se přes výběr datové struktury dostali k implementaci algoritmu pro jednotlivé podtřídy a zakončili implementací důkazového kalkulu.

Přínosem této práce je určitě praktické rozšíření dokázaných rozhodnutelných podtříd z předešlé diplomové práce. Jelikož je výsledný program implementován jako webová aplikace, může dále najít využití u studentů, kteří při studiu potřebují rozhodnout predikátové formule nebo je jen převést do Skolemovy formy.

Literatura

1. Radim Vašíček - *Rozhodnutelné podtřídy formulí predikátového počtu 1. řádu*, 2012

2. Marie Duží - *Logika pro informatiky* (a příbuzné obory), učební text, 2012

3. Egon Börger, Erich Grädel, Yuri Gurevich - *The Classical Decision Problem.* Springer, 2001

4. Monadic predicate calculus [Online] 23. června 2016.
   < https://en.wikipedia.org/wiki/Monadic_predicate_calculus>

5. Strom (datová struktura) [Online] 23. června 2016.
   < https://cs.wikipedia.org/wiki/Strom_(datov%C3%A1_struktura) >

6. Jiří Dvorský – *Algoritmy I.* (pracovní verze skript), 2007

Seznam výpisu zdrojového kódu

Výpis č. 1 – Eliminace spojek ⊃ a ≡ 22

Výpis č. 2 – Aplikace ekvivalencí druhého kroku prenexního algoritmu 23

Výpis č. 3 – Převod formule na tvar s čistými proměnnými 24

Výpis č. 4 – Vypuštění nadbytečných kvantifikátorů 25

Výpis č. 5 – Přenesení negace bezprostředně před elementární formule 26

Výpis č. 6 - Přenesení všech kvantifikátorů na začátek formule 27

Výpis č. 7 - Löb-Gurevichova třída 28

Výpis č. 8 – Naplnění listu atomickými formulemi 29

Výpis č. 9 – Naplnění listu kvantifikátory 29

Výpis č. 10 - Gödel-Kalmár-Schütteova třída 30

Výpis č. 11 - Bernays-Schönfinkelova třída 31

Výpis č. 12 - Gurevich-Maslov-Orevkovova třída 32

Výpis č. 13 - Gurevichova třída 33

Výpis č. 14 - Rabinova třída 34

Výpis č. 15 - Shelahova třída 35

Výpis č. 16 - Utvoření existenčního uzávěru formule 37

Výpis č. 17 - Přesun kvantifikátorů doprava 38

Výpis č. 18 - Eliminace existenčních kvantifikátorů (Skolemizace) 40

Výpis č. 19 - Použití distributivních zákonů 41

Výpis č. 20 – Rezoluční metoda 42

Výpis č. 21 – Rezoluce klauzulí 43

Výpis č. 22 – Rozšíření rezoluční metody o Robinsonův unifikační algoritmus 44

Výpis 23 – Užití Robinsonova algoritmu 45

Seznam obrázků

Obrázek č.1: Syntaktický strom formule 16

Obrázek č. 2: Algoritmus pro rozhodování ve vybraných podtřídách PL1 18

Obrázek č. 3: kořen stromu 20

Obrázek č. 4: konstrukce stromu po 3. kroku 20

Obrázek č.5: konstrukce stromu po 4. kroku 21

Obrázek č. 6: konstrukce stromu po 6. kroku 21