FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Resoluce pro predikátovou logiku
Student:	Michal Dvořák
Vedoucí:	Mgr. Jan Starý, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1. Popište rezoluční metodu pro predikátovou logiku.
2. Implementujte parser formulí v jazyce logiky prvního řádu.
3. Implementujte algoritmus, který převádějí formuli do prenexního tvaru.
4. Implementujte algoritmus, který formuli skolemizuje.
5. S pomocí výše uvedeného implementujte rezoluční metodu.
6. Implementaci proveďte v jazyce C nebo C++.
7. Dbejte na korektnost a přenositelnost kódu.
8. Vytvořený program zdokumentujte, nejlépe ve formě standardní manuálové stránky.
9. Vytvořený program otestujte, metody testování zdokumentujte.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 19. února 2021 v Praze.

Bakalářská práce

RESOLUCE PRO PREDIKÁTOVOU LOGIKU

Michal Dvořák

Fakulta informačních technologií ČVUT v Praze
Katedra teoretické informatiky
Vedoucí: Mgr. Jan Starý, Ph.D.
13. května 2021

Obsah

Poděkování iv

Prohlášení v

Abstrakt vi

Úvod 1

1 Resoluční metoda 3
1.1 Syntax predikátové logiky . . . 3
1.2 Sémantika predikátové logiky . . . 4
1.3 Normální tvary . . . 7
1.4 Resoluce ve výrokové logice. . . . 8
1.5 Prenexní tvary, skolemizace . . . 9
1.6 Resoluce v predikátové logice . . . 11

2 Algoritmy 19
2.1 Abstraktní syntaktický strom . . . 19
2.2 Volné a vázané proměnné . . . 20
2.3 Prenexní tvar . . . 21
2.4 Skolemizace . . . 23
2.5 Konjunktivní normální tvar . . . 23
2.6 Unifikace . . . 25
2.7 Resoluce . . . 28
2.8 Instance . . . 29

3 Implementace 31
3.1 Abstraktní syntaktický strom . . . 31
3.2 Lexer a parser . . . 34
3.3 Algoritmy . . . 35
3.4 Použití programu . . . 36

Závěr 37

A Obsah přiloženého média 41

Chtěl bych poděkovat především vedoucímu práce Mgr. Janovi Starému, Ph.D. za jeho ochotu a čas, který mi věnoval při pravidelných konzultacích. Dále bych chtěl poděkovat své rodině a všem, kteří mi během tvorby práce jakkoliv pomáhali, motivovali mě nebo mě psychicky podporovali.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 13. května 2021 .....

Abstrakt

Výroková logika je rozhodnutelná – existuje algoritmus, který pro zadanou výrokovou formuli rozhodne, zda je splnitelná. Predikátová logika rozhodnutelná není – žádný algoritmus, který by v plné obecnosti rozhodoval o splnitelnosti predikátových formulí, neexistuje. Existují však procedury, které umí otázku splnitelnosti predikátové formule částečně řešit. Jednou takovou je právě resoluční metoda. Rozšířením myšlenek této metody lze získat odvozovací systém známý jako resoluční kalkul. Popíšeme teoretické základy nutné pro pochopení problematiky a následně implementujeme resoluční metodu v jazyce C++.

Klíčová slova resoluce, resoluční metoda, predikátová logika, splnitelnost, C++

Abstract

Propositional logic is decidable – there exists an algorithm deciding whether any given propositional formula is satisfiable. Predicate logic is not decidable – in general, no algorithm can decide whether any given predicate formula is satisfiable. On the other hand, there are procedures which solve this problem partially. Resolution method is one of them. Extending the resolution method, one can obtain a deduction system called the resolution calculus. We describe the theoretical preliminaries important for the resolution method and we implement it in C++.

Keywords resolution, resolution method, predicate logic, satisfiability, C++

Úvod

Historie

Na začátku 20. let 20. století zveřejnil německý matematik David Hilbert tzv. *Hilbertův program*, což měl být seznam těch nejdůležitějších nevyřešených problémů tehdejší matematiky. Jedním z bodů Hilbertova programu byl problém *rozhodnutelnosti*. Mělo se ukázat, že existuje algoritmus, který o každé formuli v jazyce logiky prvního řádu rozhodne, zda je splnitelná. Později se tomuto problému začalo říkat *Entscheidungsproblem*. Paradoxem bylo, že se měla ukázat existence algoritmu, ale pojem *algoritmu* v té době ještě nebyl ani pořádně definovaný. V roce 1935 přišel Alonzo Church s definicí algoritmu založenou na konceptu λ-kalkulu a hned o rok později Alan Turing s definicí založenou na konceptu Turingových strojů. Turing si okamžitě uvědomil, že tyto dva koncepty – λ-kalkulus a Turingovy stroje – jsou ekvivalentními modely výpočtu a tedy, že tyto dvě definice algoritmu splývají. Za použití λ-kalkulu podal v roce 1936 Alonzo Church v článku [Chu36] negativní odpověď na Entscheidungsproblem – žádný takový algoritmus, který by rozhodoval o platnosti formulí v jazyce logiky prvního řádu, neexistuje. Tento výsledek je označován jako *Churchova věta*. O rok později dokázal nezávisle na Churchovi totéž i Alan Turing v článku [Tur37].

Splnitelnost predikátových formulí tedy nikdy nemůže žádný algoritmus rozhodovat, a tak musí matematiku stále dělat člověk. Můžeme ale dělat následující. Podle věty o úplnosti logiky prvního řádu lze otázku, zda je nějaká formule pravdivá, redukovat na finitární prostředky – důkazy. Pravdivost formule je ekvivalentní její dokazatelnosti v nějakém dokazovacím systému. Pokud tedy máme pravdivou formuli, existuje její důkaz v nějakém dokazovacím systému a takový důkaz můžeme najít. Na druhou stranu pokud formule pravdivá není, můžeme takto hledat „do nekonečna“. S rozvojem počítačů v 60. letech 20. století byla snaha vymyslet procedury[1], pomocí kterých by se tyto důkazy pro pravdivé formule daly efektivně hledat. Jednou takovou procedurou je právě *resoluční metoda*.

Cíle práce

Hlavním cílem práce je představit myšlenky resoluční metody z článku [Rob65] J. A. Robinsona z roku 1965, převést je na algoritmy a následně tyto algoritmy implementovat v jazyce C++. V první kapitole popíšeme základní pojmy z predikátové logiky a plynule navážeme resoluční metodou založenou na článku [Rob65]. V druhé kapitole se zaměříme na detailnější popis algoritmů, které při implementaci využijeme. Ve třetí kapitole shrneme hlavní myšlenky jejich implementace v jazyce C++.

[1] Tyto procedury jsou dnes základem tzv. *proof assistantů*.

Kapitola 1
Resoluční metoda

V této kapitole popíšeme základní pojmy z predikátové logiky a navážeme s resoluční metodou založenou na článku [Rob65]. Použité pojmy lze najít například v [Šve02].

1.1 Syntax predikátové logiky

Definice 1.1.1. *Jazyk predikátové logiky* $\mathcal{L}$ sestává z množiny *konstantních symbolů*, množiny *funkčních symbolů*, množiny *relačních symbolů* (nebo také *predikátů*), nekonečné dobře uspořádané množiny *proměnných*, závorek $(,)$, kvantifikátorů $\forall, \exists$ a logických spojek $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. Existuje funkce, která každému funkčnímu a relačnímu symbolu přiřazuje jeho aritu – přirozené číslo $n \geq 1$. Má-li funkční, resp. relační symbol aritu n, mluvíme o *n-árním* funkčním, resp. relačním symbolu.

Poznámka 1.1.2. Přijmeme konvenci, že jména konstantních symbolů začínají velkými písmeny nebo číslicemi a jména proměnných malými písmeny.

Definice 1.1.3. Nechť $\mathcal{L}$ je jazyk predikátové logiky. *Termy* a *formule* definujeme induktivně:

a) každý konstantní symbol a proměnná je term,

b) je-li f n-ární funkční symbol a $t_1, \ldots, t_n$ jsou termy, pak $f(t_1, \ldots, t_n)$ je term,

c) je-li r n-ární relační symbol a $t_1, \ldots, t_n$ jsou termy, pak $r(t_1, \ldots, t_n)$ je formule,

d) jsou-li φ, ψ formule, pak $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \Rightarrow \psi), (\varphi \Leftrightarrow \psi), (\neg\varphi)$ jsou formule,

e) je-li φ formule a x proměnná, pak $(\forall x)\varphi, (\exists x)\varphi$ jsou formule.

Formule je *atomická*, právě když je tvaru $r(t_1, \ldots, t_n)$ pro nějaký n-ární relační symbol r a termy $t_1, \ldots, t_n$. *Podformule* je jakákoliv část formule, která je sama formulí. *Konstantní term* je term bez proměnných.

Poznámka 1.1.4. Za předpokladu, že zůstane zápis formule jednoznačný, povolíme, aby nějaký symbol byl například zároveň funkčním symbolem, relačním symbolem a proměnnou. Stejně tak dovolíme, aby se stejné jméno vyskytovalo jako více různých funkčních nebo relačních symbolů. V takovém případě ale musí mít tyto funkční a relační symboly různé arity. Například ve formuli $(\forall f)(f(f(f(f, f(f)))) \wedge f(f(f), f(f, f)))$ je f relační symbol arity 1, relační symbol arity 2, funkční symbol arity 1, funkční symbol arity 2 a také je to proměnná.

Poznámka 1.1.5. Definice formule požaduje, abychom důsledně uzávorkovali všechny podformule. Tím by se už i zápis jednoduché formule jako $((p(x) \wedge q(y)) \vee r(z))$ dost zkomplikoval. Přijmeme konvenci, že všechny spojky jsou levě asociativní, tj. závorky se kumulují doleva, a priorita logických spojek klesá v pořadí $\{\forall, \exists, \neg\}, \{\wedge, \vee\}, \{\Rightarrow\}, \{\Leftrightarrow\}$. Jinými slovy, kvantifikátory a negace (tj. všechny unární spojky) mají stejnou prioritu, a ta je větší než priorita $\wedge$ a $\vee$ atd. V tomto smyslu považujeme kvantifikátor za unární spojku, jejíž jediný argument je formule, přes kterou kvantifikuje.

Příklad 1.1.6. Z každé formule lze vydedukovat použitý jazyk. Je-li

$$(\forall x)(\exists y)(p(f(x), y) \wedge p(w) \Rightarrow (\exists z)(p(z))) \Leftrightarrow p(g(x, y, C), 42))$$

formule jazyka $\mathcal{L}$, pak $\mathcal{L}$ obsahuje relační symbol p arity 1, relační symbol p arity 2, funkční symbol f arity 1, funkční symbol g arity 3, proměnné x, y, z, w a konstantní symboly $C, 42$.

Všimněme si, že w jsme skutečně rozpoznali jako proměnnou, nikoliv jako konstantní symbol, a naopak C a 42 jako konstantní symboly, nikoliv jako proměnné, protože jsme přijali konvenci, že konstantní symboly začínají velkým písmenem nebo číslicí a proměnné malými písmeny (viz poznámka 1.1.2).

Definice 1.1.7. Nechť φ je formule. Řekneme, že výskyt proměnné x ve formuli φ je *vázaný*, právě když je částí nějaké podformule tvaru $(\forall x)\psi$, resp. $(\exists x)\psi$. V opačném případě je *volný*. Výskyt proměnné x je *vázaný k výskytu kvantifikátoru* $\forall x$, resp. $\exists x$, právě když se x vyskytuje v podformuli $(\forall x)\psi$, resp. $(\exists x)\psi$ a je to minimální[1] taková podformule, ve které je tento výskyt vázaný. Formule bez volných výskytů proměnných je *uzavřená* nebo také *sentence*. Formule, ve které jsou všechny výskyty proměnných volné, je *otevřená*.

Příklad 1.1.8. Ve formuli $(\forall x)(p(x) \wedge (\exists x)q(x)) \vee r(x)$ je výskyt proměnné x v predikátu $p(x)$ vázaný k výskytu $\forall x$ a výskyt proměnné x v $q(x)$ vázaný k výskytu $\exists x$. Výskyt proměnné x v predikátu $r(x)$ je volný.

Příklad 1.1.9. Formule $(\forall x)p(x)$ je uzavřená. Formule $(\forall x)p(x) \vee q(z)$ není uzavřená ani otevřená. Formule $p(x)$ je otevřená. Formule $p(C)$, kde C je konstantní symbol, je otevřená i uzavřená.

Nadále budeme zkracovat a místo *výskyt proměnné je volný/vázaný (v podformuli)* budeme říkat jen *proměnná je volná/vázaná (v podformuli)*. Je důležité, že volnost/vázanost proměnné se vztahuje k jejímu *výskytu*, nikoliv k jejímu *jménu*. Dále také budeme používat obrat *formule má volné/vázané proměnné* $x_1, \ldots, x_n$. Tím myslíme, že se proměnné $x_1, \ldots, x_n$ ve formuli vyskytují volně/vázaně.

1.2 Sémantika predikátové logiky

Definice 1.2.1. Nechť $\mathcal{L}$ je jazyk predikátové logiky. *Model* jazyka $\mathcal{L}$ je neprázdná množina M zvaná *univerzum* společně s

a) konstantou $C^{\mathfrak{M}} \in M$ za každý konstantní symbol C,

b) n-ární funkcí $f^{\mathfrak{M}} : M^n \to M$ za každý n-ární funkční symbol f,

c) n-ární relací $r^{\mathfrak{M}} \subseteq M^n$ za každý n-ární relační symbol r.

Řekneme, že struktura $\mathfrak{M} = (M, r^{\mathfrak{M}}, \ldots, f^{\mathfrak{M}}, \ldots, C^{\mathfrak{M}}, \ldots)$ *realizuje* symboly jazyka $\mathcal{L}$ v množině M a píšeme $\mathfrak{M} \models \mathcal{L}$.

[1] Minimální ve smyslu syntaktické složitosti formule.

Definice 1.2.2. Nechť $\mathcal{L}$ je jazyk predikátové logiky a $\mathfrak{M} \models \mathcal{L}$ model s univerzem M. Libovolné zobrazení e z množiny proměnných jazyka $\mathcal{L}$ do M nazveme *ohodnocením proměnných*. Pro dané ohodnocení e a daný term t jazyka $\mathcal{L}$ definujeme hodnotu $t[e] \in M$ termu t při ohodnocení e induktivně:

a) je-li t konstantní symbol C, pak $t[e]$ je prvek $C^{\mathfrak{M}} \in M$,

b) je-li t proměnná x, pak $t[e]$ je prvek $e(x) \in M$,

c) je-li t tvaru $f(t_1, \ldots, t_n)$ pro nějaký n-ární funkční symbol f realizovaný funkcí $f^{\mathfrak{M}}$, pak $t[e]$ je prvek $f^{\mathfrak{M}}(t_1[e], \ldots, t_n[e]) \in M$.

Definice 1.2.3. Nechť $\mathcal{L}$ je jazyk predikátové logiky a $\mathfrak{M} \models \mathcal{L}$ a nechť e je ohodnocení proměnných. O formuli φ jazyka $\mathcal{L}$ řekneme, že je *splněna v* $\mathfrak{M}$ *při ohodnocení* e a píšeme $\mathfrak{M} \models \varphi[e]$, právě když nastává některý z následujících případů:

a) φ je tvaru $r(t_1, \ldots, t_n)$ pro nějaký n-ární predikát r a $(t_1[e], \ldots, t_n[e]) \in r^{\mathfrak{M}}$, kde $r^{\mathfrak{M}}$ je realizace predikátu r v $\mathfrak{M}$,

b) φ je tvaru $(\psi \wedge \vartheta)$ a $\mathfrak{M} \models \psi[e]$ *a zároveň* $\mathfrak{M} \models \vartheta[e]$,

c) φ je tvaru $(\psi \vee \vartheta)$ a $\mathfrak{M} \models \psi[e]$ *nebo* $\mathfrak{M} \models \vartheta[e]$,

d) φ je tvaru $(\psi \Rightarrow \vartheta)$ a $\mathfrak{M} \not\models \psi[e]$ *nebo* $\mathfrak{M} \models \vartheta[e]$,

e) φ je tvaru $(\psi \Leftrightarrow \vartheta)$ a $\mathfrak{M} \models (\psi \Rightarrow \vartheta)[e]$ *a zároveň* $\mathfrak{M} \models (\vartheta \Rightarrow \psi)[e]$,

f) φ je tvaru $(\neg\psi)$ a $\mathfrak{M} \not\models \psi[e]$,

g) φ je tvaru $(\forall x)\psi$ a pro každé $m \in M$ a každé ohodnocení e', které přiřazuje proměnné x hodnotu m a jinak se shoduje s e, je $\mathfrak{M} \models \psi[e']$,

h) φ je tvaru $(\exists x)\psi$ a pro nějaké $m \in M$ a každé ohodnocení e', které přiřazuje proměnné x hodnotu m a jinak se shoduje s e, je $\mathfrak{M} \models \psi[e']$.

Pokud $\mathfrak{M} \models \varphi[e]$ pro nějaký model $\mathfrak{M}$ a ohodnocení e, pak řekneme, že φ je *splnitelná*. Pokud $\mathfrak{M} \models \varphi[e]$ při každém ohodnocení e, pak řekneme, že φ *platí* v $\mathfrak{M}$ a píšeme jen $\mathfrak{M} \models \varphi$. Pokud φ platí v každém modelu, řekneme, že φ je *tautologie* nebo také *logicky platná formule* a píšeme $\models \varphi$. Pokud φ není splněna v žádném modelu při žádném ohodnocení[2], řekneme, že je *nesplnitelná* nebo také *kontradikce*. Dvě formule φ, ψ jsou *ekvivalentní*, právě když $\models \varphi \Leftrightarrow \psi$. O dvou formulích φ, ψ řekneme, že jsou *ekvisplnitelné* právě tehdy, když platí, že φ platí v nějakém modelu, právě když ψ platí v nějakém modelu.

Z definice plyne, že formule φ je nesplnitelná, právě když není splnitelná, φ je tautologie, právě když $\neg\varphi$ je kontradikce a φ je kontradikce, právě když $\neg\varphi$ je tautologie.

Lemma 1.2.4. *Nechť $\mathcal{L}$ je jazyk predikátové logiky a $\mathfrak{M} \models \mathcal{L}$ model. Nechť $x_1, \ldots, x_n$ jsou proměnné a nechť e_1, e_2 jsou dvě ohodnocení proměnných, která se shodují na proměnných $x_1, \ldots, x_n$. Přesněji pro každé $i \in \{1, \ldots, n\}$ je $e_1(x_i) = e_2(x_i)$. Je-li φ formule jazyka $\mathcal{L}$, jejíž všechny volné proměnné jsou mezi $x_1, \ldots, x_n$, pak $\mathfrak{M} \models \varphi[e_1]$, právě když $\mathfrak{M} \models \varphi[e_2]$.*

Důkaz. Důkaz se provede indukcí podle syntaktické složitosti formule a je spíše technický než zajímavý. Viz lemma 3.1.11. v [Šve02]. ■

Důsledek 1.2.5. *Nechť $\mathcal{L}$ je jazyk a $\mathfrak{M} \models \mathcal{L}$ model. Pokud je φ uzavřená a je splněna v $\mathfrak{M}$ při nějakém ohodnocení, pak je už splněna v $\mathfrak{M}$ při každém ohodnocení.*

[2] Pro každý model $\mathfrak{M}$ a každé ohodnocení e je $\mathfrak{M} \not\models \varphi[e]$.

Definice 1.2.6. Nechť $\mathcal{L}$ je jazyk predikátové logiky. *Teorie* je množina formulí jazyka $\mathcal{L}$. Formulím v T říkáme *axiomy* teorie T. Řekneme, že model $\mathfrak{M} \models \mathcal{L}$ je *modelem teorie* T, právě když pro každou formuli $\psi \in T$ je $\mathfrak{M} \models \psi$, píšeme $\mathfrak{M} \models T$. Pokud má teorie T model, řekneme, že je *splnitelná*, v opačném případě je *nesplnitelná*. Teorie je *uzavřená*, právě když všechny její axiomy jsou uzavřené formule.

Definice 1.2.7. Nechť φ je formule a nechť $x_1, \ldots, x_n$ jsou všechny její volné proměnné. *Uzávěr* formule φ je formule $(\forall x_1) \ldots (\forall x_n)\varphi$, značíme $\overline{\varphi}$.

Poznámka 1.2.8. Chceme-li zkoumat modely teorie T, můžeme bez újmy na obecnosti předpokládat, že T je uzavřená, protože definice modelu teorie požaduje, aby všechny $\psi \in T$ platily v daném modelu – tj. mají být splněny při každém ohodnocení. Pokud formule v T uzavřené nejsou, můžeme vyrobit jejich uzávěr. Není těžké si rozmyslet, že pro každý model $\mathfrak{M}$ a formuli ψ je $\mathfrak{M} \models \psi$, právě když $\mathfrak{M} \models \overline{\psi}$. Jinými slovy, pro každou teorii T mají teorie T a $\{\overline{\psi} \mid \psi \in T\}$ stejné modely.

Definice 1.2.9. Nechť T je teorie. Řekneme, že formule φ je *důsledkem* teorie T, právě když φ platí ve všech modelech $\mathfrak{M} \models T$. Formule ψ je *důsledkem* formule φ, píšeme $\varphi \models \psi$, právě když $\{\varphi\} \models \psi$.

Poznámka 1.2.10. Naše definice důsledku se liší od definice důsledku v [Šve02], kde je důsledek definován jemněji. Formule φ je důsledkem teorie T, právě když kdykoliv jsou všechny formule z T splněny v nějakém modelu $\mathfrak{M}$ při nějakém ohodnocení e, je i φ splněna v $\mathfrak{M}$ při ohodnocení e. Z této definice vyplývá ta naše, ale nejsou ekvivalentní.[3] Pokud bychom však požadovali uzavřenost teorie T, definice už ekvivalentní budou, jak ukazuje věta 1.2.11.

Věta 1.2.11. *Nechť T je uzavřená teorie a φ formule. Následující tvrzení jsou ekvivalentní:*

1. $T \models \varphi$.
2. *Pokud pro nějaký model $\mathfrak{M}$ a ohodnocení e je $\mathfrak{M} \models \psi[e]$ pro každé $\psi \in T$, pak $\mathfrak{M} \models \varphi[e]$.*

Důkaz. Nechť $T \models \varphi$. Pokud je každá $\psi \in T$ splněna v nějakém modelu $\mathfrak{M}$ při nějakém ohodnocení e, je každá $\psi \in T$ splněna v $\mathfrak{M}$ při každém ohodnocení, protože T je uzavřená. Protože ale $T \models \varphi$, je i φ splněna v $\mathfrak{M}$ při každém ohodnocení, specielně je tedy splněna při ohodnocení e.

Na druhou stranu nechť je pravda, že kdykoliv je každá $\psi \in T$ splněna v nějakém modelu $\mathfrak{M}$ při nějakém ohodnocení e, je $\mathfrak{M} \models \varphi[e]$. Pokud je $\mathfrak{M} \models T$, znamená to, že všechny $\psi \in T$ platí v $\mathfrak{M}$, tzn. všechny $\psi \in T$ jsou splněny při každém ohodnocení e. Z předpokladu je tedy i φ splněna při každém ohodnocení. Tedy $T \models \varphi$. ■

Věta 1.2.12. *Nechť φ, ψ jsou formule. Pokud $\models \varphi \Rightarrow \psi$, pak $\varphi \models \psi$. Pokud je navíc φ uzavřená, pak pokud $\varphi \models \psi$, pak $\models \varphi \Rightarrow \psi$.*

Důkaz. Nechť $\models \varphi \Rightarrow \psi$. Tedy $\varphi \Rightarrow \psi$ je splněna v každém modelu při každém ohodnocení. Máme ukázat, že $\varphi \models \psi$. Nechť tedy $\mathfrak{M} \models \varphi$ a pro spor nechť $\mathfrak{M} \not\models \psi[e]$ pro nějaké ohodnocení e. Zároveň s $\mathfrak{M} \models \varphi[e]$ to ale znamená, že $\mathfrak{M} \not\models (\varphi \Rightarrow \psi)[e]$. To je ale spor, protože formule $\varphi \Rightarrow \psi$ je z předpokladu splněna v každém modelu při každém ohodnocení.

Na druhou stranu nechť φ je uzavřená a nechť $\varphi \models \psi$. Máme ukázat, že $\varphi \Rightarrow \psi$ je splněna v každém modelu při každém ohodnocení. Kdyby pro spor $\mathfrak{M} \not\models (\varphi \Rightarrow \psi)[e]$ v nějakém modelu $\mathfrak{M}$ při nějakém ohodnocení e, znamená to, že $\mathfrak{M} \models \varphi[e]$ a zároveň $\mathfrak{M} \not\models \psi[e]$. Ale protože φ je uzavřená a je splněna v $\mathfrak{M}$ při nějakém ohodnocení, je splněna v $\mathfrak{M}$ při každém ohodnocení. Z předpokladu je pak ale i ψ splněna v $\mathfrak{M}$ při každém ohodnocení, to je ale spor, protože $\mathfrak{M} \not\models \psi[e]$. ■

[3] Jednoduchým protipříkladem je teorie $T = \{p(x)\}$ a formule $(\forall x)p(x)$ v jazyce s jedním unárním predikátem p. Podle naší definice je $T \models (\forall x)p(x)$, ale je zřejmé, že když je splněna $p(x)$ při nějakém ohodnocení, nemusí být splněna $(\forall x)p(x)$, takže podle definice v [Šve02] to důsledek není.

Důsledek 1.2.13. *Nechť φ, ψ jsou formule. Pokud jsou ekvivalentní, pak $\varphi \models \psi$ a $\psi \models \varphi$. Pokud jsou navíc φ, ψ uzavřené, pak pokud $\varphi \models \psi$ a $\psi \models \varphi$, pak jsou ekvivalentní.*

Věta 1.2.14. *Nechť T je teorie a φ formule. $T \models \varphi$, právě když teorie $T \cup \{\neg\overline{\varphi}\}$ není splnitelná.*

Důkaz. Nechť $T \models \varphi$ a nechť $\mathfrak{M} \models T$ je libovolný model teorie T. $T \models \varphi$ znamená, že $\mathfrak{M} \models \varphi$. To je ale totéž jako $\mathfrak{M} \models \overline{\varphi}$. Z definice splňování to znamená, že pro každé ohodnocení e je $\mathfrak{M} \models \overline{\varphi}[e]$, což je podle definice splňování ekvivalentní s $\mathfrak{M} \not\models \neg\overline{\varphi}[e]$. Jinými slovy, $\neg\overline{\varphi}$ neplatí v žádném modelu teorie T při žádném ohodnocení. Jinými slovy, teorie $T \cup \{\neg\overline{\varphi}\}$ nemá model a tedy není splnitelná.

Na druhou stranu nechť $T \cup \{\neg\overline{\varphi}\}$ není splnitelná a nechť pro spor pro nějaký model $\mathfrak{M} \models T$ a nějaké ohodnocení e je $\mathfrak{M} \not\models \varphi[e]$. Při tomto ohodnocení je také $\mathfrak{M} \not\models \overline{\varphi}[e]$. To je z definice splňování ekvivalentní s $\mathfrak{M} \models \neg\overline{\varphi}[e]$. Protože ale $\neg\overline{\varphi}$ je uzavřená a platí při nějakém ohodnocení e, platí při každém ohodnocení, tedy $\mathfrak{M} \models \neg\overline{\varphi}$. To ale není možné. $\mathfrak{M}$ je model teorie T, ve kterém navíc platí $\neg\overline{\varphi}$, jinými slovy $\mathfrak{M}$ je model pro teorii $T \cup \{\neg\overline{\varphi}\}$. To je ve sporu s předpokladem, že $T \cup \{\neg\overline{\varphi}\}$ není splnitelná. ■

1.3 Normální tvary

Motivace pro zavedení normálních tvarů je následující. Formule jsme formálně vybudovali v jazyce (kromě kvantifikátorů) s pěti logickými spojkami: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$. Například spojka $\Rightarrow$ je ale redundantní vůči zbytku, protože ji lze ekvivalentně vyjádřit pomocí $\neg$ a $\vee$. Formule $\varphi \Rightarrow \psi$ je ekvivalentní s formulí $\neg\varphi \vee \psi$. Normální tvar, kromě toho, že například neobsahuje spojky $\Rightarrow, \Leftrightarrow$, navíc zachycuje jistou strukturu formule. Ve výrokové logice lze z normálních tvarů například velmi rychle sestavit pravdivostní tabulky. Pro resoluční metodu jsou normální tvary stěžejní, protože předpokládá na vstupu formule v konjunktivním normálním tvaru.

Definice 1.3.1. Formule φ je

a) *literál*, právě když je to atomická formule nebo negace atomické formule,

b) *implikant*, právě když je konjunkcí literálů,

c) *klauzule*, právě když je disjunkcí literálů,

d) v *disjunktivním normálním tvaru*, právě když je disjunkcí implikantů,

e) v *konjunktivním normálním tvaru*, právě když je konjunkcí klauzulí.

Definice normálního tvaru je stejná pro výrokové formule i pro predikátové formule bez kvantifikátorů. Jediným rozdílem jsou atomické formule – ve výrokové logice to jsou výrokové proměnné a v predikátové logice to jsou predikáty. Každou formuli lze do normálního tvaru převést. Přesněji, pro každou formuli existuje ekvivalentní formule, která je v normálním tvaru. Pojem *ekvivalentní* je zde použit ve dvou smyslech – ekvivalence predikátových formulí a ekvivalence výrokových formulí.

Věta 1.3.2. *Nechť φ je výroková formule nebo predikátová formule bez kvantifikátorů.*

1. *Existuje formule φ_k v konjunktivním normálním tvaru ekvivalentní s φ.*
2. *Existuje formule φ_d v disjunktivním normálním tvaru ekvivalentní s φ.*

Důkaz. Ukážeme případ pro konjunktivní normální tvar a disjunktivní se udělá podobně. Nejprve se zbavíme spojek $\Rightarrow$ a $\Leftrightarrow$. Každou podformuli tvaru $\psi \Rightarrow \vartheta$ nahradíme formulí $\neg\psi \vee \vartheta$ a každou podformuli tvaru $\psi \Leftrightarrow \vartheta$ nahradíme formulí $(\psi \vee \neg\vartheta) \wedge (\neg\psi \vee \vartheta)$. Dále zajistíme, aby se negace vyskytovaly jen u atomických formulí. Každou podformuli tvaru $\neg(\psi \vee \vartheta)$ nahradíme formulí

$\neg\psi \wedge \neg\vartheta$, každou podformuli tvaru $\neg(\psi \wedge \vartheta)$ formulí $\neg\psi \vee \neg\vartheta$ a konečně každou podformuli tvaru $\neg\neg\psi$ formulí ψ. Nyní už máme původní formuli jen v jazyce se spojkami $\neg, \vee, \wedge$ a všechny negace se vyskytují jen u atomických formulí. Označme ji φ'. Indukcí podle syntaktické složitosti formule φ' sestavíme její konjunktivní tvar φ'_k, což bude konjunktivní tvar původní formule φ.

a) Je-li formule φ' atomická, je v konjunktivním normálním tvaru, tedy φ'_k je formule φ'.

b) Je-li formule φ' tvaru $\neg\psi$, pak ψ je atomická, protože negace se vyskytují jen u atomických formulí. Tedy i v tomto případě φ'_k je formule φ'.

c) Je-li formule φ' tvaru $\psi \wedge \vartheta$ a ψ_k, ϑ_k jsou konjunktivní tvary formulí ψ, ϑ, pak φ'_k je formule $\psi_k \wedge \vartheta_k$.

d) Je-li formule φ' tvaru $\psi \vee \vartheta$, pak nastávají dva případy. Pokud ψ_k, ϑ_k jsou literály nebo disjunkce literálů, pak φ'_k je formule $\psi_k \vee \vartheta_k$. Jinak je alespoň jedna z ψ_k, ϑ_k konjunkce, bez újmy na obecnosti ψ_k. Konjunktivní normální tvar získáme aplikací distributivního zákona, tj. je-li ψ_k tvaru $\chi \wedge \xi$, pak φ'_k je formule $(\chi \vee \vartheta_k)_k \wedge (\xi \vee \vartheta_k)_k$.

Všechny úpravy prováděné s formulí jsou ekvivalentní, tedy formule φ'_k a φ jsou ekvivalentní. ■

1.4 Resoluce ve výrokové logice.

Než začneme s predikátovou resolucí, připomeňme princip resoluce ve výrokové logice. Vstupem je konečná množina T výrokových formulí v konjunktivním normálním tvaru, tedy bez újmy na obecnosti množina klauzulí tvaru $(L_1 \vee \ldots \vee L_n)$, kde L_i jsou literály. Otázka je, zda je T splnitelná, tedy zda existuje ohodnocení atomických formulí v T takové, že jsou všechny klauzule z T při tomto ohodnocení splněny zároveň.

Jsou-li $(A \vee L_1 \vee \ldots \vee L_n)$ a $(\neg A \vee K_1 \vee \ldots \vee K_m)$ dvě klauzule, kde A je atomická formule a L_i, K_i literály, pak formule $(L_1 \vee \ldots \vee L_n \vee K_1 \vee \ldots \vee K_m)$ je jejich *resolventa*. Resolventa klauzulí (A) a $(\neg A)$ je prázdná klauzule, značená $\bot$ (kontradikce). Není těžké ukázat, že resolventy jsou důsledkem klauzulí, ze kterých vzešly.

Lemma 1.4.1. $(A \vee L_1 \vee \ldots \vee L_n) \wedge (\neg A \vee K_1 \vee \ldots \vee K_m) \models (L_1 \vee \ldots \vee L_n \vee K_1 \vee \ldots \vee K_m)$

Důkaz. Nechť v je splňující ohodnocení pro konjunkci klauzulí. Pokud $v(A) = 1$, pak nutně $v(K_i) = 1$ pro nějaké i, protože pravá klauzule je splněna při ohodnocení v. V tomto případě je tedy resolventa splněna. Pokud $v(A) = 0$, stejně tak je $v(L_i) = 1$ pro nějaké i, protože levá klauzule je splněna při ohodnocení v, a tedy i v tomto případě je resolventa splněna. ■

Pro množinu klauzulí T nechť $R(T)$ je sjednocení T s množinou všech resolvent všech dvojic klauzulí z T. To je zase množina klauzulí. Položme dále $R^0(T) = T$ a pro $n \in \mathbb{N}$ induktivně $R^{n+1} = R(R^n(T))$. Pro každou konečnou T je $R(T)$ konečná a existuje $k \in \mathbb{N}$ takové, že $R^{k+1}(T) = R^k(T)$. Jinými slovy, rostoucí řetěz inkluzí

$$T = R^0(T) \subseteq R^1(T) \subseteq \ldots \subseteq R^k(T) = R^{k+1}(T)$$

se stabilizuje. Pro takové k položme $R^k(T) = \mathcal{R}(T)$. Množina $\mathcal{R}(T)$ se nazývá *resoluční obal* teorie T. Z lemmatu 1.4.1 plyne, že teorie T je splnitelná, právě když $R(T)$ je splnitelná, tj. specielně T je splnitelná, právě když $\mathcal{R}(T)$ je splnitelná.

Věta 1.4.2 (o výrokové resoluci)**.** *Konečná množina výrokových klauzulí T je splnitelná právě tehdy, když její resoluční obal neobsahuje kontradikci.*

1.5 Prenexní tvary, skolemizace

Přesuňme se nyní zpátky do predikátové logiky. V predikátové logice jsou atomické formule jemnější – jsou to predikáty a ty obsahují termy. Formule mohou navíc obsahovat kvantifikátory a není na první pohled jasné, co si s nimi má resoluční metoda počít. Pokud bychom měli predikátovou teorii T, ve které jsou všechny formule bez proměnných a kvantifikátorů, mohli bychom všechny převést do konjunktivního normální tvaru a spustit výrokovou resoluci. Pokud $C_1, \ldots, C_n$ jsou všechny konstantní termy a $r_1, \ldots, r_m$ všechny predikáty v jazyce $\mathcal{L}$ a výroková resoluce našla kontradikci, znamená to, že se někde v resolučním obalu zjevila prázdná resolventa, která vzešla z nějakých dvou klauzulí $r_i(C_{j_1}, \ldots, C_{j_k})$ a $\neg r_i(C_{j_1}, \ldots, C_{j_k})$ pro nějaký k-ární predikát r_i. V opačném případě resoluční obal kontradikci neobsahuje a podle věty o výrokové resoluci existuje splňující ohodnocení v. Pak T má model. Jako univerzum stačí vzít právě všechny konstantní termy C_i. Každý k-ární predikát r realizujeme těmi k-ticemi $C_{j_1}, \ldots, C_{j_k}$, pro které je $v(r(C_{j_1}, \ldots, C_{j_k})) = 1$. Všimněme si, že pokud jazyk $\mathcal{L}$ obsahoval funkční symboly a alespoň jednu konstantu[4], pak množina konstantních termů je nekonečná, takže tento přístup by s funkčními symboly nefungoval. Tento problém se dá vyřešit postupným resolvováním konečných částí této nekonečné množiny (viz podkapitola 1.6). Problémem tedy zůstávají kvantifikátory a proměnné. Zaměříme se nejprve na kvantifikátory.

Definice 1.5.1. Řekneme, že predikátová formule φ je v *prenexním tvaru*, právě když je ve tvaru $(Q_1 x_1) \ldots (Q_n x_n)\psi$, kde Q_i jsou kvantifikátory $\forall$ nebo $\exists$ a x_i jsou proměnné a ψ neobsahuje kvantifikátory. Kvantifikátorům $(Q_1 x_1) \ldots (Q_n x_n)$ říkáme *prefix* a formuli ψ říkáme *otevřené jádro* formule φ. Pokud všechny Q_i jsou $\forall$, pak je formule *univerzální*.

Každou formuli lze převést na ekvivalentní formuli, která je v prenexním tvaru. Než ale ukážeme, jak to udělat, je potřeba se vypořádat se jmény proměnných. Přesouváním kvantifikátorů před formuli se totiž může stát, že se některé volné proměnné navážou právě na přesouvaný kvantifikátor, což může ve výsledku znamenat, že výsledná formule v prenexním tvaru nebude ekvivalentní.

Je-li t term s různými proměnnými $x_1, \ldots, x_n$, pak symbolem $t_{x_1,\ldots,x_n}[t_1, \ldots, t_n]$ označíme term, který vznikne současným nahrazením všech výskytů $x_1, \ldots, x_n$ termy $t_1, \ldots, t_n$ v tomto pořadí. Pro formuli φ označíme symbolem $\varphi_{x_1,\ldots,x_n}[t_1, \ldots, t_n]$ formuli, kde se všechny *volné* výskyty proměnných $x_1, \ldots, x_n$ nahradily termy $t_1, \ldots, t_n$ v tomto pořadí. Takto vzniklá formule je *instancí* formule φ.

Definice 1.5.2. Nechť x je proměnná, t term a φ formule. Řekneme, že term t je *substituovatelný za x do φ*, právě když pro žádnou proměnnou y termu t se proměnná x nevyskytuje volně ve φ v žádné podformuli tvaru $(\forall y)\psi$ nebo $(\exists y)\psi$.

Jinými slovy, t je substituovatelný za x do φ, právě když se žádná proměnná termu t nestane po substituci vázanou. Specielně do otevřených formulí je tedy substituovatelný každý term. Dále když napíšeme $\varphi_x[t]$, předpokládáme, že t je substituovatelný za x ve φ.

Lemma 1.5.3. *Pro každou formuli φ existuje formule φ' pro kterou platí:*

1. *Každé jméno proměnné se ve φ' vyskytuje buď jen volně, nebo jen vázaně.*
2. *Vyskytuje-li se proměnná x vázaně ve φ', pak každý její další (vázaný) výskyt je vázaný k témuž kvantifikátoru jako x.*
3. *Formule φ a φ' jsou ekvivalentní.*

[4] Pokud by naopak konstantu neměl žádnou, pak také nemá žádné termy (protože z předpokladu nemá ani žádné proměnné) a tedy žádné formule. T je v tomto případě prázdná, tedy splnitelná, protože modely prázdných teorií jsou právě modely jazyka $\mathcal{L}$.

Důkaz. Uvažujme množinu V, do které budeme přidávat postupně všechna jména proměnných, která už nesmíme použít. Na začátku jsou ve V všechna jména volných proměnných. Nyní pro každou podformuli tvaru $(Qx)\psi$ pro $x \in V$ vezmeme první proměnnou $y \notin V$ a nahradíme podformuli $(Qx)\psi$ formulí $(Qy)\psi_x[y]$ a přidáme y do V. Jinak přidáme x do V. Z této konstrukce plyne, že jsme splnili podmínky 1 a 2. Nová formule je jistě ekvivalentní s původní, protože došlo jen k přejmenování vázaných proměnných. ■

Nyní konečně k tomu, jak prenexní tvary hledat. Pokud formule v prenexním tvaru není, znamená to, že se kvantifikátor vyskytuje někde mezi spojkami $\neg, \wedge, \Rightarrow, \vee, \Leftrightarrow$. Prenexní tvar získáme aplikací *prenexních operací*. Prenexní operace jsou následující:

a) nahraď podformuli $\neg(Qx)\psi$ formulí $(\overline{Q}x)\neg\psi$

b) nahraď podformuli $\psi \Rightarrow (Qx)\vartheta$ formulí $(Qx)(\psi \Rightarrow \vartheta)$, pokud x není volná v ψ

c) nahraď podformuli $(Qx)\psi \Rightarrow \vartheta$ formulí $(\overline{Q}x)(\psi \Rightarrow \vartheta)$, pokud x není volná v ϑ

d) nahraď podformuli $(Qx)\psi \wedge \vartheta$ formulí $(Qx)(\psi \wedge \vartheta)$, pokud x není volná v ϑ

e) nahraď podformuli $(Qx)\psi \vee \vartheta$ formulí $(Qx)(\psi \vee \vartheta)$, pokud x není volná v ϑ

kde Q je nějaký kvantifikátor a $\overline{Q}$ opačný kvantifikátor (tj. $\overline{\forall}$ je $\exists$ a vice versa). Pozorný čtenář určitě postřehl, že jsme vynechali pravidlo pro spojku $\Leftrightarrow$. Ale $\varphi \Leftrightarrow \psi$ lze ekvivalentně vyjádřit pomocí $\wedge, \vee$ a $\neg$ například jako $(\varphi \vee \neg\psi) \wedge (\neg\varphi \vee \psi)$. Podle lemmatu 1.5.3 si můžeme bez újmy na obecnosti myslet, že každé jméno proměnné se ve formuli vyskytuje buď jen volně nebo jen vázaně a zároveň se nemůže stát, že by se přesunem kvantifikátoru před spojku stala nějaká proměnná x volnou v této podformuli a zároveň by existoval jiný kvantifikátor, jehož přesunem před formuli by se x navázala, protože žádné dva kvantifikátory nekvantifikují přes stejné jméno proměnné.[5] V důsledku toho při nahrazování pomocí prenexních operací b) až e) není potřeba ošetřovat případ, kdy by x byla volná v ψ (resp. v ϑ), protože se to nikdy nestane. Dá se ukázat, že prenexní operace vedou na ekvivalentní formuli, ale důkaz je spíš technický než zajímavý a proto jej neuvádíme.

Věta 1.5.4 (o prenexním tvaru)**.** *Pro každou formuli existuje ekvivalentní formule v prenexním tvaru.*

Důkaz. Viz věta 3.1.23 v [Šve02]. ■

Jak uvidíme později, univerzální kvantifikátory můžeme v prefixu formule vynechat, ale je ještě potřeba se vypořádat s těmi existenčními. Těch se lze zbavit podle následujícího pozorování (viz také [DP60], kapitola 2). Uvažme formuli $(\exists x)p(x)$. Splnění této formule v nějakém modelu $\mathfrak{M}$ s univerzem M z definice znamená existenci nějakého prvku $m \in M$ takového, že $m \in p^{\mathfrak{M}}$. To je ale to samé jako v jazyce s novým konstantním symbolem C splnit formuli $p(C)$, v tomto případě má totiž existovat nějaký prvek $C^{\mathfrak{M}} \in M$ tak, že $C^{\mathfrak{M}} \in p^{\mathfrak{M}}$.

Pokud je před existenčním kvantifikátorem univerzální kvantifikátor, jako například ve formuli $(\forall x)(\exists y)p(x, y)$, pak splnit takovou formuli v modelu $\mathfrak{M}$ s univerzem M znamená následující. Pro každé $m_1 \in M$ a ohodnocení e, které proměnné x přiřazuje m_1 existuje nějaké $m_2 \in M$ tak, že $(m_1, m_2) \in p^{\mathfrak{M}}$. To znamená totéž, že existuje funkce $f^{\mathfrak{M}} : M \to M$, která každému $m_1 \in M$ takové $m_2 \in M$ přiřazuje. Jinými slovy, pro každé $m_1 \in M$ platí $(m_1, f(m_1)) \in p^{\mathfrak{M}}$. Můžeme tedy vzít nový funkční symbol f a přepsat formuli na ekvisplnitelnou formuli $(\forall x)p(x, f(x))$.

Na těchto myšlenkách stojí *skolemizace*.

Definice 1.5.5. Nechť φ je uzavřená formule v prenexním tvaru. Její *skolemovskou variantu* φ_S definujeme následovně:

[5] Uvažujme třeba formuli $(\forall x)p(x) \wedge (\forall x)q(x)$. Po jedné prenexní operaci dostáváme $(\forall x)(p(x) \wedge (\forall x)q(x))$. Výskyt x v $p(x)$ se stal volným v podformuli $p(x) \wedge (\forall x)q(x)$ a nemůžeme provést druhou prenexní operaci. Původní formule ale právě nesplňovala bod 2 lemmatu 1.5.3.

a) je-li φ formule tvaru $(\forall x_1)\dots(\forall x_n)\psi$, kde ψ neobsahuje kvantifikátory, pak φ_S je formule φ,

b) je-li φ formule tvaru $(\forall x_1)\dots(\forall x_n)(\exists y)\psi$, nechť φ' je formule $(\forall x_1)\dots(\forall x_n)\psi_y[f(x_1,\dots,x_n)]$ v jazyce s novým n-ární funkčním symbolem f, pak φ_S je formule $(\varphi')_S$,

c) je-li φ formule tvaru $(\exists y)\psi$, nechť φ' je formule $\psi_y[C]$ v jazyce s novým konstantním symbolem C, pak φ_S je formule $(\varphi')_S$.

Nově přidaným symbolům říkáme *skolemovské funkční symboly*, resp. *skolemovské konstantní symboly.*

Každá formule má konečně mnoho existenčních kvantifikátorů a v každém kroku skolemizace ubyde alespoň jeden existenční kvantifikátor, takže tento proces jednou musí skončit. Převod na skolemovskou variantu vede na ekvisplnitelnou formuli.

Věta 1.5.6. *Nechť φ je uzavřená formule v prenexním tvaru. Pak φ a φ_S jsou ekvisplnitelné.*

Důkaz. Je-li φ_S splněna v nějakém modelu, dá se pomocí faktu $\models \varphi_x[t] \Rightarrow (\exists x)\varphi$ dokázat, že φ je také splněna ve stejném modelu. Na druhou stranu nechť $\mathfrak{M} \models \varphi$. Pro jednoduchost nechť φ má jediný existenční kvantifikátor (jinak pokračujeme indukcí) a přidáváme skolemovský funkční symbol f. Je tedy $\mathfrak{M} \models (\forall x_1)\dots(\forall x_n)(\exists y)\psi$, kde ψ už existenční kvantifikátor neobsahuje. Funkční symbol f realizujeme přirozeně. Podle definice splňování existuje pro každé $a_1,\dots,a_n$ nějaké b tak, že $\mathfrak{M} \models \psi_{x_1,\dots,x_n,y}[a_1,\dots,a_n,b]$, $f^{\mathfrak{M}}$ může tedy být nějaká výběrová funkce. Zřejmě takto vzniklá struktura je modelem φ_S. ■

1.6 Resoluce v predikátové logice

Predikátová resoluce se od výrokové liší motivací i přístupem. Výroková logika je rozhodnutelná a výroková resoluční metoda je algoritmem, který rozhodne o každé konečné množině výrokových klauzulí, zda je splnitelná. V tomto smyslu je výroková resoluce Cnf-Sat solver. Podle Churchovy věty predikátová logika rozhodnutelná není, takže predikátová resoluce nemůže být algoritmem. Navíc se liší tím, že nezkoumá splnitelnost množiny klauzulí, ale hledá spor, který je v ní obsažený (pokud tam nějaký je). Jinými slovy, zkoumá nesplnitelnost teorií. Toho lze využít při zkoumání toho, zda je nějaká formule φ důsledkem teorie T. Podle věty 1.2.14 je $T \models \varphi$, právě když teorie $T \cup \{\neg\overline{\varphi}\}$ není splnitelná. Ukážeme, že pokud nějaká teorie je nesplnitelná, resoluční metoda na to přijde. V opačném případě (pokud teorie splnitelná je) ale metoda nemusí ani doběhnout.

Vstupem do predikátové resoluční metody je, stejně jako ve výrokové resoluci, množina klauzulí. V predikátové resoluci je ale na každou klauzuli nahlíženo jako na otevřené jádro nějaké prenexní univerzální sentence. Navíc na každou klauzuli je nahlíženo jako na množinu literálů.

Pro každou formuli φ existuje ekvisplnitelná formule φ', která je otevřeným jádrem prenexní univerzální sentence. Formuli φ nejprve převedeme do prenexního tvaru a vytvoříme její uzávěr. Ten za cenu přidání nových symbolů do jazyka skolemizujeme, čímž odstraníme všechny existenční kvantifikátory. Zbylé univerzální kvantifikátory v prefixu odstraníme a to je hledané otevřené jádro φ'. Zbývá převést φ' do konjunktivního tvaru, který podle věty 1.3.2 existuje, a z něj dostat kýženou množinu klauzulí, která bude vstupem resoluční metody. Zkoumáme-li nesplnitelnost celé teorie T, stačí tento postup provést s každou formulí v T.

Navážeme s myšlenkami článku [Rob65]. Předvedeme nejprve naivní resoluci bez proměnných, která, jak později ukážeme, není moc efektivní. Později rozšíříme resoluci na klauzule s proměnnými a ukážeme, že tento postup je obecnější.

Definice 1.6.1. Pro množinu klauzulí T v jazyce $\mathcal{L}$ definujeme její *Herbrandovské univerzum* jako množinu všech konstantních termů jazyka $\mathcal{L}$. Pokud $\mathcal{L}$ neobsahuje žádný konstantní symbol (tj. množina konstantních termů je prázdná), sestává Herbandovské univerzum z konstantních termů jazyka $\mathcal{L} \cup \{C\}$, kde C je nový konstantní symbol.

Příklad 1.6.2. Obsahuje-li jazyk $\mathcal{L}$ binární funkční symbol f a unární funkční symbol g a konstanty A, B, pak Herbrandovské univerzum sestává z termů:

$$A, B, g(A), g(B), f(A,A), f(B,B), f(A,B), f(B,A), g(g(A)), g(g(B)), g(f(A,B)), \ldots$$

Zřejmě pokud $\mathcal{L}$ obsahuje alespoň jeden funkční symbol, je Herbrandovské univerzum nekonečné.

Definice 1.6.3. Nechť T je množina klauzulí a H její Herbrandovské univerzum. Pro libovolnou $P \subseteq H$ definujeme *saturaci T pomocí P* jako množinu všech klauzulí bez proměnných, které vzniknou jako instance klauzulí z T dosazením konstantních termů z P za všechny proměnné. Saturaci T pomocí P značíme $P(T)$.

Zbavili jsme se tedy proměnných a můžeme na dané klauzule pustit výrokovou resoluci. Problém ale stále je, že Herbrandovské univerzum může být nekonečné, a tak může saturace být nekonečná. Odvoláme se na Herbrandovu větu, která říká, že případný spor se nachází už v nějaké konečné části.

Věta 1.6.4 (Herbrandova,[Rob65])**.** *Nechť T je konečná množina klauzulí, H její Herbrandovské univerzum. Pak T je nesplnitelná, právě když nějaká konečná podmnožina množiny $H(T)$ je nesplnitelná.*

Důsledek 1.6.5. *Nechť T je konečná množina klauzulí s Herbrandovským univerzem H. Pokud je T nesplnitelná, pak už pro nějakou konečnou $P \subseteq H$ a nějaké $n \in \mathbb{N}$ obsahuje množina $R^n(P(T))$ kontradikci.*

Poznámka 1.6.6. Pokud bychom dokázali nějak systematicky procházet konečné fragmenty Herbrandova univerza množiny klauzulí T v nějakém pořadí $P_0 \subseteq P_1 \subseteq \ldots \subseteq P_k \subseteq \ldots$ tak, že $\bigcup_{k\geq 0} P_k = H$, a T byla nesplnitelná, tak pro nějaké $n \in \mathbb{N}$ bude saturace pomocí $\bigcup_{i=0}^{n} P_i$ obsahovat kontradikci.[6] Možností jak systematicky procházet konečné části nekonečného Herbrandovského univerza je několik. Jednou takovou je následující. Nechť P_0 jsou konstantní symboly a pro $k \geq 1$ nechť P_k je P_{k-1} společně se všemi termy, které vzniknou jako aplikace funkčních symbolů na termy z P_{k-1}. Zřejmě je pak $H = \bigcup P_k$.

Ukážeme obecnější přístup, který umožňuje pracovat s klauzulemi, které obsahují proměnné a obecně predikáty s libovolnými termy. Vraťme se znovu k výrokové resoluci a připomeňme, že cílem je hledat klauzule, které obsahují navzájem opačné literály. Například ve výrokových klauzulích $(\neg A \vee B)$ a $(A \vee C)$ jsou opačné literály A a $\neg A$ a resolventa těchto klauzulí je nová klauzule $B \vee C$. Cílem tedy bude nějak zobecnit pojem *opačné literály* i na predikátové literály – predikáty s termy uvnitř. Nadále budeme pod pojmem *výraz* myslet libovolnou formuli nebo term.

Definice 1.6.7. *Substituce* je konečná množina uspořádaných dvojic $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ kde x_i je proměnná a t_i je term neobsahující proměnnou x_i. Pro výraz e definujeme $e\sigma$ jako výraz, který vznikne současným nahrazením všech volných výskytů všech volných proměnných x_i odpovídajícími termy t_i. Pro dvě substituce $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$ a $\tau = \{y_1/s_1, \ldots, y_m/s_m\}$ definujeme jejich složení $\sigma\tau$ jako substituci $\sigma' \cup \tau'$, kde σ' jsou všechny dvojice $x_i/t_i\tau$, pro které je $t_i\tau$ term různý od proměnné x_i, a τ' obsahuje takové dvojice $y_i/s_i \in \tau$, pro které proměnná y_i není žádná z proměnných $x_1, \ldots, x_n$. Prázdnou substituci značíme ε. Pro množinu výrazů $\mathcal{E}$ a substituci σ značíme $\mathcal{E}\sigma = \{e\sigma \mid e \in \mathcal{E}\}$.

Příklad 1.6.8. Predikáty $p(x, f(y), w)$ a $\neg p(g(u), t, w)$ nejsou navzájem opačnými literály. Po substituci $\sigma = \{x/g(u), t/f(y)\}$ bude první predikát $p(g(u), f(y), w)$ a druhý $\neg p(g(u), f(y), w)$, což už jsou navzájem opačné literály.

[6] Tento přístup mimo jiné řeší problém uvedený na začátku podkapitoly 1.5.

Definice 1.6.9. Nechť $\mathcal{E}$ je množina výrazů. Její *kolizní množinu* tvoří všechny podvýrazy výrazů z $\mathcal{E}$, které začínají na pozici, na které v nějakém jiném výrazu z $\mathcal{E}$ začíná nějaký jiný podvýraz. Řekneme, že substituce σ *unifikuje* množinu $\mathcal{E}$, pokud $\mathcal{E}\sigma$ je jednoprvková. Pokud k $\mathcal{E}$ existuje taková unifikující substituce, řekneme, že $\mathcal{E}$ je *unifikovatelná*.

Existuje algoritmus, který rozhodne o unifikovatelnosti libovolné konečné množiny výrazů. Nejprve je ale potřeba se pozastavit nad technickým předpokladem. Budeme předpokládat, že symboly jazyka (tj. proměnné, konstantní symboly, funkční symboly, relační symboly) jsou nějak dobře[7] uspořádány tak, že proměnné jsou v tomto uspořádání dřív než všechny ostatní symboly. Každé takové uspořádání lze přirozeně rozšířit na dobré uspořádání na množině všech termů jazyka. Unifikační algoritmus v podobě, jak ho představuje Robinson [Rob65] právě s tímto uspořádáním pracuje (algoritmus 1). Na řádku 6 znamená *první* právě první v uspořádání zmíněném

Algoritmus 1 Unifikační algoritmus (Robinson, [Rob65])

```
1:  function UNIFY(E)
2:      k = 0, σ_0 = ε
3:  start:
4:      if Eσ_k je jednoprvková then
5:          return σ_k
6:      Nechť v_k je první a e_k první jemu odpovídající kolizní výraz množiny Eσ_k.
7:      if v_k je proměnná, která se nevyskytuje v e_k then
8:          σ_{k+1} = σ_k{v_k/e_k}
9:          k = k + 1
10:         goto start
11:     else
12:         return E není unifikovatelná
```

výše. V důsledku toho, že jsou proměnné v uvažovaném uspořádání dřív než všechny ostatní symboly, se algoritmus nejprve dívá na kolizní podvýrazy tvaru v_k, e_k, kde v_k je proměnná.

Příklad 1.6.10. Uvažujme množinu výrazů $\mathcal{E} = \{p(x), p(f(y))\}$. Unifikační algoritmus v tomto případě na řádku 6 zvolí jako $v_0 = x$ a jemu odpovídající kolizní $e_0 = f(y)$ a správně najde unifikaci $\sigma_1 = \{x/f(y)\}$. Pokud by byl výraz $f(y)$ v uspořádání před proměnnou x, nesprávně by algoritmus skončil s tím, že $\mathcal{E}$ není unifikovatelná.

Dá se ukázat, že pokud je konečná množina výrazů $\mathcal{E}$ unifikovatelná, unifikační algoritmus najde unifikující substituci. Pokud $\mathcal{E}$ unifikovatelná není, algoritmus skončí s neúspěchem. Dokonce pokud je množina $\mathcal{E}$ unifikovatelná, vrátí v jistém smyslu *nejlepší možnou* unifikující substituci. Vraťme se zpět k příkladu 1.6.8. Složením $\sigma\{w/f(g(y))\}$ získáme také unifikující substituci, ale je na první pohled zřejmé, že tato nová substituce „zbytečně“ nahrazuje i proměnnou w. Následující věta říká, že substituce nalezená unifikačním algoritmem takové „zbytečně“ dělat nebude.

Věta 1.6.11 (o unifikaci, [Rob65])**.** *Nechť $\mathcal{E}$ je konečná unifikovatelná množina výrazů. Pak unifikační algoritmus vrátí unifikující substituci σ, která unifikuje $\mathcal{E}$, a pro každou jinou unifikaci τ existuje substituce λ taková, že $\tau = \sigma\lambda$.*

Poznámka 1.6.12. Ještě než zadefinujeme resolventy pro klauzule s proměnnými, pozastavíme se nad dalším technickým předpokladem. Bude důležité, aby měly každé dvě klauzule různá jména proměnných. Jinými slovy, budeme požadovat, aby se žádné jméno proměnné nevyskytovalo ve dvou různých klauzulích. Uvažujme následující příklad. Nechť $p(f(x), y)$, $\neg p(x, f(y))$ jsou dvě klauzule, každá s jedním literálem. Pokusíme-li se unifikovat tyto dva literály spolu na navzájem opačné literály, unifikační algoritmus nalezne jako první kolizi $x, f(x)$, načež odpoví, že unifikující substituce neexistuje, protože term $f(x)$ obsahuje proměnnou x. Proměnná

[7] Uspořádání $<$ na množině X je *dobré*, právě když každá neprázdná $Y \subseteq X$ má $<$-nejmenší prvek.

x v klauzuli $p(x, f(y))$, nemá vůbec nic společného s proměnnou x v klauzuli $\neg p(f(x), y)$. Připomeňme, že klauzule jsou z pohledu resoluční metody otevřená jádra prenexních univerzálních sentencí, tedy přejmenování proměnných znamená přejmenování vázaných proměnných a to na splnitelnost nemá vliv. Bez újmy na obecnosti tedy budeme předpokládat, že jsou jména proměnných nějak standardizována. Například v první klauzuli nechť jsou všechna jména proměnných mezi $a_1, \ldots, a_m$ a v druhé klauzuli mezi $b_1, \ldots, b_n$. V tomto případě po přejmenování dostáváme $p(a_1, f(a_2))$ a $\neg p(f(b_1), b_2)$, načež $\{a_1/f(b_1), b_2/f(a_2)\}$ je zřejmě unifikující substituce. Bez přejmenování bychom tedy nenašli spor, který byl v těchto dvou klauzulích schován. Po přejmenování už jsme našli substituci, po které se staly navzájem opačnými literály a spor je nalezen. Nebudeme to explicitně psát, ale nadále budeme vždy předpokládat, že každé dvě klauzule mají různá jména proměnných, jinými slovy, že se žádné jméno proměnné nevyskytuje ve dvou různých klauzulích.

Definice 1.6.13. Nechť C, D jsou dvě různé klauzule a $L \subseteq C, M \subseteq D$ jejich neprázdné části. Nechť N je množina atomických podformulí z $L \cup M$. Nechť je dále N unifikovatelná s unifikující substitucí σ a nechť jsou $L\sigma, M\sigma$ jednoprvkové množiny obsahující navzájem opačné literály. Pak množina $((C \setminus L) \cup (D \setminus M))\sigma$ je *resolventou* klauzulí C a D.

Stejně jako ve výrokové logice pro každou množinu (predikátových) klauzulí T označme $R(T)$ jako množinu obsahující všechny klauzule z T a jejich resolventy. Dále $R^0(T) = T$ a pro $n \in \mathbb{N}$ induktivně $R^{n+1}(T) = R(R^n(T))$. Pro konečnou T je $R(T)$ konečná stejně jako ve výrokové logice, ale rostoucí řetězec inkluzí

$$T = R^0(T) \subseteq R^1(T) \subseteq \ldots \subseteq R^k(T) \subseteq R^{k+1}(T) \subseteq \ldots$$

se nemusí stabilizovat ani pro konečnou T, jak ukazuje následující příklad.

Příklad 1.6.14 ([Rob65])**.** Nechť $T = \{p(A), \neg p(x) \vee p(f(x))\}$. Pak pro každé $k \in \mathbb{N}$ obsahuje $R^k(T)$ literál $p(f^k(A))$. Tedy v tomto případě je $R^k(T) \subsetneq R^{k+1}(T)$ pro každé $k \in \mathbb{N}$.

Dá se ukázat, že tento přístup je obecnější. Klauzule získané výrokovou resolucí saturovaného Herbrandovského univerza se dají získat jako saturace resolvent zavedených v definici 1.6.13.

Lemma 1.6.15. *Nechť T je množina klauzulí, H její Herbrandovské univerzum a $P \subseteq H$. Pak $R(P(T)) \subseteq P(R(T))$.*

Důkaz. Viz kapitola 5 v [Rob65]. ■

Opačná inkluze nemusí platit ani v triviálních případech. Uvažme klauzule z poznámky 1.6.12 a množinu $T = \{p(a_1, f(a_2)), \neg p(f(b_1), b_2)\}$. Pak $P(R(T)) \not\subseteq R(P(T))$, protože $P(R(T))$ obsahuje kontradikci a $R(P(T))$ ne.

Důsledek 1.6.16 ([Rob65])**.** *Nechť T je množina klauzulí, H její Herbrandovské univerzum a $P \subseteq H$. Pak pro každé $n \in \mathbb{N}$ je $R^n(P(T)) \subseteq P(R^n(T))$.*

Všimněme si, že dosazení konstant za proměnné nemůže samo vytvořit kontradikci, tedy stačilo by se dívat jen na $R^n(T)$. Dostáváme konečně větu o predikátové resoluci:

Věta 1.6.17 (o predikátové resoluci, [Rob65])**.** *Nechť T je konečná množina predikátových klauzulí. Pak T je nesplnitelná právě tehdy, když pro nějaké $n \in \mathbb{N}$ obsahuje $R^n(T)$ kontradikci.*

Příklad 1.6.18. Ukážeme, jak predikátová resoluční metoda dokáže následující tvrzení:

Každá ireflexivní a transitivní relace je asymetrická.

Uvažujme binární relaci r a pro dva prvky x, y zapisujeme $r(x, y)$ pokud x, y jsou v relaci. Ireflexivitu a transitivitu relace r popisuje následující uzavřená teorie:

$$T = \{(\forall x)\neg r(x, x), (\forall x)(\forall y)(\forall z)(r(x, y) \wedge r(y, z) \Rightarrow r(x, z))\}.$$

Chceme ukázat, že $T \models (\forall x)(\forall y)(r(x,y) \Rightarrow \neg r(y,x))$. To je podle věty 1.2.14 ekvivalentní s tvrzením, že teorie $T' = T \cup \{\neg(\forall x)(\forall y)(r(x,y) \Rightarrow \neg r(y,x))\}$ není splnitelná.[8] Převedením formulí z T' do prenexního tvaru, skolemizací a převedením na množinu klauzulí dostáváme:

$$T' = \{\neg r(x,x), \neg r(x,y) \vee \neg r(y,z) \vee r(x,z), r(A,B), r(B,A)\}.$$

Po standardizaci jmen proměnných je ve tvaru

$$T' = \{\neg r(a_1,a_1), \neg r(b_1,b_2) \vee \neg r(b_2,b_3) \vee r(b_1,b_3), r(A,B), r(B,A)\}.$$

V první a druhé klauzuli se literály $\neg r(a_1,a_1)$ a $r(b_1,b_3)$ po substituci $\{b_1/a_1\}\{b_3/a_1\}$ stanou navzájem opačnými literály. Resolventou první a druhé klauzule je klauzule $\neg r(a_1,b_2) \vee \neg r(b_2,a_1)$. Dále se literály $r(A,B)$ a $\neg r(a_1,b_2)$ stanou navzájem opačnými pomocí substituce $\{a_1/A\}\{b_2/B\}$ a resolventa je $\neg r(B,A)$. Nově vzniklá klauzule $\neg r(B,A)$ se ale s klauzulí $r(B,A)$ resolvuje na prázdnou klauzuli, takže uvažovaná množina klauzulí T' je nesplnitelná. Tedy skutečně každá ireflexivní a transitivní relace je asymetrická.

Poznámka 1.6.19. Definice 1.6.13 povoluje při resolvování vybírat libovolné podmnožiny L, M klauzulí C, D. Stačí ale každé dvě klauzule resolvovat *po jednom literálu*, jinými slovy stačí, aby L, M byly jednoprvkové. Označme $\overline{R}(T)$ resolventy získané přesně jako v definici 1.6.13, ale L, M jsou jednoprvkové. Když resolvujeme dvě klauzule $C, D \in T$ po jednom literálu, všechny resolventy, které z nich vzešly podle definice 1.6.13, se při tomto novém přístupu objeví nejpozději v množině $\overline{R}^c(T)$, kde $c = \max\{|L|, |M|\}$. Tedy obecně $R(T) \subseteq \overline{R}^d(T)$, kde $d = \max_{C \in T}\{|C|\}$. Tento přístup je tedy stejně dobrý až na to, že se některé resolventy mohou objevit později.

Příklad 1.6.20 ([Rob65])**.** Uvažujme množinu klauzulí

$$T = \{q(x, g(x), y, h(x,y), z, k(x,y,z)), \neg q(u, v, e(v), w, f(v,w), x)\}$$

Po jednom kroku resoluce s proměnnými je nalezen spor. Kdybychom totéž dělali přístupem bez proměnných, tj. dosazováním konstantních termů podle poznámky 1.6.6 do těchto predikátů a ty pak resolvovali výrokově, spor se objeví nejdříve v množině P_5, která má řádově 10^{64} prvků, což je výpočetně neúnosné.

Tento příklad ilustruje, proč je resoluce s proměnnými lepší než saturace pomocí konstantních termů a následná výroková resoluce.

Příklad 1.6.21 ([DLL62])**.** Pomocí resoluční metody lze dokázat následující tvrzení:

Každá stejnoměrně spojitá funkce je spojitá.

Předně si zjednodušíme jazyk. Tvrzení $|x - y| < \delta$ popíšeme predikátem $s(x,y,\delta)$ a tvrzení[9] $|f(x) - f(y)| < \zeta$ predikátem $t(x,y,\zeta)$. V tomto jazyce je pak stejnoměrná spojitost popsána sentencí

$$(\forall\zeta)(\exists\delta)(\forall x)(\forall y)(s(x,y,\delta) \Rightarrow t(x,y,\zeta))$$

a spojitost je sentence

$$(\forall\zeta)(\forall x)(\exists\delta)(\forall y)(s(x,y,\delta) \Rightarrow t(x,y,\zeta)).$$

Negací druhé formule, skolemizací obou a převedením do normálního tvaru získáváme množinu klauzulí

$$T = \{\neg s(x,y,f(\zeta)) \vee t(x,y,\zeta), s(C, g(\delta), \delta), \neg t(C, g(\delta), D)\}.$$

Literály $\neg s(x,y,f(\zeta))$ a $s(C,g(\delta),\delta)$ se po substituci $\{x/C\}\{y/g(\delta)\}\{\delta/f(\zeta)\}$ stanou navzájem opačnými. Resolventa je klauzule s jediným literálem $t(C, g(f(\zeta)), \zeta)$. Literály $t(C, g(f(\zeta)), \zeta)$ a $\neg t(C, g(\delta), D)$ se po substituci $\{\delta/f(\zeta)\}\{\zeta/D\}$ stanou opačnými literály a výsledná resolventa je prázdná. Uvažovaná teorie T je nesplnitelná, tedy každá stejnoměrně spojitá funkce je spojitá.

[8] Tento postup budeme v následujících příkladech aplikovat vždy a už jej nebudeme opakovat.

[9] Symbol ε je již použit pro prázdnou substituci, použijeme ζ.

Na příkladu výše je potřeba si všimnout, že tvrzení *Každá stejnoměrně spojitá funkce je spojitá* neříká nic hlubokého o (stejnoměrně) spojitých funkcích. Na důkaz tvrzení jsme nepotřebovali axiomy uspořádaného tělesa reálných čísel ani definici absolutní hodnoty. Jde o klasický obrat prohození existenčního a univerzálního kvantifikátoru, totiž následující tautologii:

$$(\exists y)(\forall x)p(x,y) \Rightarrow (\forall x)(\exists y)p(x,y)$$

Resoluční metoda přijde správně na spor, který by nastal, kdyby platilo $(\exists y)(\forall x)p(x,y)$ a zároveň neplatilo $(\forall x)(\exists y)p(x,y)$.

Příklad 1.6.22 ([Rob65])**.** Pomocí resoluční metody lze dokázat následující tvrzení:

Pologrupa s dělením má pravý neutrální prvek.

Připomeňme, že *pologrupa* je uspořádaná dvojice $(X,*)$, kde X je množina a $*$ binární operace na X splňující asociativní zákon. Pologrupa $(X,*)$ je *pologrupa s dělením*, právě když pro každé a,b existují řešení rovnic $a*x=b$ a $x*a=b$ pro neznámou x. Pologrupa $(X,*)$ má *pravý neutrální prvek*, právě když existuje $f \in X$ takové, že pro každé $x \in X$ je $x*f=x$. Opět si zde zjednodušíme jazyk a tvrzení $x*y=z$ zapíšeme jako predikát $q(x,y,z)$. Asociativní zákon $(x*y)*z = x*(y*z)$ je pak v tomto jazyce vyjádřen těmito dvěma sentencemi:

$$(\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)(q(x,y,u) \Rightarrow (q(y,z,v) \Rightarrow (q(x,v,w) \Rightarrow q(u,z,w)))),$$
$$(\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)(q(x,y,u) \Rightarrow (q(y,z,v) \Rightarrow (q(u,z,w) \Rightarrow q(x,v,w)))).$$

Dělení je vyjádřeno těmito sentencemi:

$$(\forall a)(\forall b)(\exists x)q(x,a,b),$$
$$(\forall a)(\forall b)(\exists x)q(a,x,b).$$

Konečně, aby šlo o pologrupu, je potřeba zajistit uzavřenost operace. To je v našem jazyce sentence

$$(\forall a)(\forall b)(\exists x)q(a,b,x).$$

Existence pravého neutrálního prvku je

$$(\exists f)(\forall x)q(x,f,x).$$

Negací poslední formule, skolemizací všech a převedením do normálního tvaru získáváme množinu klauzulí

$$\begin{aligned} T = \{ & \\ & \neg q(i(x),x,i(x)), \\ & q(f(a,b),a,b), \\ & q(a,g(a,b),b), \\ & q(a,b,h(a,b)), \\ & \neg q(x,y,u) \vee \neg q(y,z,v) \vee \neg q(x,v,w) \vee q(u,z,w), \\ & \neg q(x,y,u) \vee \neg q(y,z,v) \vee \neg q(u,z,w) \vee q(x,v,w) \\ \} & . \end{aligned}$$

Jako první standardizujme proměnné.

$$
\begin{aligned}
T = \{ & \\
& \neg q(i(a_1), a_1, i(a_1)), && (1)\\
& q(f(b_1, b_2), b_1, b_2), && (2)\\
& q(c_1, g(c_1, c_2), c_2), && (3)\\
& q(d_1, d_2, h(d_1, d_2)), && (4)\\
& \neg q(e_4, e_5, e_1) \vee \neg q(e_5, e_6, e_2) \vee \neg q(e_4, e_2, e_3) \vee q(e_1, e_6, e_3), && (5)\\
& \neg q(f_4, f_5, f_1) \vee \neg q(f_5, f_6, f_2) \vee \neg q(f_1, f_6, f_3) \vee q(f_4, f_2, f_3) && (6)\\
\} &
\end{aligned}
$$

Literály $\neg q(i(a_1), a_1, i(a_1))$ a $q(e_1, e_6, e_3)$ se po substituci $\{e_1/i(e_6)\}\{a_1/e_6\}\{e_3/i(e_6)\}$ stanou navzájem opačnými literály, takže resolventou klauzulí (1) a (5) je nová klauzule

$$\neg q(e_4, e_5, i(e_6)) \vee \neg q(e_5, e_6, e_2) \vee \neg q(e_4, e_2, i(e_6)) \tag{7}$$

Dále se literály $q(f(b_1, b_2), b_1, b_2)$ a $\neg(e_4, e_5, i(e_6))$ z klauzulí (2) a (7) stanou navzájem opačnými literály po substituci $\{e_4/f(b_1, b_2)\}\{b_1/e_5\}\{b_2/i(e_6)\}$, z čehož vzejde nová klauzule

$$\neg q(e_5, e_6, e_2) \vee \neg q(f(e_5, i(e_6)), e_2, i(e_6)) \tag{8}$$

Dále se $q(f(b_1, b_2), b_1, b_2)$ a $\neg q(f(e_5, i(e_6)), e_2, i(e_6))$ z klauzulí (2) a (8) stanou navzájem opačnými literály po substituci $\{b_1/e_5\}\{b_2/i(e_6)\}\{e_5/e_2\}$ a resolventa je

$$\neg q(e_2, e_6, e_2) \tag{9}$$

Konečně, dva literály $\neg q(c_1, g(c_1, c_2), c_2)$ a $q(e_2, e_6, e_2)$ z klauzulí (3) a (9) se po substituci $\{c_1/e_2\}\{e_6/g(e_2, c_2)\}\{c_2/e_2\}$ stanou navzájem opačnými literály a resolventou je prázdná klauzule. Uvažovaná teorie T je nesplnitelná. Jinými slovy, každá pologrupa s dělením má pravý neutrální prvek.

Pozorný čtenář si jistě všiml, že jsme při resolvování nepoužili klauzule (4) a (6). Je běžné, že se při důkazu nevyužijí všechny předpoklady. Například tvrzení

Každá komutativní grupa má nanejvýš jeden neutrální prvek.

lze snadno dokázat v teorii komutativních grup, ale existence inverzních prvků, komutativita a dokonce i existence neutrálního prvku se zde při důkazu nevyužije. Nanejvýš jeden neutrální prvek má totiž i pologrupa. V našem případě by tedy pro důkaz existence pravého neutrálního prvku stačily tři axiomy:

$$
\begin{aligned}
& (\forall x)(\forall y)(\forall z)(\forall u)(\forall v)(\forall w)(q(x, y, u) \Rightarrow (q(y, z, v) \Rightarrow (q(x, v, w) \Rightarrow q(u, z, w))))\\
& (\forall a)(\forall b)(\exists x) q(x, a, b)\\
& (\forall a)(\forall b)(\exists x) q(a, x, b)
\end{aligned}
$$

Poznámka 1.6.23. Resoluční metoda vede na dokazovací systém – *resoluční kalkul*. Jeho jediné pravidlo je *pravidlo resoluce*: Z libovolných dvou klauzulí odvoď jejich resolventu. Věta o predikátové resoluci (1.6.17) je větou o úplnosti pro tento dokazovací systém. Množina klauzulí T je nesplnitelná, právě když existuje posloupnost klauzulí $\varphi_1, \ldots, \varphi_n$ taková, že φ_n je prázdná a každá z φ_i je buďto klauzule z T nebo je resolventou některých předchozích dvou. Libovolná taková posloupnost se nazývá *zamítnutí*[10] množiny T. Například v příkladu 1.6.18 je následující

[10]Na důkazy v resolučním kalkulu lze tedy pohlížet v jistém smyslu jako na *důkazy sporem*.

posloupnost zamítnutím množiny T':

$$
\begin{array}{l}
r(A, B) \\
r(B, A) \\
\neg r(a_1, a_1) \\
\neg r(b_1, b_2) \vee \neg r(b_2, b_3) \vee r(b_1, b_3) \\
\neg r(a_1, b_2) \vee \neg r(b_2, a_1) \\
\neg r(B, A) \\
\bot
\end{array}
$$

Kapitola 2
Algoritmy

V minulé kapitole jsme popsali, jak funguje predikátová resoluce teoreticky. V této kapitole popíšeme algoritmy, které použijeme při její implementaci.

2.1 Abstraktní syntaktický strom

Apriori jsou pro nás formule řetězce jistého formálního jazyka. Jako první krok tedy musíme umět poznat, že daný řetězec je skutečně validní zápis formule. Zároveň s kontrolou syntaxe lze postavit abstraktní syntaktický strom (AST). K tomuto účelu použijeme atributovaný překlad řízený LL analýzou. Jelikož jde jen o nástroj, který použijeme, a nesouvisí přímo s hlavním tématem této práce, uvedeme jen ve stručnosti gramatiku a výpočet atributů. Pro hlubší čtení o atributovaných překladech nebo LL syntaktické analýze odkazujeme čtenáře na kapitoly 4, 5 v [Aho+06]. Naše gramatika je $G = (N, \Sigma, P, F_0)$, kde

$$N = \{F_0, F_0', F_1, F_1', F_2, F_2', F_3, Q, T, T', \overline{T}, R\},$$
$$\Sigma = \{\Leftrightarrow, \Rightarrow, \wedge, \vee, \neg, \forall, \exists, i, c, (,)\},$$

přičemž i je identifikátor a má syntetizovaný atribut $i.\mathrm{str}$, jehož hodnota je řetězec daného identifikátoru. Terminál c zastupuje čárku (comma). Všechny neterminály mají syntetizovaný atribut v. Neterminály F_0', F_1', F_2' mají dědičný atribut i, neterminál R má dědičný atribut ℓ a neterminál $\overline{T}$ má dědičný atribut n. Pravidla P společně s výpočtem atributů jsou shrnuta v tabulce 2.1. Lze nahlédnout, že uvedená gramatika G je LL(1) a dokonce je L-atributovaná.

Syntetizovaný atribut v u všech neterminálů je daný uzel AST, který generují. Dědičný atribut i u neterminálů F_k' má jako hodnotu část syntaktického stromu, který se nachází před ním. Funkce Eqv, Imp, Conj, Dis, Neg, Rel, Fal, Exs, Con, Var, Fun vytvoří nový AST uzel, ve kterém je pořadě ekvivalence, implikace, konjunkce, disjunkce, negace, relační symbol, $\forall$, $\exists$, konstantní symbol, proměnná, funkční symbol. Binární operátory – ekvivalence, implikace, konjunkce, disjunkce[1] – mají jako svoje argumenty levý a pravý operand. Negace má jako svůj jediný argument svůj operand. Kvantifikátory mají dva argumenty – jméno proměnné, přes kterou kvantifikují, a formuli. V případě relačního a funkčního symbolu je prvním argumentem jméno a druhým seznam termů (argumentů). V případě konstantního symbolu a proměnné je jediným argumentem jméno. Horní indexy l, resp. r u neterminálů ve výpočtu atributů značí daný symbol na levé, resp. pravé straně pravidla.

Neterminály T' a R zajišťují parsing seznamu termů. Hodnota dědičného atributu ℓ u neterminálu R je dosavadní seznam termů. Funkce $\mathrm{list}(v)$ vytvoří jednoprvkový seznam obsahující prvek v. Funkce $\ell.\,\mathrm{pushback}(v)$ přidá prvek v na konec seznamu ℓ.

[1] Nemáme n-ární disjunkce/konjunkce, všechno je binární.

Tabulka 2.1 Pravidla gramatiky G společně s výpočtem atributů

Pravidlo	Výpočet atributů	Pravidlo	Výpočet atributů
$F_0 \to F_1 F_0'$	$F_0'.i = F_1.v$ $F_0.v = F_0'.v$	$F_3 \to (Q$	$F_3.v = Q.v$
$F_0' \to \Leftrightarrow F_1 F_0'$	$F_0'^r.i = \mathrm{Eqv}(F_0'^l.i, F_1.v)$ $F_0'^l.v = F_0'^r.v$	$F_3 \to i(T')$	$F_3.v = \mathrm{Rel}(i.\mathrm{str}, T'.v)$
$F_0' \to \varepsilon$	$F_0'.v = F_0'.i$	$Q \to \forall i) F_3$	$Q.v = \mathrm{Fal}(i.\mathrm{str}, F_3.v)$
$F_1 \to F_2 F_1'$	$F_1'.i = F_2.v$ $F_1.v = F_1'.v$	$Q \to \exists i) F_3$	$Q.v = \mathrm{Exs}(i.\mathrm{str}, F_3.v)$
$F_1' \to \Rightarrow F_2 F_1'$	$F_1'^r.i = \mathrm{Imp}(F_1'^l.i, F_2.v)$ $F_1'^l.v = F_1'^r.v$	$Q \to F_0)$	$Q.v = F_0.v$
$F_1' \to \varepsilon$	$F_1'.v = F_1'.i$	$T' \to TR$	$R.\ell = \mathrm{list}(T.v)$ $T'.v = R.v$
$F_2 \to F_3 F_2'$	$F_2'.i = F_3.v$ $F_2.v = F_2'.v$	$R \to cTR$	$R^l.\ell.\,\mathrm{pushback}(T.v)$ $R^r.\ell = R^l.\ell$ $R^l.v = R^r.v$
$F_2' \to \wedge F_3 F_2'$	$F_2'^r.i = \mathrm{Conj}(F_2'^l.i, F_3.v)$ $F_2'^l.v = F_2'^r.v$	$R \to \varepsilon$	$R.v = R.\ell$
$F_2' \to \vee F_3 F_2'$	$F_2'^r.i = \mathrm{Dis}(F_2'^l.i, F_3.v)$ $F_2'^l.v = F_2'^r.v$	$T \to i\overline{T}$	$\overline{T}.n = i.\mathrm{str}$ $T.v = \overline{T}.v$
$F_2' \to \varepsilon$	$F_2'.v = F_2'.i$	$\overline{T} \to \varepsilon$	**if** $\mathrm{isupornum}(\overline{T}.n[0])$ **then** $\overline{T}.v = \mathrm{Con}(n)$ **else** $\overline{T}.v = \mathrm{Var}(n)$
$F_3 \to \neg F_3$	$F_3^l.v = \mathrm{Neg}(F_3^r.v)$	$\overline{T} \to (T')$	$\overline{T}.v = \mathrm{Fun}(\overline{T}.n, T'.v)$

Neterminály T a $\overline{T}$ zajišťují parsing samotného termu. Dědičný atribut n neterminálu $\overline{T}$ je jméno, které se dál použije na konstantní symbol nebo proměnnou v případě pravidla $\overline{T} \to \varepsilon$ nebo na funkční symbol v případě pravidla $\overline{T} \to (T')$. Dodržujeme konvenci zavedenou v poznámce 1.1.2 – konstantní symboly začínají velkým písmenem nebo číslicí a proměnné malým. Pro řetězec s je $s[0]$ jeho první symbol a funkce $\mathrm{isupornum}(ch)$ vrací `true`, pokud je ch velké písmeno nebo číslice a `false` jinak. Ve výpočtu atributů v pravidle $\overline{T} \to \varepsilon$, tedy pokud jméno n začíná na velké písmeno nebo číslici, vracíme konstantní symbol a jinak vracíme proměnnou. Gramatika respektuje také konvenci levé asociativity, priority operátorů a uzávorkování z poznámky 1.1.5.

Od této chvíle tedy předpokládejme, že máme formuli reprezentovanou jako AST. Nadále budeme pod pojmem *výraz* opět myslet formuli nebo term. Pro výraz e budeme značit $\mathrm{children}(e)$ všechny syny daného výrazu v daném AST. Pro binární operátor budeme značit $e.\mathrm{left}$, resp. $e.\mathrm{right}$ jeho levý, resp. pravý operand, pro unární operátor (včetně kvantifikátorů[2]) bude $e.\mathrm{arg}$ jeho jediný operand a pro funkční a relační symboly bude $e.\mathrm{args}$ seznam jejich argumentů. Každá proměnná, konstantní symbol, funkční symbol a relační symbol má jméno $e.\mathrm{name}$. Pro kvantifikátor q existuje funkce $q.\mathrm{swap}()$ která vymění kvantifikátor $\forall$ za $\exists$ a vice versa, $q.\mathrm{varname}$ je jméno proměnné, přes kterou q kvantifikuje. Pro výraz e je $\mathrm{qt}(e)$ množina všech kvantifikátorů v e, $\mathrm{var}(e)$ množina všech proměnných v e a $\mathrm{names}(e)$ množina všech jmen proměnných, konstantních, funkčních a relačních symbolů v e.

2.2 Volné a vázané proměnné

Začneme algoritmem, který pozná volné a vazané proměnné (algoritmus 2). Na začátku jsou všechny proměnné nastavené jako volné. V průchodu AST si udržujeme seznam Q kvantifiká-

[2]Kvantifikátor považujeme za unární operátor s jediným argumentem – formulí (viz poznámka 1.1.5).

torů nad daným uzlem v opačném pořadí, než v jakém jsme je navštívili. Když narazíme na proměnnou, podíváme se na první kvantifikátor v Q, který kvantifikuje přes proměnnou se stejným jménem, jako je jméno aktuální proměnné a na ten se proměnná naváže. Pokud žádný takový kvantifikátor neexistuje, proměnná zůstává volná. Navázání proměnné e na kvantifikátor q realizuje funkce $\mathrm{bind}(e, q)$, která zaznamená navázání proměnné e na kvantifikátor q tak, aby se na proměnné e poznalo, že je navázaná právě na q a u kvantifikátoru q se poznaly všechny výskyty proměnných, které se na q vážou.

Algoritmus 2 Volné a vazané proměnné

```
1:  function BindVars(e, Q)
2:      if e je kvantifikátor then
3:          Q.pushfront(e)
4:          BindVars(e, Q)
5:          Q.popfront()
6:      else if e je proměnná then
7:          for all q ∈ Q do
8:              if q.varname = q.name then
9:                  bind(e, q)
10:                 break
11:     else
12:         for all e′ ∈ children(e) do
13:             BindVars(e′, Q)
```

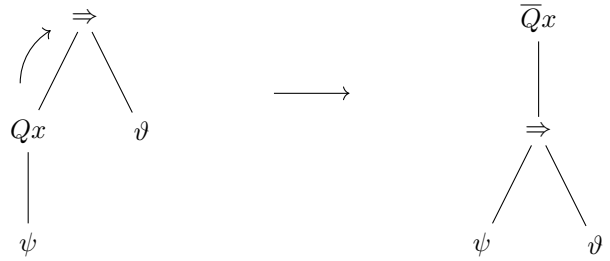
2.3 Prenexní tvar

Máme navázané proměnné na kvantifikátory a můžeme se pustit do případného přejmenovávání jmen a přípravu na převod do prenexního tvaru. Nadále budeme potřebovat generovat nová jména proměnných (a u skolemizace funkčních a konstantních symbolů). Pro tento účel definujeme funkce genvar(), gencon(), genfun(), které vrátí nějaký nový symbol, který lze v aktuálním kontextu použít pořadě jako jméno proměnné, konstantního symbolu, funkčního symbolu. Pro kvantifikátor q definujme funkci $q.\mathrm{rename}(x)$, která přejmenuje vázanou proměnnou u kvantifikátoru q na x zároveň se všemi výskyty vázanými právě k tomuto kvantifikátoru. Algoritmus 3 přejmenuje všechny vázané proměnné ve výrazu e tak, aby se jmenovaly jinak než všechny ostatní proměnné v dané formuli (viz lemma 1.5.3).

Algoritmus 3 Normalizace jmen

```
function NormalizeNames(e)
    for all q ∈ qt(e) do
        if q.varname ∈ names(e) then
            q.rename(genvar())
```

Když máme takto normalizovaná jména vázaných proměnných, můžeme sestavit prenexní tvar. Z pohledu AST znamenají prenexní operace jisté rotace. Například nahrazení $(Qx)\psi \Rightarrow \vartheta$ za $(\overline{Q}x)(\psi \Rightarrow \vartheta)$ znamená následující rotaci doprava:



Dokud se tedy jako argument nějakého operátoru vyskytuje kvantifikátor, provede se na něm příslušná rotace. Speciální případ je operátor ⇒, kde se při rotaci doprava navíc vymění kvantifikátor a u operátoru ¬ se vymění kvantifikátor vždy. Průchod končí u relačních symbolů, ty jsou samy o sobě v prenexním tvaru. Algoritmus 4 ukazuje případ pro spojku ⇒. Pro ostatní spojky se to udělá podobně.

Při vytváření prenexního tvaru se stojí za to zamyslet nad pořadím, ve kterém kvantifikátory rotujeme ze synů. Zřejmě pro formuli $(\forall x)p(x) \wedge (\exists y)q(y)$ existují dva různé prenexní tvary – $(\forall x)(\exists y)(p(x) \wedge q(y))$ a $(\exists y)(\forall x)(p(x) \wedge q(y))$. Teoreticky je tedy jedno, jestli nejprve rotujeme levého, nebo pravého syna, protože tyto dva tvary jsou ekvivalentní. Protože ale budeme dále formuli skolemizovat, budeme prenexní tvar dělat tak, aby existenční kvantifikátory byly co nejvíc vepředu. Skolemizace takové formule, kde jsou existenční kvantifikátory víc vepředu, znamená přidávání funkčních symbolů s menší aritou, což zjednoduší následnou resoluci. V tomto případě bychom dokonce místo funkčního symbolu přidávali konstantní symbol. Úprava kódu je následující. Společně s aktuálním uzlem si ve funkci prenex(e) posíláme informaci o tom, který kvantifikátor preferujeme vepředu. Výchozí stav je ∃. V případech, kdy se kvantifikátor mění (levý operand u ⇒ a operand u ¬), tuto informaci v podstromu prohodíme (například když jsme před ¬ preferovali ∃, v podstromě preferujeme ∀ a vice versa). Nyní při návratu z rekurze nejprve zkusíme, jestli je nalevo nebo napravo takový kvantifikátor, který se ve výsledku v kořeni zjeví jako ten, který jsme preferovali, a až potom se díváme na zbylé kvantifikátory.

Algoritmus 4 Prenex pro ⇒

```
1:  function PRENEX(e)
2:      e.left = PRENEX(e.left)
3:      e.right = PRENEX(e.right)
4:      if e.left je kvantifikátor then
5:          q = e.left
6:          e.left = q.arg
7:          q.arg = PRENEX(e)
8:          q.swap()
9:          return q
10:     else if e.right je kvantifikátor then
11:         q = e.right
12:         e.right = q.arg
13:         q.arg = PRENEX(e)
14:         return q
15:     else
16:         return e
```

2.4 Skolemizace

Než začneme se skolemizací, definujme funkci $\mathrm{sub}(e, x, t)$, která ve výrazu e nahradí všechny volné výskyty proměnné x za t. Skolemizaci ukazuje algoritmus 5. Vstupem je vždy uzavřená formule v prenexním tvaru. Pokud po převodu do prenexního tvaru uzavřená není, posbíráme všechny její volné proměnné a za každou volnou proměnnou x přidáme před formuli kvantifikátor $(\forall x)$. Tím vytvoříme její uzávěr. Ve skolemizaci je pak potřeba jen upravit kvantifikátory tak, jak to požaduje definice skolemovské varianty (1.5.5). Všechny univerzální kvantifikátory, které kvantifikují přes proměnnou, která se v jádru formule skutečně vyskytuje[3], si poznamenáváme do seznamu Q v pořadí, jak je procházíme, a když narazíme na existenční kvantifikátor, rozlišujeme dva případy. Pokud před ním žádný univerzální kvantifikátor nestojí (tj. Q je prázdný), vytvoříme nový konstantní symbol, jinak vytvoříme n-ární funkční symbol, kde n je počet kvantifikátorů před tímto existenčním kvantifikátorem (totiž velikost seznamu Q). Konečně, tento konstantní, resp. funkční symbol substituujeme do příslušné podformule. Na řádku 18 ještě řádně odstraníme navázání proměnných na tento kvantifikátor, protože funkce $\mathrm{sub}(e, x, t)$ nahrazuje jen volné proměnné. Jakmile projdeme všechny kvantifikátory a narazíme na otevřené jádro formule (řádek 22), průchod skončí a vrátíme pouze e. Jinými slovy, formule bez kvantifikátorů je sama svojí skolemovskou variantou.

Algoritmus 5 Skolemizace

```
1:  function SKOLEM(e, Q)
2:      if e je univerzální kvantifikátor then
3:          if e kvantifikuje přes alespoň jednu proměnnou then
4:              Q.pushback(e)
5:          e.arg = SKOLEM(e.arg)
6:          return e
7:      else if e je existenční kvantifikátor then
8:          if Q je prázdný then
9:              c = gencon()
10:             t = nový konstantní symbol se jménem c
11:         else
12:             f = genfun()
13:             a' = prázdný seznam
14:             for all q ∈ Q do
15:                 v = nová proměnná se jménem q.varname
16:                 a'.pushback(v)
17:             t = nová funkce se jménem f a argumenty a'
18:         odstraň navázání u všech proměnných, které se vázaly ke kvantifikátoru e
19:         sub(e.arg, e.varname, t)
20:         return e.arg
21:     else
22:         return e
```

2.5 Konjunktivní normální tvar

Po skolemizaci následuje převod otevřeného jádra formule do konjunktivního normálního tvaru. Převod rozdělíme do tří kroků přesně jako v důkazu věty 1.3.2. Nejprve převedeme celou for-

[3] Pokud je kvantifikováno přes nějakou proměnnou, která se v jádru nevyskytuje, nemusí být tato proměnná zahrnutá ve skolemovském funkčním symbolu. Například ve formuli $(\forall x)(\exists z)p(z)$ je $(\forall x)$ zbytečně. Skolemovská varianta je přísně vzato $(\forall x)p(f(x))$, ale ekvisplnitelná je i formule $(\forall x)p(C)$, resp. $p(C)$.

muli do jazyka $\{\wedge, \vee, \neg\}$. Tu převedeme na ekvivalentní formuli, kde se negace vyskytuje jen u atomických formulí – predikátů – a tu pak převedeme do konjunktivního normálního tvaru.

Algoritmus 6 ukazuje funkci $\mathrm{cdn}(e)$, která danou formuli e bez kvantifikátorů převede do jazyka $\{\wedge, \vee, \neg\}$. Implikace $\varphi \Rightarrow \psi$ nahradíme za $\neg\varphi \vee \psi$ a ekvivalence $\varphi \Leftrightarrow \psi$ nahradíme za $(\varphi \vee \neg\psi) \wedge (\neg\varphi \vee \psi)$. Algoritmus dělá jen to, co musí. Pokud při průchodu narazí na spojku $\Rightarrow$ nebo $\Leftrightarrow$, nahradí ji příslušnou formulí. U každé jiné spojky případně jen upraví odkazy na syny, kteří se mohli změnit. Průchod končí u relačních symbolů, ty jsou v jazyce $\{\wedge, \vee, \neg\}$ triviálně. Na řadě je převod formule do tvaru, kde se negace vyskytuje jen u atomických formulí – propa-

Algoritmus 6 Převod do $\{\wedge, \vee, \neg\}$

```
1:  function CDN(e)
2:      if e je spojka ⇒ then
3:          return ¬CDN(e.left) ∨ CDN(e.right)
4:      else if e je spojka ⇔ then
5:          l = CDN(e.left)
6:          r = CDN(e.right)
7:          return (l ∨ ¬r) ∧ (¬l ∨ r)
8:      else if e je jiná logická spojka then
9:          for all e' ∈ e.children do
10:             e' = CDN(e')
11:         return e
12:     else                                          ▷ e je predikát
13:         return e
```

gace negace k predikátům. Myšlenka je velmi podobná jako u funkce $\mathrm{cdn}(e)$. Podformule tvaru $\neg(\varphi \wedge \psi)$ nahradíme za $(\neg\varphi \vee \neg\psi)$, $\neg(\varphi \vee \psi)$ nahradíme za $(\neg\varphi \wedge \neg\psi)$ a $\neg\neg\varphi$ nahradíme za φ. Funkci $\mathrm{propneg}(e)$ ukazuje algoritmus 7. Na vstupu předpokládá formuli v jazyce $\{\wedge, \vee, \neg\}$ a bez kvantifikátorů.

Algoritmus 7 Propagace negace k predikátům

```
1:  function PROPNEG(e)
2:      if e je spojka ¬ then
3:          if e.arg je ¬ then
4:              return PROPNEG(e.arg.arg)
5:          else if e.arg je ∧ then
6:              return PROPNEG(¬e.left) ∨ PROPNEG(¬e.right)
7:          else if e.arg je ∨ then
8:              return PROPNEG(¬e.left) ∧ PROPNEG(¬e.right)
9:          else                                      ▷ e.arg je predikát
10:             return e
11:     else if e je binární spojka then
12:         e.left = PROPNEG(e.left)
13:         e.right = PROPNEG(e.right)
14:         return e
15:     else                                          ▷ e je predikát
16:         return e
```

Konečně tedy samotný převod do konjunktivního normálního tvaru (algoritmus 8). Algoritmus na vstupu předpokládá formuli v jazyce $\{\wedge, \vee, \neg\}$, bez kvantifikátorů a s negacemi jen u predikátů. Algoritmus odráží důkaz indukcí podle syntaktické složitosti věty 1.3.2. Je-li formule e ve tvaru $\psi \wedge \vartheta$ a známe konjunktivní tvary ψ a ϑ (označme je ψ_k a ϑ_k), pak $\psi_k \wedge \vartheta_k$ je kon-

junktivní tvar formule e. Pokud je formule e tvaru $\neg\psi$, je v konjunktivním tvaru. Formule ψ je v tomto případě predikát, protože negace se vyskytují jen u atomických formulí. Stejně tak pokud je formule e sama o sobě predikát. Pokud je formule e tvaru $\psi \vee \vartheta$, pak pokud známe konjunktivní tvary ψ_k a ϑ_k, nastávají dva případy. Pokud ani jedna z ψ_k, ϑ_k není konjunkce, je každá z nich buď literál nebo disjunkce literálů. V tom případě je e součástí nějaké klauzule a konjunktivní tvar pro e je $\psi_k \vee \vartheta_k$. V opačném případě je alespoň jedna z ψ_k, ϑ_k konjunkce. Formule e je tedy bez újmy na obecnosti tvaru $(\chi \wedge \xi) \vee \vartheta_k$ (pokud je ϑ_k konjunkce, udělá se to podobně). Zde použijeme distributivní zákon a budeme rekurzivně hledat konjunktivní tvar $\chi \vee \vartheta_k$ a $\xi \vee \vartheta_k$. V tomto případě je tedy konjunktivní tvar $(\chi \vee \vartheta_k)_k \wedge (\xi \vee \vartheta_k)_k$.

Algoritmus 8 Konjunktivní normální tvar

```
1:  function CNF(e)
2:      if e je spojka ∧ then
3:          e.left = CNF(e.left)
4:          e.right = CNF(e.right)
5:          return e
6:      else if e je spojka ∨ then
7:          ψ = CNF(e.left)
8:          ϑ = CNF(e.right).
9:          if (ψ je literál nebo ∨) a zároveň (ϑ je literál nebo ∨) then
10:             e.left = ψ
11:             e.right = ϑ
12:             return e
13:         else
14:             if ψ je spojka ∧ then
15:                 χ = ψ.left
16:                 ξ = ψ.right
17:                 return CNF(χ ∨ ϑ) ∧ CNF(ξ ∨ ϑ)
18:             else                                   ▷ ϑ je spojka ∧
19:                 χ = ϑ.left
20:                 ξ = ϑ.right
21:                 return CNF(ψ ∨ χ) ∧ CNF(ψ ∨ ξ)
22:     else                                           ▷ e je ¬ nebo predikát
23:         return e
```

V konjunktivním normálním tvaru se všechny uzly AST s $\wedge$ vyskytují nad $\vee$. Můžeme teď projít AST znovu a zastavit průchod na $\vee$ a tím získat všechny klauzule. Z množiny formulí už jsme tedy schopni získat množinu klauzulí.

2.6 Unifikace

Nyní popíšeme unifikační algoritmus. Předvedeme variantu pro unifikaci dvou literálů. Podle poznámky 1.6.19 stačí brát z každé klauzule jeden literál a později tento přístup při resoluci použijeme. Algoritmus pro unifikaci dvou literálů (algoritmus 9) paralelně prochází oba syntaktické stromy a pokud narazí na kolizi, kterou jde vyřešit substitucí (tj. v jednom ze stromů je na kolizním místě proměnná), provede substituci v obou stromech. Z tohoto důvodu je potřeba si kromě aktuálních výrazů posílat i původní kořeny celých výrazů (v pseudokódu jako r_1 a r_2). Volání funkce na výrazy e_1, e_2 tedy bude vypadat takto: unify(e_1, e_2, e_1, e_2, U). U je odkaz na nějaký seznam, ve kterém si poznamenáváme dosavadní substituci a pokud nějaká větev průchodu selže (tj. zahlásí, že podvýrazy nejsou unifikovatelné), substituce zapsaná v U je neplatná. Do seznamu U si zapisujeme jen dvojice proměnná/term a v rámci seznamu U neřešíme skládání

substitucí. Pokud bychom chtěli danou substituci provést na výrazu e, zavoláme postupně pro všechny dvojice $x/t \in U$ funkci $\text{sub}(e, x, t)$. Nadále budeme pro jednoduchost značit unifikaci dvou výrazů jako volání funkce $\text{unify}(e_1, e_2)$, která bude vracet objekt[4], který je buď prázdný (v tom případě výrazy e_1, e_2 nejsou unifikovatelné) nebo obsahuje validní substituci[5], která unifikuje výrazy e_1, e_2. Pozastavme se ještě nad důležitou vlastností tohoto algoritmu. Pokud je jeden výraz instancí druhého, jsou také navzájem unifikovatelné. Navíc se to v takovém případě dá poznat z unifikující substituce, kterou algoritmus vrátí. Tato vlastnost se nám bude dál hodit.

Věta 2.6.1. *Nechť e_1, e_2 jsou dva výrazy s různými proměnnými. Pak e_2 je instancí e_1, právě když funkce* $\text{unify}(e_1, e_2)$ *popsaná v algoritmu 9 vrátí unifikující substituci U takovou, že pro každé $x/t \in U$ je $x \in \text{var}(e_1) \setminus \text{var}(e_2)$.*

Důkaz. Na jednu stranu pokud algoritmus vrátí takovou substituci, zřejmě je e_2 instancí e_1. Na druhou stranu pokud je e_2 instancí e_1, všechny kolize lze vyřešit substitucí za nějakou proměnnou v e_1. Když je e_1 proměnná, algoritmus na řádku 14 zjistí, že e_2 je substituovatelný výraz za e_1. e_2 nemůže obsahovat proměnnou e_1, protože z předpokladu jsou jména disjunktní. Na řádku 20 substituuje za proměnnou e_1 term e_2. Nemůže nastat případ, kdy by v rekurzi byl například podstrom e_1 funkce a e_2 proměnná, protože e_2 je z předpokladu instance e_1. ■

[4]Takový objekt může být v jazyce C++ například ukazatel nebo od C++17 šablona `std::optional<T>`.

[5]Taková validní substituce může být například prázdná. Je tedy rozdíl, jestli funkce vrátí prázdný objekt (resp. `NULL`) nebo prázdný seznam. V prvním případě unifikace neexistuje, v druhém případě je unifikující substitucí prázdná substituce.

Algoritmus 9 Unifikační algoritmus pro dva literály

```
1:  function UNIFY(e1, e2, r1, r2, U)
2:      if e1 je ¬ then
3:          if e2 je ¬ then
4:              return UNIFY(e1.arg, e2.arg, r1, r2, U)
5:          else
6:              return unifikující substituce neexistuje
7:      else if e1 je predikát then
8:          if e2 je predikát stejné arity a jména jako e1 then
9:              for i = 0, i < e1.args.size, i = i + 1 do
10:                 if !UNIFY(e1.args[i], e2.args[i], r1, r2, U) then
11:                     return unifikující substituce neexistuje
12:         else
13:             return unifikující substituce neexistuje
14:     else if e1 je proměnná then
15:         if e2 není term nebo e2 je term a obsahuje proměnnou jména e1.name then
16:             return unifikující substituce neexistuje
17:         else if e2 je proměnná a zároveň e1.name = e2.name then
18:             return                                  ▷ Není co unifikovat
19:         else
20:             sub(r1, e1, e2), sub(r2, e1, e2), U.pushback(e1/e2)
21:     else if e1 je konstantní symbol then
22:         if e2 není term a nebo e2 je konstantní symbol a e1.name ≠ e2.name then
23:             return unifikující substituce neexistuje
24:         else if e2 je proměnná then
25:             sub(r1, e2, e1), sub(r2, e2, e1), U.pushback(e2/e1)
26:         else
27:             return unifikující substituce neexistuje
28:     else if e1 je funkce then
29:         if e2 není term then
30:             return unifikující substituce neexistuje
31:         else if e2 je proměnná a e1 neobsahuje proměnnou se jménem e2.name then
32:             sub(r1, e2, e1), sub(r2, e2, e1), U.pushback(e2/e1)
33:         else if e2 je funkce stejného jména a arity jako e1 then
34:             for i = 0, i < e1.args.size, i = i + 1 do
35:                 if !UNIFY(e1.args[i], e2.args[i], r1, r2, U) then
36:                     return unifikující substituce neexistuje
37:         else
38:             return unifikující substituce neexistuje
```

2.7 Resoluce

V tuto chvíli už jsme připraveni na resoluci. Nabízí se naivně resolvovat podle definice 1.6.13 a věty 1.6.17. Můžeme vzít množinu klauzulí T a pro každé $C, D \in T$ zkusit všechny podmnožiny $L \subseteq C, M \subseteq D$. Všechny nové resolventy pak můžeme přidat do T a postup opakovat. Můžeme se však vyhnout iteraci přes všechny podmnožiny L, M a resolvovat po jednom literálu za cenu pozdějšího objevení resolvent (viz poznámka 1.6.19). Tento přístup byl vyzkoušen a i na jednoduchých příkladech byl dost neefektivní.[6] Dalšího zefektivnění dosáhneme, pokud nebudeme do T přidávat resolventu, která je instancí[7] nějaké klauzule $C \in T$. Určitě taky nemá smysl resolvovat dvě stejné klauzule víc než jednou. Dále si lze všimnout, že do T nemusíme přidávat tautologie. Nakonec před přidáním nové klauzule, která má jeden literál, můžeme zkontrolovat, zda už tato není ve sporu s nějakou jinou klauzulí (také s jedním literálem), přesněji zda se po nějaké substituci nestanou navzájem opačnými literály. I po aplikaci těchto vylepšení běžel algoritmus i na jednoduchých příkladech poměrně dlouho.[8]

2.7.1 Heuristika

Na příkladech 1.6.18 a 1.6.22 si lze všimnout, že jsme sporu poměrně rychle dosáhli tak, že jsme nejprve resolvovali krátké klauzule s málo proměnnými a novou klauzuli – resolventu – jsme opět resolvovali, dokud to šlo. Apriori nemají klauzule v T žádné uspořádání, ale nabízí se je procházet v pořadí od nejkratších a s co nejméně proměnnými. Chceme tedy na množině T zavést uspořádání $\prec$, přičemž $\prec$-nejmenší klauzule by měly být nejkratší a mít co nejméně proměnných. Pro klauzuli C označme $v(C)$ počet různých proměnných obsažených v C. Pro libovolné dvě klauzule C, D definujme[9]

$$C \prec D \overset{\text{def}}{\iff} (\max\{|C|, v(C)\}, v(C), |C|) <_{\text{lex}} (\max\{|D|, v(D)\}, v(D), |D|)$$

kde $<_{\text{lex}}$ je ostré lexikografické uspořádání na $\mathbb{N}\times\mathbb{N}\times\mathbb{N}$. Budeme tedy klauzule resolvovat v tomto pořadí, a kdykoliv objevíme novou resolventu, zařadíme ji do T vzhledem k tomuto uspořádání a začneme resolvovat znovu. Samozřejmě před každým resolvováním je potřeba nějak standardizovat jména, aby v každých dvou klauzulích byla různá jména proměnných (viz poznámka 1.6.12). To realizuje funkce disjointnames(T). Abychom zbytečně neresolvovali nějaké literály v nějakých klauzulích dvakrát, budeme si udržovat tabulku Res, kde si pro každou dvojici klauzulí C, D uložíme dvojici indexů $(i, j) \in \mathbb{N} \times \mathbb{N}$, přičemž význam je následující. $\text{Res}(C, D) = (i, j)$, právě když další resolvování klauzulí C a D má začít od i-tého literálu v C a od j-tého literálu v D (indexujeme od nuly). Vždy když následuje resolvování klauzulí C, D, podíváme se, jestli máme záznam $\text{Res}(C, D)$. Pokud ano, začínáme od indexů $(i, j) = \text{Res}(C, D)$. Pokud ne, začínáme od $(i, j) = (0, 0)$. Pro jednoduchost tedy předpokládejme, že Res je funkce, která pro dvě klauzule C, D buď vrátí korektní dvojici indexů literálů, od kterých se má pokračovat, nebo vrátí $(0, 0)$, pokud se C, D spolu ještě nikdy předtím neresolvovaly. Všechny tyto myšlenky shrnuje algoritmus 10. Pro klauzuli C a přirozené číslo $k < |C|$ je L^C_k k-tý literál v klauzuli C. Funkce nextpair(i, j) realizuje posun dvojice indexů (i, j) klauzulí (C, D) na další dvojici literálů, které se mají resolvovat. Cykly na řádcích 4 a 6 vždy postupují v lexikografickém pořadí dvojice indexů k, ℓ, resp. i, j.

Po aplikaci této heuristiky už program na příkladech 1.6.18, 1.6.21, 1.6.22 doběhl téměř hned a také s poměrně rozumným počtem resolvent.

[6] Například pro teorii z příkladu 1.6.22 program ani nedoběhl.

[7] Rozpoznat, zda je jedna klauzule instancí druhé není tak triviální, jak by se na první pohled mohlo zdát (viz další podkapitola).

[8] Na příkladu 1.6.18 doběhl téměř hned, ale při resolvování vzniklo spoustu zbytečných resolvent, na příkladu 1.6.22 stále nedoběhl.

[9] Tomuto uspořádání se též říká *maximo-lexikografické*.

Algoritmus 10 Resoluce

```
1: function RESOLVE(T,Res)
2:     Nechť C_1, ..., C_n jsou klauzule v T uspořádané podle relace ≺
3:     disjointnames(T)
4:     for all k, ℓ, k < ℓ ≤ n do
5:         (i', j') = Res(C_k, C_ℓ)
6:         for all i, j, i' ≤ i < |C_k|, j' ≤ j < |C_ℓ| do
7:             U = Unify(L_i^{C_k}, L_j^{C_ℓ})
8:             if U then
9:                 C_new = C_k ∪ C_ℓ \ {L_i^{C_k}, L_j^{C_ℓ}}
10:                for all L ∈ C_new do
11:                    for all x/t ∈ U do
12:                        sub(L, x, t)
13:                if C_new = ∅ then
14:                    return T je nesplnitelná
15:                if |C_new| = 1 then
16:                    for all C ∈ T, |C| = 1 do
17:                        Nechť K ∈ C je jediný literál v C
18:                        Nechť L ∈ C_new je jediný literál v C_new
19:                        if ∃σ taková, že Lσ a Kσ obsahují navzájem opačné literály then
20:                            return T je nesplnitelná.
21:                if C_new obsahuje navzájem opačné literály then        ▷ C_new je tautologie
22:                    continue
23:                if C_new je instancí nějaké klauzule z T then
24:                    continue
25:                Zařaď C_new do T vzhledem k uspořádání ≺.
26:                Res(C_k, C_ℓ) = nextpair(i, j)
27:                return RESOLVE(T, Res)
28:            Res(C_k, C_ℓ) = (|C_k|, |C_ℓ|)
29:    return T je splnitelná                                  ▷ žádné nové resolventy
```

2.8 Instance

Již jsme téměř hotovi s algoritmem pro resoluční metodu. Poslední netriviální věc, kterou je potřeba vyřešit, je rozpoznání instancí klauzulí. Přísně vzato je formule φ instancí formule ψ, právě když φ vznikla z ψ substitucí termů za některé proměnné v ψ. Stačilo by projít příslušné dva syntaktické stromy a případnou substituci provést. To bychom uměli dokonce pomocí našeho unifikačního algoritmu a věty 2.6.1. Jednu klauzuli můžeme ale považovat za instanci druhé až na pořadí literálů uvnitř. Nabízí se naivní $O(n!n^{O(1)})$ algoritmus, kde $n = |C|$, který pro každou permutaci literálů v C v čase $O(n^{O(1)})$ zjistí, zda jde o instance. Stačilo by ale permutovat jen literály stejných jmen a dostali bychom složitost $O(k_1! \dots k_m! n^{O(1)})$, kde m je počet různých literálů[10] a k_i je počet výskytů i-tého literálu v C. Prakticky to uděláme následovně. Uvažujme na vstupu dvě klauzule C, D (s různými proměnnými) jakožto množiny literálů. Můžeme předpokládat, že $|C| = |D|$, jinak to instance nejsou triviálně. Vezmeme první $L \in C$ a zkusíme najít odpovídající L' tak, že L' je instance L. Pokud je L' skutečně instance L a σ je příslušná substituce, tak σ aplikujeme na celou klauzuli C a v rekurzivním volání zakážeme substituci za ty proměnné, které se vyskytují v termu t pro nějaké $x/t \in \sigma$. Pokračujeme s instancí problému pro

[10] V tomto smyslu jsou například literály $q(x)$ a $q(f(h(w)))$ *stejné* literály a literály $p(x)$ a $\neg p(x)$ *různé*. Jde jen o jméno a případnou negaci u literálu.

$(C \setminus \{L\})\sigma, (D \setminus \{L'\})\sigma$. Pokud žádné takové L' neexistuje, zřejmě nejde o instance. Pro zjištění instancí dvou literálů můžeme použít unifikaci a větu 2.6.1. Tyto myšlenky popisuje algoritmus 11. Množina F je množina právě zakázaných proměnných, které se už dříve substituovaly tak, aby to mohla vůbec být instance. Na začátku algoritmu je prázdná.

Algoritmus 11 Instance

```
1:  function Instances(C,D,F)
2:      if C = ∅ then                                   ▷ pak taky D = ∅, protože |C| = |D|
3:          return true
4:      Nechť L ∈ C je nějaký literál v C.
5:      for all L' ∈ D do
6:          U = unify(L, L')
7:          if U then
8:              V = ⋃_{x/t∈U} {x}     ▷ V je množina proměnných, které se přejmenovaly v unifikaci
9:              if V ⊆ var(L) a zároveň F ∩ V = ∅ then      ▷ V obsahuje jen proměnné z L a
    zároveň nebylo substituováno za nic z F
10:                 F' = F
11:                 C' = C \ L
12:                 D' = D \ L'
13:                 for all K ∈ C' do
14:                     for all x/t ∈ U do
15:                         sub(K, x, t)
16:                 for all x/t ∈ U do
17:                     F' = F' ∪ var(t).
18:                 if Instances(C', D', F') then
19:                     return true
20:                 else
21:                     continue
22:     return false
```

Kapitola 3
Implementace

V minulé kapitole jsme popsali algoritmy potřebné k implementaci resoluční metody. V této kapitole popíšeme jejich implementaci v jazyce C++. Všechny zdrojové kódy a další soubory spojené s implementací jsou k nalezení na přiloženém médiu a nebo v online gitlab repozitáři na odkaze `https://gitlab.fit.cvut.cz/dvora125/flas`. Jak už to u každého software bývá, je dost pravděpodobné, že se zdrojové kódy budou do budoucna měnit a vylepšovat. Všechny soubory, které zde budeme popisovat, odpovídají stavu na přiloženém médiu nebo commitu v repozitáři s hashem `b24c754ce51c22635d764d7626c0d6873c8c73ff` z 4. května 2021.

3.1 Abstraktní syntaktický strom

Ústředním objektem celého programu je abstraktní syntaktický strom, který reprezentuje formule. AST je realizován následujícími třídami:

```
ast_node
|-fla_node
| |-fla_node_relation
| |-fla_node_binop
| | |-fla_node_equivalence
| | |-fla_node_implication
| | |-fla_node_disjunction
| | `-fla_node_conjunction
| `-fla_node_unop
|   |-fla_node_negation
|   `-fla_node_qtifier
`-term_node
  |-term_node_function
  |-term_node_variable
  `-term_node_constant
```

Stromová struktura tříd značí dědičnost, tzn. třída `fla_node` dědí z `ast_node`, `fla_node_relation` dědí z třídy `fla_node` atd. Třída `fla_node_binop` má (členské) proměnné `m_left` a `m_right` typu `fla_node*` a třída `fla_node_unop` proměnnou `m_arg` typu `fla_node*`. Všechny identifikátory (u proměnných, funkčních symbolů, relačních symbolů, konstantních symbolů) ukládáme jako proměnné `m_name` typu `std::string`. U kvantifikátoru je proměnná `m_variable` typu `std::string` což je jméno proměnné, přes kterou kvantifikátor kvantifikuje. Pro seznam argumentů v `term_node_function` a `fla_node_relation` je použita proměnná `m_args` typu

`std::list<term_node*>`. Pro zachycení vázaných proměnných je ve třídě `fla_node_qtifier` proměnná `m_bindings` typu `std::list<term_node_variable*>`, což je seznam ukazatelů na proměnné, které jsou vázané na tento kvantifikátor a ve třídě `term_node_variable` je proměnná `m_binding` typu `const fla_node_qtifier*`, což je ukazatel na kvantifikátor, ke kterému je proměnná vázána. Pokud je volná, je nastaven na `nullptr`. V kvantifikátoru si ukládáme typ kvantifikátoru v proměnné `m_type`, která je typu `qt_type`, což je `enum` se dvěma hodnotama: `FORALL` a `EXISTS`. Význam všech těchto proměnných v jednotlivých třídách je zřejmý.

Veškeré algoritmy nad syntaktickým stromem jsou realizovány pomocí *visitor patternu*, který funguje následovně. Existuje třída[1] `astv_base`, která vypadá takto:

```
class astv_base
{
  public:
    virtual ~astv_base() = default;
    virtual void visit(term_node_constant* n);
    virtual void visit(term_node_function* n);
    virtual void visit(term_node_variable* n);
    virtual void visit(fla_node_binop* n);
    virtual void visit(fla_node_relation* n);
    virtual void visit(fla_node_unop* n);
    virtual void visit(fla_node_negation* n);
    virtual void visit(fla_node_qtifier* n);
    virtual void visit(fla_node_equivalence* n);
    virtual void visit(fla_node_disjunction* n);
    virtual void visit(fla_node_conjunction* n);
    virtual void visit(fla_node_implication* n);
};
```

a uzel `ast_node` má následující pure virtual metodu:

```
class ast_node
{
  public:
    virtual void accept(astv_base&) = 0;
    ...
};
```

V důsledku tedy každý konkrétní[2] uzel přepíše tuto metodu a tím „přijme“ visitora právě nad svým typem, jinými slovy zavolá příslušnou přetíženou variantu metody `visit` daného visitora. Implementace je všude stejná a například ve třídě `fla_node_disjunction` vypadá takto:

```
void fla_node_disjunction::accept(astv_base& v)
{
    v.visit(this);
}
```

Pro každý algoritmus nad AST nyní stačí naimplementovat třídu, která dědí z třídy `astv_base` a přepisuje virtuální metody `visit` nad příslušnými typy. Navštívení uzlu `n` visitorem `v` se pak realizuje voláním `n->accept(v)`.

Pozorný čtenář si jistě všiml, že metody `visit` v `astv_base` nejsou pure virtual. Všichni visitoři budou mít výchozí chování takové, že projdou AST až do listů a neudělají nic. To se bude hodit například při implementaci funkce bindvars(e, Q) z podkapitoly 2.2. Tuto funkci zřejmě bude zajímat jen chování na uzlech `term_node_variable` a `fla_node_qtifier` a všechny ostatní

[1] Ve skutečnosti existuje ještě třída `astv_c_base` a třída `astv_base` uvedená v textu se jmenuje `astv_m_base` přičemž `astv_c_base` navštěvuje konstantní AST, tj. při průchodu se nemění, tedy všechny metody `visit` přijímají nikoliv ukazatel `node*` nýbrž `const node*`. Jak už to v C++ bývá, je potřeba implementovat `const` i `mutable` variantu.

[2] tj. uzel, jehož třída není abstraktní

potřebuje jen projít. Stejně tak visitor implementující převod do konjunktivního tvaru se vůbec nemusí starat o uzly s termy. Výchozí chování tedy implementujeme následovně. Všechny konkrétní binární operátory se budou chovat jako `fla_node_binop` a ten má výchozí chování takové, že navštíví svého levého a pravého syna a neudělá nic. Všechny konkrétní unární operátory se budou chovat jako `fla_node_unop` a ten se bude chovat tak, že navštíví svůj argument. Výchozí chování `fla_node_relation` a `term_node_function` je navštívení všech svých argumentů a konečně výchozí chování `term_node_variable` a `term_node_constant` (listů syntaktického stromu) je nic nedělání. V kódu to vypadá následovně:

```
void astv_base::visit(fla_node_implication* n)
{
    this->visit(static_cast<fla_node_binop*>(n));
}
/*Simillar for fla_node_conjunction,
fla_node_disjunction, fla_node_equivalence*/
...
void astv_base::visit(fla_node_negation *n)
{
    this->visit(static_cast<fla_node_unop*>(n));
}
/*Simillar for fla_node_qtifier*/
...
void astv_base::visit(fla_node_binop* n)
{
    n->m_left->accept(*this);
    n->m_right->accept(*this);
}
void astv_base::visit(fla_node_unop* n)
{
    n->m_arg->accept(*this);
}
void astv_base::visit(fla_node_relation* n)
{
    for(auto& arg : n->m_args)
        arg->accept(*this);
}
/*Simillar for term_node_function*/
...
void astv_base::visit(term_node_variable * n) { }
/*Simillar for term_node_constant*/
...
```

Visitoři sami o sobě tedy neumí vracet žádnou hodnotu z funkce visit(n). To se dá ale zařídit například tak, že se v třídě, která z visitora dědí, zavede proměnná, která tento výsledek držet bude, a ten pomocí nějaké funkce po navštívení uzlu vrátí. Pro tento účel definujeme šablonu[3] `astv<T>`.

```
template<typename T>
class astv : public astv_base
{
    public:
        virtual T get_result() = 0;
};
```

Tato šablona reprezentuje visitora, který po navštívení nějakého uzlu vrátí výsledek typu `T`. Všichni konkrétní visitoři tedy budou dědit z šablony `astv<T>` a případně, pokud by visitor

[3]Ve skutečnosti jsou opět dvě: `astv_c` a `astv_m`, viz poznámka 1

nevracel žádnou hodnotu, lze specifikovat `T = void`. Navštívení nějakého uzlu a následné vrácení hodnoty můžeme pak realizovat funkcí `apply_visit`, která vytvoří daného visitora `TVst` se svými argumenty `VstArgs`, navštíví uzel `n` zavoláním `n->accept(v)` a vrátí výsledek pomocí `v.get_result()`.

```
template<typename TVst, typename ...VstArgs>
auto apply_visit(ast_node * n, VstArgs&& ... args)
{
    TVst v(std::forward<VstArgs>(args)...);
    n->accept(v);
    return v.get_result();
}
```

Navštívení uzlu visitorem se tedy realizuje výhradně skrz šablonu `apply_visit`. Jako ukázku implementace visitora uvedeme třídu `astv_binder`, která realizuje právě funkci bindvars(e) z podkapitoly 2.2. Implementace všech ostatních visitorů lze dohledat v souborech s prefixem `astv`.

```
class astv_binder : public astv<void>
{
  public:
    virtual void visit(fla_node_qtifier* n) override
    {
        n->m_bindings.clear();
        m_active_qts.push_front(n);
        n->m_arg->accept(*this);
        m_active_qts.pop_front();
    }
    virtual void visit(term_node_variable* n) override
    {
        n->m_binding = nullptr;
        for(auto& qt : m_active_qts)
            if(qt->m_variable == n->m_name)
            {
                n->m_binding = qt;
                qt->m_bindings.push_back(n);
                break;
            }
    }
    virtual void get result() override { }
  private:
    std::list<fla_node_qtifier*> m_active_qts;
};
```

3.2 Lexer a parser

Další důležitou součástí programu je lexer a parser. Lexer má za úkol převést řetězec – posloupnost znaků – na posloupnost lexikálních tokenů. Pro zápis formulí chceme použít ideálně ASCII znaky a tak pro vyjádření spojek $\forall, \exists, \wedge, \vee, \Rightarrow, \Leftrightarrow, \neg$ použijeme znaky `A`, `E`, `&`, `|`, `>`, `=`, `!`. Identifikátory jsou libovolné neprázdné sekvence znaků z množiny $\{\mathtt{a}, \ldots, \mathtt{z}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{1}, \ldots, \mathtt{9}, \mathtt{\_}\}$ kromě řetězců `A` a `E`, které jsou rezervovány pro kvantifikátory. Identifikátory proměnných a konstantních symbolů rozlišujeme přesně podle konvence přijaté v poznámce 1.1.2. Znak podtržítka považujeme za malé písmeno. Každý lexikální token má svůj typ, to je enum:

```
enum class lexical_token_type
{
    EQUIVALENCE, IMPLICATION, CONJUNCTION, DISJUNCTION, FORALL,
```

```
    EXISTS, NEGATION, IDENTIFIER, LPAR, RPAR, COMMA, EOI,
};
```

a samotný token reprezentuje třída `lex_token` držící typ tokenu a řetězec pro identifikátor.

```
class lexical_token
{
    ...
  private:
    std::string m_str;
    lex_token_type m_type;
};
```

Samotný lexer realizuje třída `lexer`, která v sobě drží referenci na nějaké rozhraní, které umí „načíst další znak“. Vhodným kandidátem pro toto rozhraní je v C++ třída `std::istream`. Veřejné rozhraní třídy `lexer` sestává z jediné metody `read_lex` která vrací další lexikální token ze vstupu.

```
class lexer
{
  public:
    lexer(std::istream& input);
    lexical_token read_lex();
  private:
    std::istream& m_input;
    ...
};
```

Metoda `read_lex` je implementována pomocí konečného automatu, který přijímá regulární jazyk, který popisuje právě lexikální tokeny. Detailněji se lze o implementaci lexikální analýzy dočíst v [Aho+06] v kapitole 3. Jakákoliv chyba lexeru (například neznámý znak) je řešena pomocí výjimky.

Parser je implementován pomocí rekurzivního sestupu (recursive descent) na základě gramatiky popsané v podkapitole 2.1 a je realizován třídou `parser`. Každý neterminál v gramatice má svoji metodu a syntetizované atributy jsou návratovou hodnotou metody a dědičné atributy jsou parametry metody. Stejně jako `lexer` v sobě drží referenci na vstupní proud znaků, `parser` v sobě drží referenci na `lexer` – vstupní proud lexikálních tokenů. Celé veřejné rozhraní třídy `parser` tvoří metoda `parse_fla()` která vrací `fla_node*` – ukazatel na naparsovaný syntaktický strom. Všechny chyby jsou řešeny opět výjimkami.

```
class parser
{
  public:
    parser(lexer& lex);
    fla_node* parse_fla();
  private:
    lexer& m_lex;
    ...
};
```

3.3 Algoritmy

Jak již bylo řečeno v podkapitole 3.1, všechny algoritmy nad AST jsou implementovány pomocí visitorů a jejich implementace následuje pseudokódy popsané v kapitole 2. Všichni tito visitoři jsou v souboru `algorithms.cpp` spojeni do příslušných algoritmů. Funkce z `algorithms.cpp` jsou pak volané z funkce `main` a tvoří celý program.

3.4 Použití programu

Společně se zdrojovými kódy existuje `Makefile` který pro utilitu `make` definuje, jak se mají zdrojové kódy zkompilovat do spustitelného souboru `formulas`. Program je určen pro používání z terminálu a vstupem a výstupem je výhradně text. Použití programu je následující:

```
formulas [-cdepsv] [-T theory]
```

Program bez přepínačů přečte ze standardního vstupu formule a ukončí se. Na každém řádku očekává právě jednu formuli. Další chování lze měnit právě pomocí přepínačů:

- přepínače `-c` a `-d` – program vytiskne s každou naparsovanou formulí její konjunktivní, resp. disjunktivní normální tvar
- přepínač `-e` – program každou naparsovanou formuli vytiskne znovu na standardní výstup přičemž vynechá všechny zbytečné závorky (viz poznámka 1.1.5)
- přepínač `-p` – program vytiskne s každou naparsovanou formulí její prenexní tvar
- přepínač `-s` – program vytiskne s každou naparsovanou formulí její skolemovskou variantu
- přepínač `-T` s argumentem `theory` – program přečte soubor `theory`, který obsahuje teorii T ve formě formulí – na každé řádce jednu. O každé další formuli ze standardního vstupu se program pokusí pomocí resoluce rozhodnout, zda je důsledkem teorie T
- přepínač `-v` – program detailněji na chybový výstup rozepisuje kroky, které dělá, většinou se kombinuje s ostatními přepínači

O něco detailnější popis chování programu nabízí dokumentace v podobě standardní manuálové stránky, která byla zároveň s programem napsána, jde o soubor `formulas.1`. Kromě kompilace do spustitelného souboru `formulas` lze pomocí podpříkazu `make test` zkompilovat zdrojové kódy do spustitelného souboru `test`, který spustí unit testy jednotlivých visitorů a algoritmů. Dále jsou přítomny textové soubory s prefixem `example`, které slouží jednak jako příklady použití programu a jednak jako test programu z pohledu uživatele. Volání podpříkazu `make check` spustí program `formulas` právě s těmito příklady.

Závěr

Cílem práce bylo popsat resoluční metodu z článku [Rob65] a následně ji implementovat v jazyce C++. Resoluční metodu jsme popsali a navrhli jsme konkrétní algoritmy, které jsme při její implementaci využili. Samotnou resoluční metodu jsme nejprve naimplementovali naivně, což už na jednoduchých příkladech bylo zoufale pomalé – program ani nedoběhl. V podkapitole 2.7.1 jsme navrhli heuristiku, která na našich vybraných příkladech dobře zafungovala a resoluci zefektivnila. Určitě lze vymyslet protipříklad, na kterém tato heuristika fungovat nebude, ale žádný explicitní jsme nenašli.

Do budoucna se tedy nabízí detailněji prozkoumat heuristiku, kterou jsme navrhli, a případně prozkoumat další heuristiky pro resolvování a jejich dopad nejen na příklady 1.6.18, 1.6.21 a 1.6.22.

Dále se nabízí uplatnit další zefektivnění, které zmiňuje Robinson v článku [Rob65] v kapitole 7. Z množiny T můžeme vyřadit klauzule obsahující *pure literály*. Literál $L \in C \in T$ je *pure literál v* T pokud pro žádnou $D \in T, D \neq C$ neexistuje resolventa klauzulí $\{L\}, D$. Dále můžeme z T vyřadit takovou klauzuli D pro kterou existuje nějaká $C \in T, C \neq D$ taková, že pro nějakou substituci σ je $C\sigma \subseteq D$. Těmito vylepšeními jsme se v práci nezabývali a určitě by stálo za to je hlouběji prozkoumat.

Literatura

[Aho+06] AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

[DLL62] DAVIS, Martin; LOGEMANN, George; LOVELAND, Donald. A Machine Program for Theorem-Proving. *Commun. ACM*. 1962, roč. 5, č. 7, s. 394–397. ISSN 0001-0782. Dostupné z DOI: 10.1145/368273.368557.

[DP60] DAVIS, Martin; PUTNAM, Hilary. A Computing Procedure for Quantification Theory. *J. ACM*. 1960, roč. 7, č. 3, s. 201–215. ISSN 0004-5411. Dostupné z DOI: 10.1145/321033.321034.

[Chu36] CHURCH, Alonzo. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*. 1936, roč. 1, č. 1, s. 40–41. Dostupné z DOI: 10.2307/2269326.

[Rob65] ROBINSON, John Alan. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*. 1965, roč. 12, č. 1, s. 23–41. ISSN 0004-5411. Dostupné z DOI: 10.1145/321250.321253.

[Šve02] ŠVEJDAR, Vítězslav. *Logika: neúplnost, složitost, nutnost*. Academia, 2002.

[Tur37] TURING, Alan Mathison. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*. 1937, roč. s2-42, č. 1, s. 230–265. Dostupné z DOI: 10.1112/plms/s2-42.1.230.

Příloha A

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `impl` ..... adresář s implementací
- `thesis`
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `thesis.pdf` ..... text práce ve formátu PDF