Automatizace dedukce ve znalostních systémech

texty pro distanční studium

RNDr. PaedDr. Hashim Habiballa, Ph.D.

Ostravská univerzita v Ostravě, Přírodovědecká fakulta
Katedra informatiky a počítačů

OBSAH

1 LOGIKA A AUTOMATIZACE DEDUKCE 4

1.1 DEDUKCE VE VÝROKOVÉ LOGICE 5
1.2 PREDIKÁTOVÁ LOGIKA 9
1.3 DEDUKCE A JEJÍ FORMALIZACE 12
1.4 APLIKACE FORMÁLNÍ DEDUKCE V PROGRAMOVÁNÍ 16

2 VÍCEHODNOTOVÁ LOGIKA 21

2.1 KLASICKÁ A FUZZY MATEMATIKA 21
2.2 STRUKTURY PRAVDIVOSTNÍCH HODNOT 25
2.3 PREDIKÁTOVÁ VÍCEHODNOTOVÁ LOGIKA 29
2.4 LINGVISTICKÁ LOGIKA 32

3 DEDUKTIVNÍ SYSTÉM 38

3.1 STRUKTURA DEDUKTIVNÍHO SYSTÉMU 38
3.2 INFERENČNÍ JÁDRO 39
3.3 DEFINICE JAZYKA 39
3.4 SYNTAKTICKÁ ANALÝZA A PŘEKLAD 42
3.5 PŘÍKLAD STROMOVÉ REPREZENTACE ZNALOSTÍ 47
3.6 KONSTRUKCE STROMU 49
3.7 OPTIMALIZACE VÝRAZU 52

4 STRATEGIE PRO AUTOMATIZOVANOU DEDUKCI 65

4.1 ZÁKLADNÍ PRAVIDLA PRO LOGICKOU DEDUKCI 66
4.2 NEKLAUZULÁRNÍ REZOLUCE V PREDIKÁTOVÉ LOGICE 69
4.3 NEKLAUZULÁRNÍ REZOLUCE PRO FUZZY LOGIKU 77
4.4 KLASICKÉ REZOLUČNÍ STRATEGIE VE DVOUHODNOTOVÉ LOGICE 81
4.5 DETEKCE KONSEKVENTNÍCH FORMULÍ (DCF ALGORITMUS) 84

5 MODELOVÉ DEDUKTIVNÍ SYSTÉMY 87

5.1 GENERALIZED RESOLUTION DEDUCTIVE SYSTEM 87
5.2 PROGRAM PROVER9 106

LITERATURA 115

1 Logika a automatizace dedukce

V této kapitole se dozvíte:

- Symbolická logika
- Výroková logika
- Splnitelnost a rozhodnutelnost
- Logická dedukce
- Syntaktické metody pro dedukci

Po jejím prostudování byste měli být schopni:

- Pracovat a modelovat realitu pomocí výrokové logiky.
- Provádět dedukci různými formálními metodami.

Klíčová slova této kapitoly:

Symbolická logika, logická dedukce, formální dedukce

Doba potřebná ke studiu: 8 hodin

Průvodce studiem

Studium této kapitoly je poměrně náročné zejména pro ty z Vás, kteří dosud nemají žádné znalosti z logiky. V takovém případě Vám zřejmě některé příklady budou připadat obtížně pochopitelné, ovšem nenechte se tím odradit, neboť pochopením této části se Vám usnadní studium následujících kapitol.
Na studium této části si vyhraďte alespoň 8 hodin.

Logika je oproti jiným disciplínám teoretické informatiky vědou velmi starou a její kořeny lze najít již ve starověku. Tehdy byla spjata spíše s filozofickými otázkami než s rodící se matematikou. Přesto otázky, které byly v té době aktuální, jsou v mnohém stejné jako je uplatnění logiky v informatice. Na logiku je možné se dívat v kontextu mnoha věd a výsledný pohled může být velmi odlišný. Touto vědou se zabývají filozofové, právníci, matematici i informatici a jejich zájem a cíl bývá velmi odlišný. Čtenář může sám posoudit, jak odlišné mohou tyto pohledy být - stačí si pročíst například filozoficky a informaticky orientovanou knihu o logice. Přesto lze najít společnou snahu o **modelování lidského úsudku**. I když matematik logiku spíše používá k formulaci a dokazování vlastností matematických objektů, pro informatika je logika nejen nástrojem, ale i plnoprávnou disciplínou zkoumanou v rámci teoretické informatiky.

Přístupy při modelování úsudku založené na logice patří do rodiny symbolických přístupů, existují pak také tzv. konekcionistické přístupy (např. umělé neuronové sítě), ale těmito přístupy se nebudeme zabývat. Tyto přístupy se často inspirují biologickými procesy a snaží se je modelovat matematicky. Často se objevují také velmi úspěšné kombinace obou přístupů.

1.1 Dedukce ve výrokové logice

Klasickou logikou myslíme formalismus, který je v různých podobách znám již stovky let. Lidé jsou schopni komunikovat a formulovat své myšlenky v přirozeném jazyce a dále je zpracovávat a **dedukovat závěry**. Právě přirozený jazyk je však poměrně složitý pro stroje, které by měly simulovat tento způsob reprezentace znalostí. Proto logika nabízí různé úrovně zjednodušení formulace znalostí do co nejmenšího množství exaktně definovaných symbolů. Druhou stránkou je motivace, abychom z těchto statických znalostí reprezentovaných symboly dokázali také odvozovat závěry. K tomu slouží různé symbolické metody, z nichž některé si popíšeme v tomto článku. Proveďme analogii s právními předpisy. Samotné zapsané zákony jsou sice velice užitečné, ale k čemu by nám byly, pokud by nebylo možné je interpretovat a zjišťovat, zda se například některý subjekt nedopustil porušení zákona. To můžeme provést jedině tak, že dedukujeme závěr (zda někdo jedná protizákonně nebo ne) z předpokladů (fakta popisující situaci a dané zákony).

Nejjednodušším formalismem vhodným pro reprezentaci znalostí je výroková logika. U každé logiky si musíme uvědomit, že má svou **syntaxi** a **sémantiku**. Zjednodušeně můžeme říci, že syntaxe je definicí symbolů a jejich používání (bez ohledu na význam těchto symbolů). Sémantika je pak chováním těchto symbolů s ohledem na smysl daných znalostí (např. pravdivost daných tvrzení). Právě možnost oddělit syntaxi a sémantiku ,po formulaci a dokázání potřebných vztahů mezi nimi, dává logice význam v informatice a pro automatizaci úsudku.

Syntaxe výrokové logiky pracuje se symboly zastupujícími elementární výroky (např. a,b,c,...), logické konstanty (pravda a nepravda - T,⊥) a dále je umožňuje spojovat do složitějších formulí pomocí symbolů logických **spojek** a pomocných symbolů (např. $\wedge$ konjunkce, $\rightarrow$ implikace). Sémantika pak popisuje smysl těchto použitých symbol a pojmy, které souvisejí s popisem tohoto smyslu (tzv. interpretací formulí - v případě klasické logiky to může být buď formule pravdivá nebo nepravdivá). Jednotlivým výrokům můžeme přiřadit logickou hodnotu podle toho, jak se tyto syntaktické elementy chovají v našem modelovaném případě z reality, příp. to můžeme udělat také zcela nesmyslně. V obou případech pak můžeme uvažovat jaká bude interpretace celých složených formulí. Záleží to na použitých unárních a binárních spojkách. Čtenář se zřejmě setkal se spojkami jako je negace $\neg$(opak ve smyslu pravdivosti), konjunkce $\wedge$ (současná platnost výroků), disjunkce $\vee$ (platnost alespoň jednoho z výroků), implikace $\rightarrow$ (podmínka), ekvivalence $\leftrightarrow$ (identická pravdivost výroků). Binárních spojek existuje samozřejmě více - v elektrotechnice se kupříkladu používají spojky jako je negace konjunkce (nand). Formule tedy v tomto případě platí (je interpretována jako pravdivá), pokud neplatí dva výroky současně (jinak řečeno pokud alespoň jeden neplatí). Druhým případem, který se zase vyskytuje často v reálných situacích je tzv. vylučovací nebo. V přirozené řeči často používáme výrok typu "buď ... anebo ...", který spíše odpovídá situaci, že musí platit alespoň jeden z výroků, ale

zároveň nesmí platit oba současně. Nejde tedy o disjunkci, ale spojku, kterou někdy také označujeme jako negaci ekvivalence. Interpretaci logických spojek můžeme přehledně realizovat pomocí tabulky, ve které jsou vypsány všechny možné kombinace ohodnocení pravdivosti elementárních výroků. Interpretaci pravda označíme 1 a nepravda 0. Podívejme se na příklad identického chování negace ekvivalence a vylučovacího nebo.

výrok A	výrok B	vylučovací nebo	ekvivalence	negace ekvivalence
0	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

V klasické logice existuje pouze konečný (a navíc velmi omezený) počet spojek. Stačí se zamyslet nad všemi možnostmi, které mohou nastat a dojdeme k tomu, že pro 2 výroky ohodnocené 4 možnostmi pravda/nepravda existuje 2^4 možností dvouhodnotové interpretace tedy 16 možných spojek.

Sémantika jazyka výrokové logiky umožňuje na základě interpretace formule definovat další důležité pojmy.

Pokusme se nyní na příkladu namodelovat pomocí výrokové logiky jednoduchou situaci.

Příklad 1.

K soudu byli předvedeni tři podezřelí z loupeže - A, B a C. Při výslechu se zjistily tyto skutečnosti:

1. Do případu nebyl zapleten nikdo jiný než A, B a C.
2. A pracuje vždycky alespoň s jedním společníkem.
3. C je nevinen.

Nejprve si musíme stanovit, co jsou elementární výroky. Jelikož se vyjadřujeme k vině a nevině podezřelých, bude rozumné pomocí výroků A, B a C symbolizovat, že daný podezřelý je vinen nebo nevinen. Pak můžeme zapsat větu 1. jako složený výrok vyjadřující pomocí disjunkce, že alespoň jeden z podezřelých je vinen. Druhá věta může být popsána formulí pomocí implikace (podmínky), která říká, že je-li vinen A, pak musí být vinen také alespoň jeden z podezřelých B, C. Výsledné formule výrokové logiky jsou tedy následující:

1. $A \vee B \vee C$, 2. $A \rightarrow (B \vee C)$, 3. $\neg C$

Samotná reprezentace znalostí je pouze polovičním úspěchem při pokusu o modelování úsudku. Dalším nevyhnutelným krokem je možnost ověřovat, zda nějaké tvrzení vyplývá z daných předpokladů. K tomu si nejprve musíme definovat další pojmy sémantiky. Jde o tzv. **splnitelnost** (nesplnitelnost) a **platnost** formulí. Splnitelná formule je laicky řečeno taková, která má vůbec smysl pro určité ohodnocení výroků. Tedy taková formule musí alespoň pro nějakou kombinaci ohodnocení výroků pomocí pravda/nepravda dávat interpretaci pravda. Jinak je taková formule nesplnitelná resp. v logice se takové formuli říká kontradikce. Pokud bychom se obrátili zpátky na interpretační tabulku, pak by tabulka ve sloupci interpretace splnitelné formule musela mít alespoň v jednom řádku symbol 1. Některé ze splnitelných formulí pak mohou být ještě na vyšším sémantickém stupni a mohou být pravdivé pro všechna možná ohodnocení. Takovým formulím se pak říká platná formule resp. v logice je zažitý pojem tautologie. Podívejme se na příklad.

Příklad 2.

Mějme platné tvrzení: "Pokud prší, beru si deštník." Jestliže namodelujeme tvrzení, že "prší" pomocí P a "beru si deštník" pomocí D, pak výsledná formule je následující:

$P \rightarrow D$

Uvažujme nyní o výroku: "Pokud si neberu deštník, neprší." Je takové tvrzení ekvivalentní prvnímu? Platí-li první tvrzení, mohu ho brát jako postulát, kterému se každý musí podřídit. Zdá se tedy logické, že když si deštník neberu a dodržuji přitom postulát, pak nemůže pršet. Mnohem formálněji bychom tento vztah mohli dokázat právě pomocí pojmu tautologie. Druhá forma tvrzení se dá zapsat jako $\neg D \rightarrow \neg P$. Pak využijeme spojky ekvivalence, která interpretuje jako pravdivé dvě formule se stejnou pravdivostní hodnotou a tedy vlastně popisuje ekvivalentní chování dvou formulí z hlediska pravdivosti. Sestrojíme tedy takovou formuli a prokážeme, že je to tautologie pomocí interpretační tabulky. Implikace je nepravdivá pouze pro případ, kdy první formule je pravdivá a druhá nepravdivá - to je v souladu s naším chápáním podmínky. Pokud je splněn předpoklad podmínky, pak závěr musí platit - jinak by nebyla formule pravdivá. V případech kdy podmínka splněna není, může i nemusí závěr platit a v obou případech je to v pořádku. Jelikož sestrojená ekvivalence obou formulí je pravdivá ve všech interpretacích, můžeme konstatovat, že jde opravdu o tautologii.

P	D	$P \rightarrow D$	$\neg D$	$\neg P$	$\neg D \rightarrow \neg P$	$(P \rightarrow D) \leftrightarrow (\neg D \rightarrow \neg P)$
0	0	1	1	1	1	1
0	1	1	0	1	1	1
1	0	0	1	0	0	1
1	1	1	0	0	1	1

Nyní se již konečně dostáváme ke klíčovému pojmu tzv. **logického důsledku** množiny předpokladů. Máme-li množinu předpokladů a závěr, můžeme se ptát, zda tento závěr je logickým důsledkem (vyplývá z) předpokladů. To platí v případě, že pro každé ohodnocení výroků s interpretací pravda pro všechny formule z množiny předpokladů je pravdivá také formule reprezentující závěr. Jde tedy o chování podobné implikaci. Závěr nebude logicky vyplývat z předpokladů, jen pokud se najde takové ohodnocení výroků, kdy všechny předpoklady jsou pravdivé a závěr není pravdivý. Tato činnost, kdy odvozujeme závěry, se nazývá dedukce. Důležitým faktorem pak ještě zůstává tzv. konsistence předpokladů. Jde o to, že formulovaná vlastnost vyplývání degraduje na triviální situaci, pokud předpoklady nemají žádné ohodnocení, ve kterém by byly současně pravdivé. Takovéto množiny předpokladů jsou sporné (nekonsistentní) a z hlediska definice důsledku, je důsledkem této množiny jakákoliv formule. To samozřejmě v realitě není zcela odpovídající. Jednalo by se například o situaci, kdy určitý balík zákonů obsahuje dvě navzájem protichůdná nařízení a podle této množiny by pak jakékoliv jednání bylo v souladu s tímto zákonem.

Pokusme se nyní jednoduše pomocí tabulky prověřit, zda z množiny předpokladů z příkladu 1. vyplývá vina či nevina některého z podezřelých. Můžeme vyloučit podezřelého C, protože jeho nevina je přímo obsažena v předpokladech a tudíž musí vyplývat z množiny předpokladů a zároveň nemůže vyplývat jeho vina, pokud není množina předpokladů sporná. Postačí tedy sestrojit tabulku pro jednotlivé předpoklady a pro formule A,¬A,B,¬B ověříme, zda nesplňují vlastnost důsledku. V tabulce se vyskytuje také sloupec s hvězdičkami, který zvýrazňuje ohodnocení, kde jsou všechny předpoklady platné a tedy v těchto řádcích musíme kontrolovat interpretaci potenciálního důsledku. U závěrů pak vyznačujeme pomocí hvězdičky resp. lomítkem, zda skutečně splňuje resp. nesplňuje formule podmínku důsledku. Pokud ji splňují všechna zvýrazněná ohodnocení jde skutečně o vyvoditelný logický důsledek.

A	B	C	A ∨B ∨C	A → (B ∨C)	¬C		A		¬A		B		¬B	
0	0	0	0	1	1		0		1		0		1	
0	0	1	1	1	0		0		1		0		1	
0	1	0	1	1	1	*	0	/	1	*	1	*	0	/
0	1	1	1	1	0		0		1		1		0	
1	0	0	1	0	1		1		0		0		1	
1	0	1	1	1	0		1		0		0		1	
1	1	0	1	1	1	*	1	*	0	/	1	*	0	/
1	1	1	1	1	0		1		0		1		0	

Z tabulky je patrné, že jediný logický důsledek z prověřovaných závěrů je, že B je vinen. O podezřelém A nemůžeme konstatovat na základě předložených předpokladů ani jeho vinu ani nevinu. Vezmeme-li v úvahu možnost, že A je vinen, pak by musel být vinen i B. Pokud A vinen není, zároveň C vinen není

dle předpokladu a někdo vinen být musí, pak padá vina na B. B je tedy vinen v každém případě a o A nás ale nic neopravňuje to tvrdit.

Na uvedeném příkladu dedukce může ilustrovat hned několik klíčových otázek a problémů, které s logikou a naším článkem souvisí.
1. Pomocí přesně definovaných postupů jsme schopni dedukovat důsledky zcela automatizovaně a to zvládnou nejen lidé, ale zejména je to vhodné pro stroje (počítače). Už námi řešený příklad nemusí být pro každého člověka jednoduchý a složitost dedukce roste jednak s počtem výroků a jednak s počtem předpokladů. Už z těchto důvodů nejsou prostředky logiky zbytečné.
2. Problém automatizované dedukce lze řešit různě efektivní metodami. V tomto článku jsme se setkali prozatím jen s tabulkovou metodou, která je sice velmi triviální a dokonce si asi čtenář dokáže představit, že by takovýto algoritmus dokázal naprogramovat, nicméně její jednoduchost je také její slabinou. Už na příkladu 2 a 3 můžeme vidět, že rozsah tabulky roste exponenciálně vzhledem k počtu elementárních výroků. S ohledem na poznatky teorie vyčíslitelnosti a složitosti 3 víme, že exponenciální časová a prostorová složitost je pro klasické deterministické počítače prakticky nepoužitelná. Už pro 10 výrokových proměnných je možností ohodnocení přes 1000, pro 20 přes milion atd... Proto bylo a stále je vyvíjeno mnoho různorodých metod a celých formálních systémů, které jsou schopny automatizovat dedukci mnohem efektivněji.
3. Klasická výroková logika je jednou z nejjednodušších logik, což má své výhody i nevýhody. Výhodou je její transparentnost, pochopitelnost a především vlastnost rozhodnutelnosti 3. Rozhodnutelnost ve smyslu schopnosti o dané formuli říci jednoznačně, zda je splnitelná nebo ne na základě algoritmu, je důležitou vlastností pro automatizaci dedukce. Složitější logiky tuto vlastnost mít nemusejí. Nevýhodou výrokové logiky je naopak neschopnost rozlišit objekty a vztahy mezi nimi, nemožnost kvantifikovat vztahy, pojmout proměnlivost pravdivosti v čase a podobně. V neposlední řadě je u klasických logik nevýhodou jejich "černobílé vidění světa". Myslíme tím neschopnost zachytit jinou než absolutní pravdivost nebo nepravdivost. V reálném životě je mnoho situací, které nelze popsat jednoznačně. Například to zda je člověk spokojený, lze stěží popsat jen výrokem pravdivým nebo nepravdivým. Člověk může být spokojený v určitém stupni spokojenosti. Proto se v tomto článku chceme dotknout i tématu vícehodnotových logik, které jsou nyní velmi intenzivně zkoumány.

1.2 Predikátová logika

Ještě než se budeme blíže zabývat otázkou automatizace dedukce, chtěli bychom také stručně popsat logiku predikátovou. Predikátová logika je v podstatě zobecněním logiky výrokové a dává ji schopnost pracovat nejen s elementárními výroky, ale také rozlišit objekty a jejich vztahy. Na syntaktické úrovni se zavádí pojem termu a formule. Termem může být buď symbol zastupující objekt (konstanta) nebo funkci (funktor) nebo proměnná, za kterou lze dosadit libovolný term. Klíčovým pojmem je predikát, který jako

argumenty může mít termy a tím vlastně umožňuje vytvářet vazby mezi termy. Například bychom chtěli prostřednictvím predikátu zachytit vazby mezi rodiči a jejich dětmi. Zavedli bychom predikát s názvem dítě, který má dva argumenty - dítě a jeho rodiče. Samozřejmě pracujeme se symboly, takže skutečnými argumenty jsou pouze konstanty reprezentující objekty - konkrétní děti, rodiče atd. Konstanty jsou nejjednoduššími termy. Termy tedy nevyjadřují narozdíl od predikátů vazby, ale zastupují symbolicky objekty modelované reality. Takovými konstantami by mohla být například jména dětí a jejich rodičů. Tedy bychom mohli sestavit velmi jednoduchou formuli dítě(johana, lucie), která by měla pomocí predikátu vyjadřovat znalost, že mezi symbolem reprezentujícím objekty johana a lucie je vztah dítěte a rodiče. Termy mohou být však složitější a to funktory a proměnné. Funktory jsou symbolickými ekvivalenty pro nám dobře známé funkce. Funkce může mít několik argumentů (opět termů) a její interpretací je pak požadovaná hodnota. Například goniometrická funkce cos by pro argument - symbol 0 (intepretovaný číslem 0) - vracela číslo 1 (cos(0)=1). Proměnné pak jsou symbolickým prvkem, do kterých mohou být dosazeny jiné termy. Jelikož interpretace termů a predikátů zaleží (narozdíl od logických spojek) na uživateli predikátové logiky, mohl by si sémantiku uživatel uzpůsobit dle svých požadavků. Mohl by tedy vytvořit absurdní intepretace, kde by například funkce s názvem cos byla interpretována pro argument - číslo 0 - třeba číslem 333 nebo zcela nečíselným objektem. Chceme tím říci, že predikátová logika je velmi flexibilní a je potřeba mít stále na paměti rozdíl mezi syntaxí a sémantikou. Symboly mají své interpretace (sémantiku), kterou v případě logiky predikátové značně ovlivňuje ten, kdo ji používá. A i pod zcela totožnými symboly se tak může skrývat (nekonečně) mnoho různých významů.

Predikát může ze sémantického hlediska nabývat opět hodnotu pravda nebo nepravda. Jeho sémantickým operátorem je relace. Od úrovně predikátů výše se pak formule chovají jako ve výrokové logice a můžeme je tedy spojovat pomocí logických spojek. Jediným rozdílem je, že můžeme formule kvantifikovat a to buď univerzálním ($\forall x$) resp. existenčním ($\exists x$) kvantifikátorem. Ty pro zvolenou proměnnou pak zaručují, že kvantifikovaná formule bude pravdivá pro všechny resp. alespoň jeden objekt dosaditelný za proměnnou x.

Příklad 3.

Chtěli bychom pomocí predikátové logiky namodelovat situaci, kdy libovolné dítě je šťastné, pokud má otce i matku. Zavedli bychom si predikátové symboly pro vlastnosti objektů - stastny, muz a zena a dále pro vztah býti dítětem někoho - dite. Pomocí formule 1. pak můžeme vyjádřit, že pro každé dítě, ke kterému existuje objekt, jenž je ženou a zároveň dítě je dítětem tohoto objektu a zároveň platí totéž pro objekt typu muž, pak platí, že toto dítě je šťastné. Mnohem jednodušší je pak vyjádřit, které objekty jsou dítě, žena atd.. (např. johana je dítě lucie - formule 2. - 5.).

1. $\forall X[\exists Y[dite(X, Y) \wedge zena(Y)] \wedge \exists Y[dite(X, Y) \wedge muz(Y)] \rightarrow stastny(X)]$.
2. dite(johana, hashim).
3. dite(johana, lucie).
4. muz(hashim).
5. zena(lucie).

Můžeme pozorovat, že predikátová logika má mnohem vyšší expresivitu (vyjadřovací schopnost) než logika výroková. Zásadní je především možnost modelovat vazby mezi objekty pomocí predikátů. Silným prvkem je rovněž schopnost modelovat existenci. Formule 2.-5. z příkladu mohou připomínat řádky relační databáze. Relační databáze jsou založeny na prezentaci znalostí v relacích, které modelují rovněž vztahy mezi objekty. Skutečně bychom našli jistou analogii mezi tabulkou databáze (např. tabulka dětí) a jejich atributy (kdo je dítětem koho). Zkuste si představit databázovou tabulku studentů s atributy jméno, příjmení, bydliště, ročník. Taková tabulka relační databáze se dá vyjádřit predikátem student. Databáze by pak obsahovala třeba následující řádky (a jejich ekvivalentí vyjádření v predikátové logice):

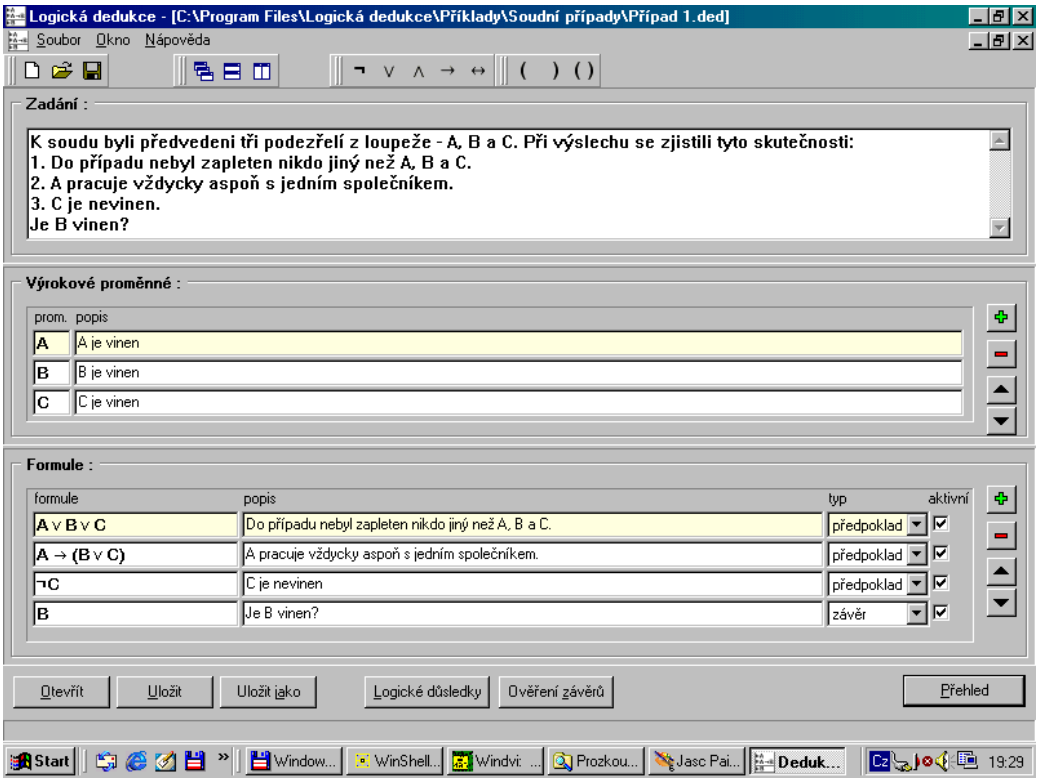
jan, novák, ostrava, 3 ~ formule: student(jan, novák, ostrava, 3)
jan, veselý, ostrava, 4 ~ formule: student(jan, veselý, ostrava, 4)
jiří, novák, ostrava, 1 ~ formule: student(jiří, novák, ostrava, 1)
...
petr, novotný, praha, 2 ~ formule: student(petr, novotný, praha, 2)

Klasické relační databáze jsou ale velmi úzkou podmnožinou způsobu reprezentace pomocí predikátové logiky. V databázích nemáme možnost používat proměnné a zejména logické spojky. Ty dokáží ušetřit mnoho prostoru - co by se muselo v databázi vyjádřit explicitně (třeba tisíci relacemi), lze elegantně zapsat jedním pravidlem a aplikovat dedukci. Samozřejmě, že již dávno začaly snahy o přibližování databázové technologie a logiky a to v tzv. deduktivních databázích (např. systém Datalog). Jde o inteligentní databáze, které kromě klasického explicitního vyjmenování vztahů umožňují také používání omezených logických prostředků a dedukce.

Jak už to ale bývá téměř všude v reálném životě, za výhody se platí určitými nevýhodami. Zatímco u výrokové logiky lze vždy rozhodnout (různě efektivně) o splnitelnosti dané formule, v predikátové logice je tento problém pouze částečně rozhodnutelný. Hlavním viníkem je potenciální možnost pracovat s nekonečnými doménami objektů (např. množina přirozených čísel - lze dosazovat za proměnné jakékoliv číslo). Potenciálně existuje možnost vytvořit také o něco složitější interpretační tabulku jako u logiky výrokové, ovšem takováto tabulka může být nekonečná. V případě univerzální kvantifikace formule musí tato formule platit pro všechny možné dosaditelné konstanty. Těch ale může být ve vztahu k nekonečným doménám také nekonečně mnoho a tudíž bych dostali tabulku s nekonečným počtem řádků. Proto existují snahy predikátové logice "něco vzít", tj. odebrat jí některé vyjadřovací schopnosti při zachování některých výhod tak, aby se stala rozhodnutelná a zároveň existovaly efektivní algoritmy pro dedukci.

1.3 Dedukce a její formalizace

Automatizace dedukce vyžaduje najít efektivní algoritmy pro generování důsledku nebo pro prověřování konsistence množin formulí. V praktických situací jde o automatizované zjišťování, zda z logických axiómů (vybrané tautologie) a speciálních axiomů (formalizované předpoklady) vyplývá závěr. V zásadě existují **metody sémantické a formální**. Tabulky z předchozí kapitoly jsou typickou sémantickou metodou. U sémantických metod musíme provádět interpretaci formulí, což je s ohledem na již zmiňované obrovské množství ohodnocení elementárních výroků velmi neefektivní přístup. I když některé sémantické metody vylepšují tuto nevýhodu, v zásadě je sémantický přístup pro automatizaci zcela nevhodný. Druhý formální (syntaktický) přístup se snaží se zcela oprostit od interpretace (smyslu) a používat prověřená pravidla pro práci se symboly (formulemi) bez ohledu na to, co znamenají. To je v principu velice efektivní, protože časová složitost pak nezávisí na možných interpretacích, ale na velikosti předpokladů jako symbolických formulí. Tyto metody vyžadují tedy jisté "know-how", je složitější je pochopit, ale v konečném důsledku jsou zásadně lepší. Lze je rovněž naprogramovat a pak již mohou sloužit v aplikacích na „inteligentní" usuzování. Právě pro složitost těchto metod (jsou dnes zcela v režii vysokoškolské výuky) je nebudeme všechny ani naznačovat. Existuje však velmi dobře použitelná a precizně zpracovaná práce autorky Libuše Pavliskové (diplomová práce na Ostravské Univerzitě).



Obrázek 1: Grafické rozhraní aplikace pro dedukci

Tato práce jednak obsahuje populární výklad problematiky dedukce a zejména je její součástí aplikace pro dedukci. Tato aplikace pracuje s výrokovou

logikou (tudíž je dostatečně jednoduchá i pro středoškolskou výuku) a zároveň umožňuje zapsat libovolnou množinu předpokladů a buď generovat všechny možné důsledky nebo o konkrétním závěru zjistit, zda je to důsledek. Možná ještě významnější je pro výuku na středních školách existence velkého množství připravených příkladů s atraktivním zadáním jako jsou soudní případy podobné tomu z kapitoly 1. a další (samozřejmě i mnohem složitější). Didaktický text, popis aplikace i samotnou aplikaci pro operační systém Windows lze získat na adrese:

http://www1.osu.cz/home/habibal/dedukce/

Formální metody dedukce lze dále v principu provádět dvěma způsoby - **přímo a nepřímo**. Zjednodušeně řečeno, při přímé metodě z předpokladů generujeme určitý důsledek pomocí odvozovacích (nebo jiných) pravidel. Při tomto způsobu tedy vždy hledáme potenciálně jiný důsledek podle toho, který závěr chceme dokázat. To v sobě samozřejmě skrývá jednu nevýhodu, jde především o možnost vygenerovat obrovské množství (potenciálně také nekonečné) různých důsledků a ten náš konkrétní se skrývá někde mezi nimi. To vede k neefektivitě a zejména se těžko hledají různé pomocné postupy, jak dedukci urychlit. Proto se používají spíše postupy nepřímé. Jsou podobné principu známého nepřímého důkazu. K předpokladům přidáme náš zkoumaný závěr, ovšem opačný (tedy se spojkou negace). Jelikož závěr vyplývá pokud je jeho interpretace pravda ve všech případech, kdy jsou pravdivé předpoklady, pak při opačném (negovaném) závěru taková množina nemůže být splnitelná. Při nepřímé metodě tedy s negovaným závěrem chceme dokázat nesplnitelnost. Tím se vyhneme základnímu problému přímé metody, protože náš cíl při dokazování je vždy stejný. Je jím prokázání nesplnitelnosti bez ohledu na dokazovaný závěr. To činí dedukci efektivnější. I přestože je stále obrovské množství možností, jak můžeme pomocí pravidel nakládat s předpoklady, můžeme již najít obecné postupy, jak urychlit (v určitých případech) proces hledání důkazu. Například v nepřímé rezoluční metodě (kterou si krátce vysvětlíme níže) jde o to, najít prázdnou formuli. Dá se tedy použít heuristika (metoda, která nefunguje zcela dokonale, ale v mnoha případech urychluje řešení), že je výhodnější pro následující krok hledání použít formule s co nejmenším počtem unikátních atomů (atomem se rozumí ve výrokové logice elementární výrok bez spojek). To samozřejmě není vždy pravda, ale intuitivně to je docela rozumné, pokud chceme dospět k formuli prázdné.

V tomto kurzu s omezeným rozsahem bychom se ještě chtěli zmínit o dvou formálních metodách. První trochu méně známá, ale i přesto velmi účinná je **metoda tablová**. Je založena na rozkladu formule ve stromu podle logických spojek a hledání větví, které obsahují navzájem negativní atomy (stejný atom s negací a bez negace). To ji činí velice účinnou, protože její časová náročnost je závislá jen na složitosti formule (počtu spojek). Bohužel v predikátové logice musí být obohacena o některé kroky, které její využití v praxi poněkud snižují. Druhou mnohem známější a široce používanou metodou je takzvaný **rezoluční princip** (rezoluce) resp. metody odvozené od něj. Myšlenku rezolučního principu formuloval v roce 1965 A. Robinson a od té doby byla velmi rozvinuta a zejména aplikována v různých systémech pro automatizované

usuzování. I když existuje samozřejmě i její varianta pro logiku predikátovou, omezíme se zde pro jednoduchost pouze na logiku výrokovou. V klasickém pojetí rezoluce funguje pouze na formulích převedených do formy tzv. klauzulí (existují i zobecnění pro libovolné formule, ale zatím nejsou používány ani příliš prozkoumány). Klauzule je taková formule, kde se vyskytuje pouze binární spojka disjunkce, která spojuje jednotlivé atomy s negací nebo bez negace. Každou formuli lze pomocí logických pravidel (ekvivalencí - viz např. příklad 2) převést na ekvivalentní množinu klauzulí (může jich být i velmi mnoho). Rezoluční pravidlo pak umožňuje generovat ze dvou klauzulí obsahujících stejný atom (symbol elementárního výroku), který je v jednom případě s negací a v druhém bez negace, novou klauzuli spojenou disjunkcí obou výchozích klauzulí a zároveň vypustíme onen pár atomů, na kterých jsme prováděli rezoluci. Schématicky to lze znázornit následovně (atomy a jejich negace lze v klauzuli libovolně přesunovat bez změny interpretace dané formule).

Rezoluční pravidlo

$$\frac{C_1 \vee x \qquad C_2 \vee \neg x}{C_1 \vee C_2}$$

C_1 a C_2 jsou zbývající části klauzulí a x je atom, na kterém rezoluci provádíme.

Máme-li pravidlo, jsme schopni z výchozích předpokladů (speciálních axiomů) a logických axiomů pomocí tohoto pravidla konstruovat sekvenci formulí, které říkáme důkaz. U každé syntaktické metody nebo formálního systému je důležité mít ověřeny dvě vlastnosti. Jde o to, aby metoda nebo systém, který obchází problematickou sémantiku tím, že pracuje pouze slepě" se symboly bez ohledu na jejich interpretaci, byl s touto sémantikou v souladu. První vlastnost, které se říká korektnost systému, zaručuje, že používaná pravidla generují pouze logické důsledky axiomů. Kdyby tomu tak nebylo, systém by z nekorektních pravidel generoval s předpoklady zcela nesouvisející formule (nesmyslné). Druhou vlastností je tzv. úplnost systému. Ta zaručuje, abychom pouze s danou množinou pravidel a axiomů byli schopni vygenerovat veškeré možné logické důsledky. Kdyby tomu tak nebylo, měli bychom sice korektní systém, který však neumí některé důsledky generovat/ověřovat, což je jako univerzální algoritmus opět nefunkční. Pokud je systém korektní a úplný, pak se chová zcela ekvivalentně sémantice a přesto ji nijak během dedukce nemusíme uvažovat.

Nyní rezoluci aplikujeme na již sémanticky řešený soudní případ.

Příklad 4.

Nejprve je nutné formule 1. $A \vee B \vee C$, 2. $A \rightarrow (B \vee C)$, 3. $\neg C$ převést do klauzulí. Formule 1. a 3. jsou klauzulemi bez převodu. Formule 2. vyžaduje převod. Využít můžeme pravidla pro přepis implikace na disjunkci. Toto pravidlo lze schématicky zapsat jako $A \rightarrow B \Leftrightarrow \neg A \vee B$ (čtenář si může lehce ověřit pomocí tabulky, že jde opravdu o formule s totožnou interpretací). Tímto pravidlem pak formuli 2. převedeme na formuli (disjunkce je navíc komutativní a asociativní): 2. $\neg A \vee B \vee C$. Nyní již můžeme buď přímo nebo nepřímo ověřovat závěry. Nejprve se pokusíme pomocí rezolučního pravdila (rezoluce) přímo vygenerovat, že B je vinen (B). Navíc přitom použijeme pomocná pravidla (například vyskytuje-li v klauzuli atom vícenásobně, je klauzule s jedním výskytem ekvivalentní; tyto kroky označíme pomocí $\Rightarrow$). Dalším platným pravidlem je, že formule $\perp \vee F$ je ekvivalentní s F. Toto využijeme v případě, že jedna z premis obsahuje pouze jeden atom a tím pádem je po rezoluci ekvivaletní s $\perp$.

1. $A \vee B \vee C$ (axiom),

2. $\neg A \vee B \vee C$ (axiom),

3. $\neg C$ (axiom),

4. (rezoluce na A v 1. a 2.): $(B \vee C) \vee (B \vee C) \Rightarrow B \vee C$,

5. (rezoluce na C v 4. a 3. - z formule 3. nezbylo nic): B

Tím jsme provedli důkaz toho, že B je vinen (jelikož systém založený na rezoluci je korektní a úplný).

Nyní se stejný závěr pokusíme dokázat nepřímo. Přitom přidáme k předpokladům negaci závěru a budeme se snažit prokázat nesplnitelnost (to znamená vygenerovat prázdnou klauzuli, která je nesplnitelná).

1. $A \vee B \vee C$ (axiom),

2. $\neg A \vee B \vee C$ (axiom),

3. $\neg C$ (axiom),

4. $\neg B$ (negace závěru),

5. (rezoluce na B v 4. a 2. - z formule 4. nezbylo nic): $\neg A \vee C$,

6. (rezoluce na B v 4. a 1. - z formule 4. nezbylo nic): A $\vee$C,

7. (rezoluce na A v 5. a 6.): C $\vee$C $\Rightarrow$ C,

8. (rezoluce na C v 7. a 3. - z formule 7. ani 3. nezbylo nic): $\perp$

Jelikož jsme dospěli k prázdné formuli, prokázali jsme nesplnitelnost množiny formulí 1. - 4., čímž je také prokázáno nepřímo, že B je logickým důsledkem množiny formulí 1. - 3.

Nepřímý důkaz v předchozím příkladě jsme samozřejmě mohli realizovat různými způsoby. Univerzálním přístupem je konstrukce nepřímého důkazu pomocí přímého přidáním jediného kroku, kdy provedeme rezoluci na vygenerovaný přímý důsledek a jeho negaci (pokud jde jen o atom). Na tom je vidět, že zřejmě existuje vždy mnoho způsobů, jak důkaz provést. Z hlediska časové složitosti jde principiálně o úlohu s exponenciální složitostí, o kterých jsme se již zmínili v předchozím článku v MFI. Nicméně existují přístupy, jak důkaz urychlovat a těm se říká rezoluční strategie. Některé pomáhají málo, ale jsou v principu úplné (tedy zachovávají úplnost systému) a některé jsou velmi efektivní za cenu neúplnosti (ale ve většině praktických úloh to nevadí). V některých formálních systémech je rezoluce nazývána jinak, resp. je její obecná myšlenka skryta v jiné symbolice. Například se můžete setkat s tzv. klauzulární logikou, kde se rezoluční pravidlo skrývá v tzv. pravidle řezu. Řez je výstižným pojmenováním, neboť jsme viděli, že rezoluce vlastně "vyřezává" atomy z původních formulí.

1.4 Aplikace formální dedukce v programování

V praxi se rezoluční metoda uplatňuje především v logickém programování. **Logické programování** není tak rozšířeno jako procedurální programování (např. algoritmizace v jazyce Pascal). Při procedurálním přístupu programátor musí vymyslet způsob, jak dosáhnout řešení cíle a to pomocí řízení výpočtu (musí správně použít proměnné, přiřazování, podmínky, cykly atd.). Logické programování vychází z myšlenky automatizace dedukce. Programátor nedefinuje postup řešení, ale pouze zadává formule (pravidla a fakta), která specifikují "logiku" řešení úlohy. Na takto zapsaný program se pak může dotazovat, podobně jako při prověřování závěrů dedukce a systém sám odpoví na dotaz. Způsob řešení je univerzální a programátor se o něj nemusí starat. Jako jednoduchý příklad může sloužit výpočet faktoriálů. V procedurálním programování musí programátor vytvořit řízený výpočet, tj. musí naprogramovat cyklus nebo rekurzivní proceduru. V logickém programování stačí naprogramovat dvě pravidla (resp. jedno pravidlo a jeden fakt). Pravidlo udává hodnotu faktoriálu pro argument předchozího přirozeného čísla pomocí

vazby na term s proměnnou (v logickém programování se nemá používat klasické přiřazování, měly by se používat výlučně prostředky logiky). Fakt udává hodnotu faktoriálu pro argument 0. V praxi by používání pouze logických prostředků způsobovalo neefektivní řešení, proto implementace logického programování často umožňují použití také některých omezených procedurálních prvků (např. řez). Asi nejznámějším prostředkem logického programování je jazyk PROLOG (PROgramming in LOGic). Vzhledem k omezenému rozsahu tohoto článku jej nebudeme rozebírat, ale čtenář má možnost využít elektronický učební text s mnohými příklady 2. Pro vlastní pokusy doporučujeme získat z Internetu některou z freewarových implementací. Logické programování je výhodné především u úloh, kdy hledáme řešení v rozsáhlém stavovém prostoru a v úlohách typických v umělé inteligenci.

Jak již bylo napsáno v předchozím odstavci je Prolog založen především na matematické logice. Jak již víte z kurzů vztahujících se k tomuto oboru teoretické informatiky, je způsob formulace znalostí pomocí logiky stavěn na symbolické reprezentaci pomocí formulí. Setkali jste se s logikou výrokovou a predikátovou. První z nich je poměrně jednoduchá a umožňuje formulovat platné věty pomocí výroků (tvrzení o modelované realitě, která mohou pravdivá nebo nepravdivá) a logických spojek (umožňují spojovat tvrzení do složitějších formulí např. ve tvaru implikace – podmínky). Právě tyto podmínky – implikace – jsou základním stavebním kamenem programů v Prologu. Pomocí nich můžeme formulovat bázi znalostí. V Prologu je ovšem nutné respektovat jistá omezení, aby bylo možné programy efektivně vyhodnocovat.
Jistě si pamatujete na převody do normálních forem. Jejich existence umožňovala jednodušší dokazování splnitelnosti a platnosti formulí např. pomocí rezolučního pravidla. Vzniklé klauzule byly sice méně přehledné než původní zápis v obecné výrokové logice, avšak práce s nimi byla při určitých operacích jednodušší a efektivnější. Právě těchto principů bylo využito při konstrukci mechanismů Prologu. Omezení je však pro vyšší efektivitu ještě větší. V Prologu je nutné znalosti formulovat pomocí tzv. Hornových klauzulí.

Hornovy klauzule

Hornova klauzule je konečná množina literálů spojených spojkou disjunkce, kde nejvýše jeden literál je pozitivní (není negovaný):

$$\neg \mathbf{p}_1 \vee \neg \mathbf{p}_2 \vee ... \vee \neg \mathbf{p}_n \vee \mathbf{q}.$$

Tato klauzule má v implikativním zápise tvar:

$$(\mathbf{p}_1 \,\&\, \mathbf{p}_2 \,\&...\&\, \mathbf{p}_n) \rightarrow \mathbf{q}$$

Intuitivní význam takového zápisu je jednoduchý. Jde o platnost výroku **q** podmíněnou současnou platností výroků $\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n$.
Jinak řečeno logické programování spočívá ve formulaci množiny podmínek typu: Pokud platí $\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n$, pak platí také **q.** Pomocí těchto podmínek lze formulovat chování celé modelované báze znalostí. V případě, že máme formuli, která neobsahuje žádné $\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_n$, jde o nepodmíněný fakt.

Příklad 5.

Vezměme v úvahu následující zápis tvrzení v přirozeném jazyce.

Student složil úspěšně přijímací zkoušku, pokud udělal test z matematiky a zároveň i z informatiky.

V jazyce výrokové logiky bychom tuto bázi znalostí zapsali například následovně.

Výroky:

z – student složil úspěšně přijímací zkoušku,
m – student udělal zkoušku z matematiky,
i – student udělal zkoušku z informatiky,

$(m \,\&\, i) \rightarrow z$

V Prologu se podobná formule dá vyjádřit poněkud jiným zápisem.

z :- m, i.

Vidíte, že způsob zápisu je „opačný“. Tedy nejprve vyjádříte konsekvent (tedy co platí) a teprve za symbolem :- podmínky, které musí být splněny.

Pokud by Prolog měl pouze tyto možnosti, byl by sice schopen řešit velmi malou skupinu problémů, ale jeho pravé využití mu dává teprve obohacení o prvky predikátové logiky. Jistě si vzpomínáte, že predikátová logika je už mnohem složitější než výroková. Umožňuje používat také predikáty, funktory a proměnné (zjednodušeně by se dalo říct, že umožňuje formulovat vztahy a vlastnosti objektů pomocí relací). Díky této vlastnosti, už nejsme odkázáni na prosté výroky, ale můžeme dosazovat proměnné a jejich objekty. Další důležitou charakteristikou predikátové logiky jsou kvantifikátory. Ty je při logickém programování bohužel nutno obětovat opět kvůli efektivitě programování v Prologu. Všechny proměnné jsou zde chápány jako univerzální.
Podívejme se nyní na podobný problém jako v příkladu 1, avšak nyní již mnohem bohatěji popsaný pomocí predikátové logiky.

Příklad 6.

Vezměme v úvahu následující zápis tvrzení v přirozeném jazyce.

Student složil úspěšně přijímací zkoušku, pokud udělal test z matematiky nejméně na 200 bodů a zároveň i z informatiky nejméně na 100 bodů.

V jazyce predikátové logiky bychom tuto bázi znalostí zapsali například následovně.

Predikáty:

zk(<student>) – student složil úspěšně přijímací zkoušku,
mat(<student>,<počet bodů>) – student složil zkoušku z matematiky na počet bodů,
inf(<student>,<počet bodů>) – student složil zkoušku z informatiky na počet bodů

$\forall X ((mat(X, M) \& inf(X, I) \& M \geq 200 \& I \geq 100) \rightarrow zk(X))$

V Prologu by se tato formule dala zapsat následovně.

```
zk(X) :- mat(X, M), inf(X, I), M >= 200, I >= 100.
```

Proměnné jsou od funktorů a predikátových symbolů rozlišeny pomocí velkého písmena na začátku. Vidíte, že v Prologu se rovněž mohou používat relační operátory. Prolog obsahuje ještě mnoho dalších pomocných operátorů a predikátů, se kterými se seznámíte v tomto kurzu. Umíme-li pomocí Prologu sestavit složitý systém podmínek a faktů, pak ještě stále chybí něco, co ho činí použitelným pro praxi. Samotná báze znalostí by nám jako taková příliš neposloužila, pokud bychom z ní nemohli nějakým způsobem dedukovat nové neznámé znalosti. Opět si vzpomeňte na výuku logiky. Máme-li slovní úlohu na dedukci, potřebujeme kromě zápisu předpokladů (tedy báze znalostí), také nějaký závěr, který chceme ověřit. Ten se v Prologu nazývá dotaz (u některých implementací bývá uvozen řetězcem "?-". A tou nejdůležitější věcí, kterou potřebujeme je mechanismus (metoda), kterou dotaz můžeme potvrdit. Znáte jistě celou řadu metod z výrokové i predikátové logiky. Např. tabulkovou metodu, sémantické tablo nebo rezoluci. Jak jste již poznali, rezoluce je zvlášť účinnou metodou, pokud se omezíme pouze na klauzule. Pokud se navíc omezujeme na Hornovy klauzule, je dokazování ještě efektivnější. Provádí se tak formální odvozování s pomocí pravidla, které je obdobou pravidla řezu v klauzulární logice. Jelikož však je vždy v klauzuli nejvíce jeden literál bez negace má pravidlo jednodušší tvar a máme možnost jednoduše nahradit jeho výskyt všemi podmínkami pro pravidlo se stejným literálem jako má tento negovaný literál. Odvozování je prováděno nepřímou metodou – tedy dotaz se považuje za negovaný a jádro Prologu se snaží dospět k prázdné klauzuli (množině literálů). Podívejme se na příklad.

Příklad 7.

Mějme pravidlo z příkladu 2, které nám říká, za jakých podmínek složí student přijímací zkoušku.

```
zk(X) :- mat(X, M), inf(X, I), M >= 200, I >= 100.
```

Přidejme k nim následující fakta, která o studentovi Jiřím říkají, že složil zkoušku matematiky na 203 bodů a z informatiky na 152 bodů.

mat(jiří, 203). inf(jiří, 152).

Položme nyní v Prologu dotaz (zkusme prověřit), zda Jiří složil zkoušku.

Dotaz: zk(jiří).

Prolog, pak provede díky svému vnitřnímu mechanismu následující nepřímou důkazovou sekvenci.

Krok 1: Dotaz se považuje za negovaný a tedy je možné provést s pravidlem řez, pokud provedeme substituci jiří za X.

mat(jiří, M), inf(jiří, I), M >= 200, I >= 100.

Krok 2: Na výslednou formuli je možné opět aplikovat pravidlo řezu s faktem mat(jiří, 203), pokud provedeme substituci 203 za M.

inf(jiří, I), 203 >= 200, I >= 100.

Krok 3: Výsledek je možné dále zpracovat s faktem inf(jiří, 152), pokud substituujeme 152 za I.

203 >= 200, 152 >= 100.

Krok 4: Interně se zpracuje výsledek porovnání 203 >= 200 s výsledkem true.

152 >= 100.

Krok 5: Zpracuje se výsledek 152 >= 100 s výsledkem true a dostáváme prázdnou klauzuli.

Zkuste aplikovat Prologovskou inferenci na vlastní příklad formální dedukce.

- Logika jako prostředek pro modelování lidského úsudku
- Formální vs. Sémantická dedukce
- Logické programování a dedukce

2 Vícehodnotová logika

V této kapitole se dozvíte:

- Vícehodnotové logiky
- Fuzzy matematika a logika
- Dedukce ve fuzzy logice

Po jejím prostudování byste měli být schopni:

- Pracovat a modelovat realitu pomocí fuzzy logiky.
- Provádět modelování reality pomocí fuzzy logiky.
- Zobecnit znalosti z dvouhodnotové logiky na vícehodnotové logiky

Klíčová slova této kapitoly:

Symbolická logika, logická dedukce, fuzzy logika

Doba potřebná ke studiu: 8 hodin

Průvodce studiem

Studium této kapitoly je poměrně náročné zejména pro ty z Vás, kteří dosud nemají žádné znalosti z fuzzy logiky. V takovém případě Vám zřejmě některé příklady budou připadat obtížně pochopitelné, ovšem nenechte se tím odradit, neboť pochopením této části se Vám usnadní studium následujících kapitol.
Na studium této části si vyhraďte alespoň 8 hodin. Doporučujeme studovat s přestávkami vždy po pochopení jednotlivých podkapitol.

2.1 Klasická a fuzzy matematika

Fuzzy matematika a logika

Pro modelování situací v reálném světě je tradiční logika poměrně chudá. Výrazné obohacení škály odstínů pravdivosti tvrzení reprezentovatelných formálními prostředky představuje vedle pravděpodobnostní logiky fuzzy logika, obecněji fuzzy matematika.
Přes všechny její přednosti je zásadním problémem tradiční logiky především dvouhodnotová logická interpretace. Již dlouhou dobu existují formalismy zavádějící například logiku trojhodnotovou, kde třetí logická hodnota kromě pravda/nepravda je "nevím". Různé pokusy o zobecnění například pomocí pravděpodobnosti byly zastíněny v šedesátých letech 20. století tzv. fuzzy matematikou. Fuzzy matematika vychází z principu fuzzy množiny, která narozdíl od klasické množiny, která buď obsahuje nebo neobsahuje prvek, může prvek obsahovat na určitém stupni příslušnosti. Prvky tedy mohou být v množině buď zcela nebo vůbec, ale také "jen trochu". Fuzzy logika pak tento

princip využívá tím, že rovněž interpretace formule nemusí být jen pravda nebo nepravda, ale může to být pravda v určitém stupni.

I když myšlenka fuzzy logiky je velmi prostá, lehce pochopitelná a tudíž implementovatelná do různých automatizovaných systémů, je potřeba se při jejím používání držet některých vlastností. I v běžném životě je možné setkat se s aplikacemi fuzzy logiky. Například elektrospotřebiče - pračky - mají dnes často na sobě nápis "Fuzzy-logic". Tím se může myslet například schopnost dávkovat automaticky prací prostředky nikoliv v přesně vymezených mantinelech podle váhy prádla (např. pro 1kg prádla přidej přesně 10g pracího prostředku), ale pouze vágními pravidly (např. je-li prádla málo, přidej pracího prostředku málo). Termínem fuzzy logika se označuje schopnost pracovat s neurčitou (vágní) informací, ale samozřejmě pojem fuzzy logika ve smyslu matematicko-informatickém je mnohem složitější. Základním problémem často bývá živelné používání množin stupňů příslušnosti bez ohledu na to, že musí existovat přímá souvislost také s používanými logickými spojkami. Proto je nezbytné, aby množina stupňů příslušnosti (pravdivosti) formule byla některou ze zavedených algeber. Množinou bývá v praktických aplikacích většinou interval reálných resp. racionálních čísel [0, 1], i když teorie fuzzy množin a fuzzy logika má teoretické výsledky i pro mnohem složitější struktury. Tyto algebry mají vždy své výhody a nevýhody podle toho, jaké interpretace spojek na intervalu [0, 1] jsou použity. Tyto spojky musí být navíc ve vzájemné souvislosti tj. použití určité konjunkce předurčuje též použití ostatních spojek. Nevýhodou algeber pak může být například nespojitost interpretačních funkcí spojek, neplatnost některých zásadních logických pravidel (zákonů) tak, jak jsou známy z klasické logiky. Tato kapitola se pokusí pouze naznačit chování těchto algeber, exaktní definice budou uvedeny později.

Pravděpodobně nejlepší kompromis představuje tzv. Lukasiewiczova algebra pojmenovaná po významném polském logikovi. Tato algebra je následující struktura:

Struktury pravdivostních hodnot

$$L_L = \langle [0,1], \wedge, \vee, \otimes, \rightarrow, 0, 1 \rangle$$

kde $\wedge, \vee$ jsou klasickými operacemi infima, resp. suprema, [0,1] je interval reálných hodnot mezi 0 a 1. Základní operace v této algebře jsou definovány následovně:

$$a \otimes b = 0 \vee (a + b - 1) \qquad a \rightarrow b = 1 \wedge (1 - a + b) \qquad a \oplus b = 1 \wedge (a + b)$$

$$\neg a = 1 - a$$

Operace $\otimes$ umožňuje definovat tzv. Lukasiewiczovu konjunkci a operace $\oplus$ Lukasiewiczovu disjunkci. Operace tzv. residua $\rightarrow$ pak definuje sémanticky operaci implikace.

Operace tzv. birezidua $\leftrightarrow$ může být definována jako $a \leftrightarrow b =_{df} (a \rightarrow b) \wedge (b \rightarrow a)$ a díky ní lze popsat chování ekvivalence.

Takováto struktura pak umožňuje definovat sémantiku (chování) logických spojek. V klasické dvouhodnotové logice struktura pravdivostních hodnot obsahuje pouze dva prvky nepravda – 0 a pravda – 1. Na této dvouprvkové struktuře je možno například definovat sémantiku (interpretaci) spojky konjunkce A & B pro formule A a B jednoduše tak, že I(A & B) = 1, právě tehdy když I(A) = 1 a zároveň I(B) = 1. Pokud je struktura {0, 1} považována za množinu s minimálním prvkem 0 a maximálním prvkem 1, pak je možno

definovat interpretaci konjunkce také jako $I(A \& B) = I(A) \wedge I(B)$, kde $\wedge$ je operací infima (resp. minima).
Podobným způsobem pak lze definovat interpretaci Lukasiewiczovy konjunkce.
Ve fuzzy logice při použití Lukasiewiczovy algebry jako struktury pravdivostních pak lze definovat interpretaci Lukasiewiczovy konjunkce jako:
$I(A \& B) = 0 \vee (I(A) + I(B) - 1)$

Příklad 8.

Jde o situaci v rodině, o níž se předpokládá, že je tak šťastná, jak je šťastný otec a matka. V klasické dvouhodnotové logice lze pak pomocí výroku O popsat štěstí otce a pomocí M štěstí matky. Štěstí rodiny je pak popsáno formulí O & M. V klasické logice je bohužel potřeba popsat „šťastnost“, buď jako absolutně platnou nebo absolutně neplatnou. To samozřejmě neodpovídá příliš modelované realitě – nikdo asi není absolutně šťastný ani nešťastný.
Při použití složitější struktury pravdivostních hodnot, např. Lukasiewiczovy algebry, lze pak přirozeněji štěstí definovat tzv. stupněm příslušnosti – to je jedna z hodnot, které poskytuje nosič algebry. Existuje tu samozřejmě možnost popsat je jako v klasické logice, použije-li se buď minimální nebo maximální prvek intervalu [0, 1] (absolutní pravda, absolutní nepravda). Mnohem zajímavější je pak použití třeba poloviční pravdivosti – tedy výrok O se interpretuje hodnotou 0,5 (otec je šťastný asi napůl), výrok M hodnotou 0,9 (matka je šťastná téměř úplně). Interpretace štěstí rodiny popsané formulí O & M je pak:

$I(O \& M) = 0 \vee (I(O) + I(M) - 1) = 0 \vee (0{,}5 + 0{,}9 - 1) = 0{,}4$ (tudíž rodina je šťastná o něco méně než napůl)

Pomocí fuzzy logik je možno lehce vyřešit paradoxy, se kterými si klasická logika neumí poradit. Vyskytují se různé formulace, ale v zásadě jsou všechny totožné.

Známý paradox hromady. Jde o to, v klasické dvouhodnotové logice namodelovat situaci hromady, ke které jsou přidávány další kameny.
Je zřejmé, že hromada bez kamenů je rozhodně malá. Dále je rozumný předpoklad, pokud máme malou hromadu kamenů, pak hromada s přidaným jedním kamenem bude stále malá. V klasické logice, kde jsou formule pravdivé nebo nepravdivé, by pak každá hromada byla paradoxně malá. Můžeme totiž potenciálně nekonečnou sekvencí implikací vždy dokázat, že hromada s počtem kamenů o x větším je stále malá. Ve fuzzy logice můžeme díky pravdivosti s určitým stupněm příslušnosti namodelovat tuto situaci dvěmi formulemi (zde se použije predikát malahromada(t), kde t je počet kamenů v hromadě):

1. $\text{malahromada}(x) \rightarrow \text{malahromada}(x + 1)$ - pravdivá ve stupni 0.999
2. malahromada(0) - pravdivá ve stupni 1

Formule 1. není narozdíl od klasické logiky pravdivá zcela a díky tomu nedojde k paradoxní dedukci. Díky omezené pravdivosti bude při dedukci s každou aplikací formule 1. při navyšení počtu kamenů o 1 klesat pravdivost vyvozeného predikátu malahromada o 0.001. Například jde o to, ověřit stupeň pravdivosti důsledku malahromada(1) z předpokladů 1. a 2. Platí-li formule 1. na 0.999 a formule 2. na 1, pak lze na základě definice operátoru $\rightarrow$ interpretace (I) implikace a platného vztahu:

$$I(malahromada(0)) = 1$$

sestavit rovnici:

$0.999 = 1 \wedge (1 - 1 + I(malahromada(1))) \Rightarrow I(malahromada(1)) = 0.999$

Pro malahromada(2):

$0.999 = 1 \wedge (1 - 0.999 + I(malahromada(2))) = 0.001 + I(malahromada(2)) \Rightarrow I(malahromada(2)) = 0.998$

A tak dále pro zvětšující se počet kamenů, až pro malahromada(1000) je možno dojít ke stupni pravdivosti 0.

Zajímavé jsou také aplikace teorie fuzzy množin a fuzzy logiky, například existuje přístup s využitím tzv. lingvistických proměnných. Tyto proměnné mohou místo klasických číselných hodnot nabývat hodnot blízkých přirozenému jazyku jako jsou například lingvistické výrazy typu: "velmi malý", "zhruba střední" atd. Pomocí těchto proměnných pak lze modelovat velmi srozumitelně a podobně jako člověk reálné situace.

Například lze popisovat řízení auta, kde dvě z pravidel by mohla znít:

"KDYŽ na semaforu svítí jen žluté světlo A ZÁROVEŇ vzdálenost od křižovatky je **malá** A ZÁROVEŇ rychlost auta je **malá** PAK sešlápni brzdu **střední** silou."

"KDYŽ na semaforu svítí jen žluté světlo A ZÁROVEŇ vzdálenost od křižovatky je **velmi malá** A ZÁROVEŇ rychlost auta je **střední** PAK sešlápni plyn **velkou** silou."

Tato pravidla mohou pak být interpretována pomocí fuzzy logiky a lze díky nim velmi lehce modelovat a zejména automatizovat postupy z mnoha oblastí života. Existují systémy, které se v praxi skutečně používaly a používají jako je systém LFLC (Linguistic Fuzzy Logic Controller) vyvinutý na Ostravské Univerzitě. Pomocí něj se modelovaly například technologické procesy v hutích a oproti klasickým prostředkům jsou velkým zlepšením. Lze totiž na rozdíl od klasických regulačních technik, které vyžadují složitou matematiku jako jsou diferenciální rovnice a se kterými může pracovat jen velmi úzká skupina odborníků, tyto systémy svěřit i poučenému laikovi. Ten dobře zná například svůj technologický proces, který ručně obsluhoval dlouhou dobu a je schopen formulovat slovně své akční zásahy. Ty pak stačí naformulovat a odzkoušet a máme ve velmi krátkém čase funkční automatizaci daného procesu, založenou na fuzzy logice.

Teorie Fuzzy množin je prostředek, který umožňuje matematicky popsat vágní pojmy a pracovat s nimi. Pojem Fuzzy je možno přeložit jako nepřesně

ohraničené, neostré, matné, mlhavé, neurčité, vágní… Základním pojmem této teorie je pojem fuzzy množiny.

Myšlenka je přibližně následující: Není-li možno stanovit přesné hranice třídy určené vágním pojmem, je vhodné nahradit rozhodnutí o náležení či nenáležení daného prvku do ní mírou vybíranou z nějaké škály. Každý prvek bude mít přiřazenu míru vyjadřující jeho místo a roli v této třídě. Bude-li škála uspořádaná, pak menší míra bude vyjadřovat, že daný prvek je někde blíže k okraji třídy. Tato míra se nazývá *stupeň příslušnosti* prvku do dané třídy a třída, v níž každý prvek je charakterizován stupněm příslušnosti do ní, se nazývá *fuzzy množina*. Lze také říci, že stupeň příslušnosti vyjadřuje stupeň přesvědčení, že daný prvek patří do dané fuzzy množiny.

Mlhavě definované třídy objektů mají v myšlení člověka velmi důležitou roli. Většina souborů v reálném světě se vyznačuje absencí ostrých hranic které by jednoznačně určovaly zda prvek do množiny patří či nikoli. Například při popisu vágního pojmu „nízká teplota“ lze pak každé teplotě, která připadá v úvahu (jsme omezeni tzv. absolutní nulou, což odpovídá hodnotě –273,15 °C) přiřadit číslo vyjadřující stupeň přesvědčení o tom že taková teplota je nízká. Tento stupeň vyplývá z toho jak je pojímán pojem „nízká teplota“. Je vidět že přiřazování stupně příslušnosti je subjektivní a závisí také na kontextu (samotný člověk žijící v rovníkové Africe má o nízké teplotě jistě jiné představy něž obyvatel Grónska).

Stupeň příslušnosti však nemá nic společného s pravděpodobností. U pravděpodobnosti jde o nejistotu, zda nastane či nenastane určitý jev – nastane-li, lze ho jednoznačně přiřadit do přesně popsaného souboru (množiny s ostrými hranicemi). Naproti tomu stupeň příslušnosti souvisí s nejistotou spojenou s příslušností prvku do souboru s neostrými hranicemi (fuzzy množiny).

2.2 Struktury pravdivostních hodnot

Definice fuzzy množiny:

Definovat fuzzy množinu je možno podle [14] takto:

Pro danou množinu **U** ={x}(universum) je fuzzy množina A v **U** charakterizována funkcí příslušnosti A(x), která přiřazuje každému prvku x∈ **U** reálné číslo z intervalu <0, 1> představující stupeň příslušnosti x k A. Nabývá-li A(x) pouze hodnot 0 a 1, pak je A klasickou množinou. Funkce příslušnosti je tedy zobrazením množiny **U** do intervalu <0, 1>,

$$A(x): \mathbf{U} \rightarrow <0, 1>, \ \forall x \in U ,$$

neboť každému prvku x∈ **U** přiřazuje právě jeden prvek (reálné číslo) z intervalu <0, 1>.

Je-li univerzum **U** spočetnou množinou $\{x_1, x_2, \dots, x_n\}$, popisuje se fuzzy množina A v **U** buď množinou párů $\{(A(x_i),x_i)\}$, tedy

$$A = \{(A(x_1),x_1), (A(x_2),x_2), \dots , (A(x_n),x_n)\}$$

nebo zápisem

$$A = A(x_1)/x_1 + A(x_2)/x_2 + \ldots + A(x_n)/x_n = \sum_{x_i \in U} A(x_i)/x_i \ ,$$

Kde symboly „+“ a „Σ“ mají význam sjednocení (nikoliv součtů), symbol „/“ odděluje stupně příslušnosti od korespondujících prvků. Funkce příslušnosti je diskrétní.

Je-li univerzum U nespočetnou množinou (např. množinou reálných čísel), popisuje se fuzzy množina A v U jako

$$A = \int_{x \in U} A(x) / x \ ,$$

Kde symbol „∫“ představuje sjednocení nespočetně mnoha prvků (nikoliv integrál) a funkce příslušnosti je spojitá.

Pro každý prvek *x* z univerza platí:

$A(x) = 1$, pak *x* jistě patři do *A*,

$A(x) = 0$, pak *x* jistě nepatři do *A*,

$0 < A(x) < 1$, nejsme si jisti, zda *x* patři do *A*.

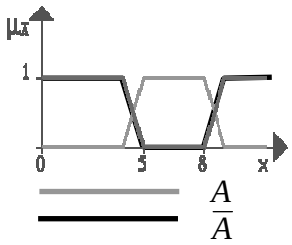
Základní operace

S fuzzy množinami lze, podobně jako s klasickými množinami, definovat základní operace sjednocení, průniku a doplňku. Kromě nich však lze definovat ještě řadu dalších operací, které v klasické teorii množin buď nemají smysl, nebo dávají výsledek ekvivalentní s některou ze základních operací.

a) Fuzzy doplněk

Doplněk A fuzzy množiny A je důležitou operací a je definován pomocí vztahu

$$\overline{A}(x) = 1 - A(x)$$



Fuzzy negace - je unární operace ¬ : (0, 1) → (0, 1) taková, že

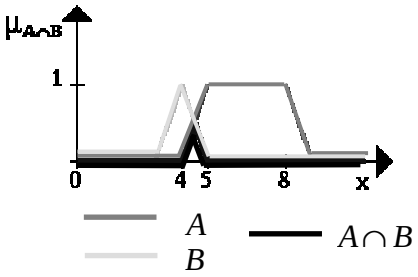
$$\alpha \le \beta \Rightarrow \neg\beta \le \neg\alpha$$

$$\neg\neg\alpha = \alpha$$

b) Fuzzy průnik

Průnik dvou fuzzy množin *A* a *B* je fuzzy množina *C*, která má funkci příslušnosti

$C = A \cap B$, právě když $C(x) = A(x) \wedge B(x)$.



Obrázek 2:

Obrázek 3: **obr. 2.1**

Fuzzy konjunkce - je binární operace $\wedge : \langle 0,1 \rangle^2 \to \langle 0,1 \rangle$ splňující následující axiomy pro všechna $\alpha, \beta, \gamma \in \langle 0,1 \rangle$:

$$\alpha \wedge \beta = \beta \wedge \alpha \text{ (komutativita)}$$

$$\alpha \wedge (\beta \wedge \gamma) = (\alpha \wedge \beta) \wedge \lambda \text{ (asociativita)}$$

$$\beta \le \gamma \Rightarrow \alpha \wedge \beta \le \alpha \wedge \gamma \text{ (monotomie)}$$

$$\alpha \wedge 1 = \alpha \text{ (okrajová podmínka)}$$

Příklady fuzzy konjunkcí:

- Standardní (min, Gödelova, Zadehova, …):

$$\alpha \underset{S}{\wedge} \beta = \min(\alpha, \beta).$$

- Součinová (produktová, pravděpodobnostní, Goguenova, angl. algebraic produkt):

$$\alpha \underset{P}{\wedge} \beta = \alpha \cdot \beta.$$

- Lukasiewiczova (Gilesova, angl. bold):

$$\alpha \underset{L}{\wedge} \beta = \begin{cases} \alpha + \beta - 1 & \text{pro } \alpha + \beta - 1 > 0, \\ 0 & \text{jinak.} \end{cases}$$

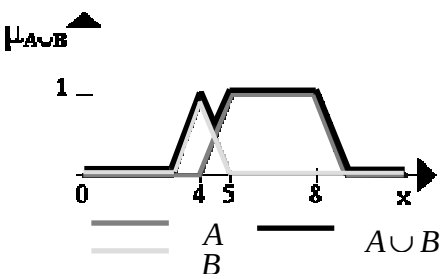
- Drastická (slabá, angl. weak):

$$\alpha \underset{D}{\wedge} \beta = \begin{cases} \alpha & \text{pro } \beta = 1, \\ \beta & \text{pro } \alpha = 1, \\ 0 & \text{jinak.} \end{cases}$$

c) Fuzzy sjednocení

je operace na fuzzy množinách definovaná pomocí fuzzy disjunkce:

$C = A \cup B,$ právě když $C(x) = A(x) \vee B(x).$



Fuzzy disjunkce - je binární operace $\vee : \langle 0,1 \rangle^2 \to \langle 0,1 \rangle$ splňující

$$\alpha \vee \beta = \beta \vee \alpha \text{ (komutativita)}$$

$$\alpha \vee (\beta \vee \gamma) = (\alpha \vee \beta) \vee \lambda \text{ (asociativita)}$$

$$\beta \leq \gamma \Rightarrow \alpha \vee \beta \leq \alpha \vee \gamma \text{ (monotomie)}$$

$$\alpha \vee 1 = 1 \text{ (okrajová podmínka)}$$

Příklady fuzzy disjunkcí:

- Standardní (max, Gödelova, Zadehova, …):

$$\alpha \overset{S}{\vee} \beta = \max(\alpha, \beta).$$

- Součinová (produktová, pravděpodobnostní, …):

$$\alpha \overset{P}{\vee} \beta = \alpha + \beta - \alpha \cdot \beta.$$

- Lukasiewiczova (Gilesova, angl. bold, bounded sum):

$$\alpha \overset{L}{\vee} \beta = \begin{cases} \alpha + \beta & \text{pro } \alpha + \beta < 1, \\ 1 & \text{jinak.} \end{cases}$$

- Drastická (slabá, angl. weak…):

$$\alpha \overset{D}{\vee} \beta = \begin{cases} \alpha & \text{pro } \beta = 0, \\ \beta & \text{pro } \alpha = 0, \\ 1 & \text{jinak.} \end{cases}$$

- Einsteinova:

$$\alpha \overset{E}{\vee} \beta = \frac{\alpha + \beta}{1 + \alpha\beta}$$

d) R-implikace

Speciální operací fuzzy množin a je to operace:

$$\alpha \overset{R}{\to} \beta = \sup\{\gamma : \alpha \wedge \gamma \leq \beta\}$$

kde $\wedge$ je fuzzy konjunkce (je-li spojitá, lze supremum nahradit maximem)

Fuzzy implikace - je jakákoli operace $\to : \langle 0,1 \rangle^2 \to \langle 0,1 \rangle$, která se na $\{0,1\}^2$ shoduje s klasickou implikací. Mějme tedy:

$$\alpha \to \beta = 1 \Leftarrow \alpha \leq \beta,$$

$$\alpha \to \beta = 1 \Rightarrow \alpha \leq \beta,$$

$$1 \to \beta = \beta,$$

$\to$ je nerostoucí v 1. argumentu a neklesající v 2. argumentu,

$$\alpha \to \beta = \underset{S}{\neg}\beta \to \underset{S}{\neg}\alpha,$$

$$\alpha \to (\beta \to \gamma) = \beta \to (\alpha \to \gamma),$$

spojitost.

Příklady R-implikací:

- Od standardní konjunkce $\underset{S}{\wedge}$ je odvozena Gödelova implikace

$$\alpha \overset{R}{\underset{S}{\to}} \beta = \begin{cases} 1 & \text{pro } \alpha \le \beta, \\ \beta & \text{jinak.} \end{cases}$$

Je po částech lineární a spojitá s výjimkou bodů $(\alpha, \alpha), \alpha < 1$.

- Od Lukasiewiczovy konjunkce $\underset{L}{\wedge}$ je odvozena Lukasiewiczova implikace

$$\alpha \overset{R}{\underset{L}{\to}} \beta = \begin{cases} 1 & \text{pro } \alpha \le \beta, \\ 1 - \alpha + \beta & \text{jinak.} \end{cases}$$

Je po částech lineární a spojitá.

- Od součinové konjunkce $\underset{P}{\wedge}$ je odvozena Goguenova (též Gainesova) implikace

$$\alpha \overset{R}{\underset{P}{\to}} \beta = \begin{cases} 1 & \text{pro } \alpha \le \beta, \\ \dfrac{\beta}{\alpha} & \text{jinak.} \end{cases}$$

Má jediný bod nespojitosti, (0, 0).

2.3 Predikátová vícehodnotová logika

Struktury pravdivostních hodnot

Podobně jako v klasická logika tak i fuzzy logika pracuje s termy, formulemi, spojkami a kvantifikátory, jejichž repertoár je však mnohem širší. Spojky a kvantifikátory lze rozdělit na základní a doplňující. Základní spojky ve fuzzy logice jsou disjunkce ($\vee$), konjunkce ($\wedge$), silná konjunkce (&) a implikace (=>). Interpretace těchto spojek je analogická klasické logice. Dva druhy konjunkce představují dva limitní případy konjunkce, kdy mezi oběma kondukty existuje nějaký vnitřní vztah. Nabývá-li pravdivost obou konjunktů pouze hodnot 0 nebo 1, pak oba typy konjunkce jsou totožné. Základní kvantifikátory jsou existenční ($\exists$) a obecný ($\forall$).

Jazyk fuzzy logiky prvního řádu

I. Abecedu jazyka fuzzy logiky prvního řádu tvoří:

a) Proměnné *x*, *y*, …

b) Konstanty *c, d, r, …*

c) Symboly pro pravdivostní hodnoty α; $\alpha \in L$.

d) n-ární funkční symboly $f^{(G)}, g^{(H)}, \ldots$ s různými indexy *G, H, …*

e) n-ární predikátové symboly *p, q, …* a vyznačený symbol rovnosti =.

f) Binární spojky $\vee, \wedge, \&, =>$ a množina n-árních spojek $\{o_j; j \in jop\}$ (árnost o_j závisí na indexu *j*)

g) Symboly pro obecný kvantifikátor $\forall$ a existenční kvantifikátor $\exists$ a množina zobecněných kvantifikátorů $\{Q_j; j \in Jq\}$.

h) Pomocné symboly (,) apod.

II. Termy

a) Proměnná nebo konstanta je term bez indexu.

b) Jestliže $f^{(G)}$ je n-ární funkcionální symbol a (a_1, … a_n) jsou termy bez indexu nebo se stejn7m indexem G, pak $f^{(G)}(a_1, \ldots a_n)$ je term s indexem *G*. Jestliže v textu neuvedeme u termu index, pak je to term bez indexu nebo na indexu nezáleží. Každou n-ární fuzzy funkci lze nahradit n + 1-ární fuzzy relací. Můžeme tedy funkcionální symboly z jazyka vynechat. Pak za termy budeme považovat pouze konstanty a proměnné.

III. Formule

a) Symbol α, $\alpha \in L$ pro pravdivostní hodnotu je (atomická) formule.

b) Jestliže *a, b,* $a_1, \ldots, a_n$ jsou termy s libovolným indexem, pak $a = b$ a $p(a_1, \ldots, a_n)$ jsou (atomické) formule.

c) Jestliže $\varphi, \psi, \varphi_1, ..., \varphi_n$ jsou formule, pak $\varphi \vee \psi, \varphi \wedge \psi, \varphi \& \psi, \varphi => \psi, o_j(\varphi_1, ..., \varphi_n)$ pro $j \in Jop, (\forall x)\varphi, (\exists x)\varphi, (Q_j x)\varphi$ pro $j \in Jq$ jsou formule.

Symboly α, $\alpha \in L$ se nazývají interní pravdivostní hodnoty. Speciálně 1 je *pravda* a 0 je *nepravda*.

Analogicky jako v klasické logice jsou zavedeny pojmy *volné* a *vázané* proměnné a *substituovatelný* term. Jestliže substituujeme term do formule, musíme dávat pozor na index: term *b* je substituovatelný do termu *a* za proměnnou *x*, jestliže *a* a *b* mají buď stejný index, nebo jeden z nich je bez indexu. Jestliže *a, b* jsou termy se stejným indexem a φ je formule, pak $a_x[b]$ a $\varphi_x[b]$ označují term, resp. formuli, která vznikne substitucí termu *b* za proměnnou *x*. Výraz $\varphi(x_1, ..., x_n)$ označuje formuli, jejíž všechny volné proměnné se vyskytují mezi proměnnými $x_1, ..., x_n$.

Interpretace jazyka

I. Interpretace symbolů jazyka

a) Konstantám jsou přiřazeny fuzzy jednotky ve tvaru $\{\alpha/d\}$, kde $\alpha \in L$ a $d \in D$, které budou označovány symbolem d_α. Často bude použit zápis $d_\alpha \in D$ místo $d_\alpha \subseteq D$.

b) Každému n-árnímu funkcionálnímu symbolu $g^{(G)}$ je přiřazena *n*-ární fuzzy funkce s oborem $G \subseteq D$.

c) Každému n-árnímu predikátovému symbolu *p* je přiřazena *n*-ární fuzzy relace $p_{\mathscr{D}} \subseteq D^n$.

II. Interpretace termů

Termy jsou interpretovány jako fuzzy jednotky.

a) Jestliže *a* je d_α, $\alpha \in L - \{0\}$, kde d_α je jméno fuzzy jednotky $\{\alpha/d\}$, pak

$\mathscr{D}(a) = \{\alpha/d\}$.

Jestliže *a* je konstanta bez indexu α, pak $\mathscr{D}(a) = \{1/d\}$.

b) Jestliže $a = g^{(G)}\left(b_1^{(G)}, ..., b_n^{(G)}\right)$, pak

$$\mathscr{D}(a) = \left\{ Gg_{\mathscr{D}}(\mathscr{D}(b_1^{(G)}), ..., \mathscr{D}(b_n^{(G)})) \Big/ g_{\mathscr{D}}(\mathscr{D}(b_1^{(G)}), ..., \mathscr{D}(b_n^{(G)})) \right\},$$

kde $\mathscr{D}(a) = \left\{g_{\mathscr{D}}(\mathscr{D}(b_1^{(G)}), ..., \mathscr{D}(b_n^{(G)}))\right\}$ označuje složenou fuzzy funkci.

Jestliže se v jazyce neuvažují funkční symboly, pak se interpretace termů velmi zjednoduší, neboť není třeba vůbec uvažovat fuzzy jednotky a stačí položit $\mathscr{D}(a) = d \in D$.

III. Interpretace formulí

Nechť $\mathscr{D}$ je struktura pro jazyk $\mathscr{J}$ a $\varphi, \psi, \varphi_1, ..., \varphi_n \in F_{\mathscr{J}}$. Pravdivostní ohodnocení uzavřených formulí $\chi \in F_{\mathscr{J}}$ je definováno takto:

a) $\chi =_D \alpha, \alpha \in L$

$$\mathscr{D}(\chi) = \alpha,$$

b) $\chi =_D p(a_1, ..., a_n)$

$$\mathscr{D}(\chi) = p_{\mathscr{D}}(\mathscr{D}(a_1), ..., \mathscr{D}(a_n)),$$

c) $\chi =_D a = b$

$$\mathscr{D}(\chi) = \begin{cases} 1, \textit{ jestliže } D('a) = D(b), \textit{ tj. jestliže } D(a) = d_\alpha \textit{ a } D(b) = d'_\beta, \alpha = \beta \textit{ a } d = d' \\ 0, \textit{ jinak}, \end{cases}$$

d) $\chi =_D \varphi \vee \psi$

$$\mathscr{D}(\chi) = \mathscr{D}(\varphi) \vee \mathscr{D}(\psi),$$

e) $\chi =_D \varphi \wedge \psi$

$$\mathscr{S}(\chi) = \mathscr{S}(\varphi) \wedge \mathscr{S}(\psi),$$

f) $\chi =_D \varphi \,\&\, \psi$

$$\mathscr{S}(\chi) = \mathscr{S}(\varphi) \otimes \mathscr{S}(\psi),$$

g) $\chi =_D \varphi \Rightarrow \psi$

$$\mathscr{S}(\chi) = \mathscr{S}(\varphi) \rightarrow \mathscr{S}(\psi),$$

h) $\chi =_D o_j(\varphi_1, ..., \varphi_n),\ j \in Jop$

$$\mathscr{S}(\chi) = o_j\left(\mathscr{S}(\varphi_1), ..., \mathscr{S}(\varphi_n)\right),$$

i) $\chi =_D (\exists x)\varphi$

$$\mathscr{S}(\chi) = \bigvee_{d \in_\alpha D} \mathscr{S}(\varphi_x[d_\alpha])$$

j) $\chi =_D (\forall x)\varphi$

$$\mathscr{S}(\chi) = \bigwedge_{d \in_\alpha D} \mathscr{S}(\varphi_x[d_\alpha]),$$

k) $\chi =_D (\mathsf{Q}_j x)\varphi,\ j \in Jq$

$$D(\chi) = Q_j \underset{d \in_\alpha D}{} \mathscr{S}(\varphi_x[d_\alpha]).$$

… kvantifikátor se označuje jako Q a Q jako zobecněná operace, která je jeho interpretací ve struktuře $\mathscr{S}$.

2.4 Lingvistická logika

Mnohem radikálnějším zobecněním klasické dvouhodnotové logiky, než je tomu u vícehodnotové logiky, je logika lingvistická. Je založená na předpokladu, že pro člověka je mnohem přirozenější vyjadřovat pravdivostní hodnoty pomocí slov nebo termínů, např. velký, větší, obrovský apod. Přičemž významy těchto slov jsou fuzzy množiny v množině L pravdivostních hodnot, nikoli individuální pravdivostní hodnoty. Tato logika má složitější strukturu jazykových pravdivostních hodnot než vícehodnotová logika, navíc struktura není uzavřená a může se stát že výsledkem je fuzzy množina v L, která nemá odpovídající jazykové vyjádření a je zapotřebí lingvistické aproximace, což právě velice často vede ke kritice této logiky.

Jelikož se jedná o lingvistickou logiku, je základem jazykových pravdivostních hodnot právě jazyková proměnná

$$\left\langle \mathrm{Pr}\, avdivost, \langle 0, 1\rangle, \mathscr{T}(pravdivost), G, M_G \right\rangle,$$

Kde G je bezkontextová gramatika $G = \langle V_T, V_N, I, P \rangle$, V_T je množina termů {*pravda, absolutně pravda, nedefinováno, a, nebo, ne, velmi, více méně, zhruba, značně (,)* } a P je množina přepisovacích pravidel

$$I \to A \mid I \text{ nebo } A,$$

$$A \to B \mid A \text{ a } B,$$

$$B \to C \mid \text{ne } C,$$

$$C \to (I) \mid D,$$

$$D \to mD \mid p,$$

kde p je jeden z terminálních symbolů pravda, *absolutně pravda*, *nedefinováno* a *m* je jazykový operátor. Sémantické pravidlo $\mathcal{M}_G$ je asociované s G, kde atomické termy *pravda, absolutně pravda* a *nedefinováno* mají přiřazen význam

$$M(\textit{pravda}) = \bigcup_{\tau \in \langle 0,1 \rangle} \tau / \tau,$$

$$M(\textit{absolutně pravda}) = \{1/1\},$$

$$M(\textit{nedefinováno}) = \bigcup_{\tau \in \langle 0,1 \rangle} 1/\tau$$

A přepisovacímu pravidlu $A \to A$ a B je v P přiřazeno pouze pravidlo

$$M(A) \to M(A) \cap M(B).$$

I s těmito pravdivostními hodnotami lze provádět logické operace, a pro odlišení logických spojek od spojek vytvářejících termy T budou vyznačeny slovem tučně.

Nechť $T_1, T_2 \in \mathcal{T}(\textit{pravdivost})$ jsou termy s významy

$$M(T_1) = \bigcup_{\tau \in \langle 0,1 \rangle} T_1\tau / \tau,$$

$$M(T_2) = \bigcup_{\tau \in \langle 0,1 \rangle} T_{21}\tau / \tau.$$

Disjunkce **nebo**

$$M(T_1 \textbf{ nebo } T_2) = \bigcup_{\tau_1, \tau_2 \in \langle 0,1 \rangle} (T_1\tau_1 \wedge T_2\tau_2) / \tau_1 \vee \tau_2,$$

konjunkce **a**

$$M(T_1 \textbf{ a } T_2) = \bigcup_{\tau_1, \tau_2 \in \langle 0,1 \rangle} (T_1\tau_1 \wedge T_2\tau_2) / \tau_1 \wedge \tau_2,$$

odvážná konjunkce **i**

$$M(T_1 \textbf{ i } T_2) = \bigcup_{\tau_1, \tau_2 \in \langle 0,1 \rangle} (T_1\tau_1 \wedge T_2\tau_2) / \tau_1 \oplus \tau_2,$$

negace **ne**

$$M(\mathbf{ne}\ T_1) = \bigcup_{\tau\in\langle 0,1\rangle} {(T_1\tau)}\big/_{1-\tau},$$

implikace **implikuje**

$$M(T_1\ \mathbf{implikuje}\ T_2) = \bigcup_{\tau_1,\,\tau_2\in\langle 0,1\rangle} {(T_1\tau_1 \wedge T_2\tau_2)}\big/_{\tau_1 \to \tau_2}.$$

Vyčíslení pravdivostních hodnot

Při popisu nějaké situace vágními pojmy modelovanými pomocí fuzzy množin, se zpravidla dospěje opět k jazykovým pravdivostním hodnotám. Například je možno zjišťovat pravdivost výroku „Tato restaurace je na vysoké úrovni“, je-li dobře známo, že „Tato restaurace je průměrná“. Obecně musí být úloha formulována takto:

Nechť **L** je výrok

„X je $\mathscr{A}$“,

popisující stav nějakého objektu nebo výsledek nějaké činnosti a **V** je výrok

„X je $\mathscr{B}$“,

vyjadřující názor subjektu na tento objekt nebo činnost.

Je tedy třeba vyčíslit pravdivost **T(L, V)** výroku **V** vzhledem k výroku **L**. Přitom $\mathscr{A}$ a $\mathscr{B}$ jsou pojmy, jejichž význam je modelován pomocí fuzzy množin

$$A, B \subseteq U, \text{ tj. } M(\mathscr{A}) = A \text{ a } M(\mathscr{B}) = B.$$

Jsou-li **L** a **V** uvedené výroky a *A*, *B* jsou fuzzy množiny

$$A = \bigcup_{x\in U} {A_x}\big/_{x},$$

$$B = \bigcup_{x\in U} {B_x}\big/_{x},$$

pak pravdivost **T(L, V)** výroku **V** vzhledem k výroku **L** je jazyková pravdivostní hodnota s významem

$$\mathbf{T}(\boldsymbol{L},\boldsymbol{V}) = \bigcup_{\tau\in\langle 0,1\rangle} {T_\tau}\big/_{\tau},$$

$$\text{kde } T_\tau = \bigvee_{\substack{x\in U\\ \tau = Bx}} Ax \quad a \quad \tau = Bx.$$

Modely na základě těchto lingvistických výrazů umožňují jednoduše popsat různé situace z „reálného života“. Výpočty jsou pak také výpočetně jednoduché na rozdíl například od dedukce v klasických logikách, kde hledání daného důsledku resp. nesplnitelnosti množiny formulí je kombinatorickým problémem. Překvapivě ale tyto modely dobře reprezentují některé situace.

Popis procesu výběru nejvhodnějšího vysavače pomocí lingvistických výrazů.
Vhodnou volbou potřebných lingvistických proměnných lze zjednodušit situaci na dvě vstupní proměnné a jednu výstupní proměnnou:

Technická úroveň (T) – rozsah ⟨200, 450⟩ (vstupní)
Sací výkon (S) – rozsah ⟨1, 5⟩ (vstupní)
Technická kvalita (K) – rozsah ⟨0, 1⟩ (výstupní)

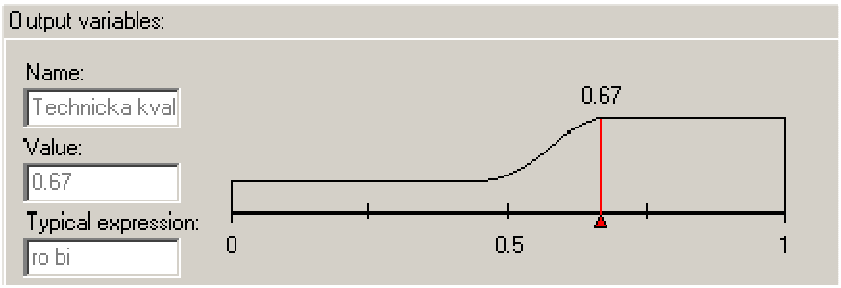
Na těchto proměnných pak lze formulovat pravidla. Například lze vyjádřit, že platí:
Když sací výkon je velmi malý (ve sm) a technická úroveň je střední (me), pak technická kvalita je malá (sm).
Všech 14 pravidel lze vidět na obrázku vyjádřené pomocí anglických zkratek výrazů malý(sm), střední(me), velký(bi) a pomocí takzvaných modifikátorů, které upravují význam základních výrazů – velmi (ve), velmi zhruba (vr), zhruba (ro), víceméně (ml) apod.

Saci vykon & Technicka uroven --> Technicka kvalita

	Saci vykon	Technicka uroven	Technicka kvalita	Group	Inconsiste
1.	ve sm	me	sm		
2.	sm	sm	sm		
3.	sm	me	vr sm		
4.	sm	bi	vr bi		
5.	me	sm	ro sm		
6.	vr bi	bi	ro bi		
7.	me	me	me		
8.	ro bi	me	vr bi		
9.	ro bi	bi	ml bi		
10.	bi	me	ml bi		
11.	bi	sm	me		
12.	bi	bi	bi		
13.	ve bi	bi	ve bi		
14.	ve bi	me	bi		

Obrázek 4: ukázka báze pravidel

Pak nastává fáze inference, kdy lze buď zadat konkrétní hodnoty proměnných pro zkoumaný objekt anebo pomocí vhodné tzv. fuzzyfikační funkce získat nejvhodnější lingvistický výraz. Pak proběhne nalezení pravidel, které se hodí dané situaci a na nich se zjistí výraz výstupní proměnné. Ten pak zpětně defuzzyfikuje vhodná funkce a vznikne opět „přesná“ hodnota (slovo přesná by mohlo být mylně interpretováno, tato hodnota už má za sebou fuzzyfikaci a defuzzyfikaci, které už dopředu „rozmazávají“ význam hodnoty).



Obrázek 5: Příklad inference

Jde o konkrétní vysavač se sacím výkonem 400 a technickou úrovní ohodnocenou známkou 2.5.

Pravidlo 14, které se hodí k této situaci vyjadřuje:
Když je sací výkon velmi velký a technická úroveň je střední, pak technická kvalita je velká.

Na obrázku lze pak vidět výsledek inference – fuzzy množinu reprezentující hodnotu výstupní proměnné a její defuzzyfikaci na hodnotu 0.67.

Pokuste si uvědomit, které vlastnosti z dvouhodnotové logiky se dají zobecnit na fuzzy logiku a které ne?

- Fuzzy matematika
- Fuzzy logika
- Lingvistická logika
- Motivace pro studium fuzzy logiky

3 Deduktivní systém

V této kapitole se dozvíte následující pojmy:

- Struktura a účel deduktivního systému
- Tvorba jednotlivých součástí systému

Po jejím prostudování byste měli být schopni:

- Vystavět jednoduchý deduktivní systém nebo jeho část
- Provádět dedukci různými automatizovanými formálními systémy.

Klíčová slova této kapitoly:

Symbolická logika, logická dedukce, formální dedukce

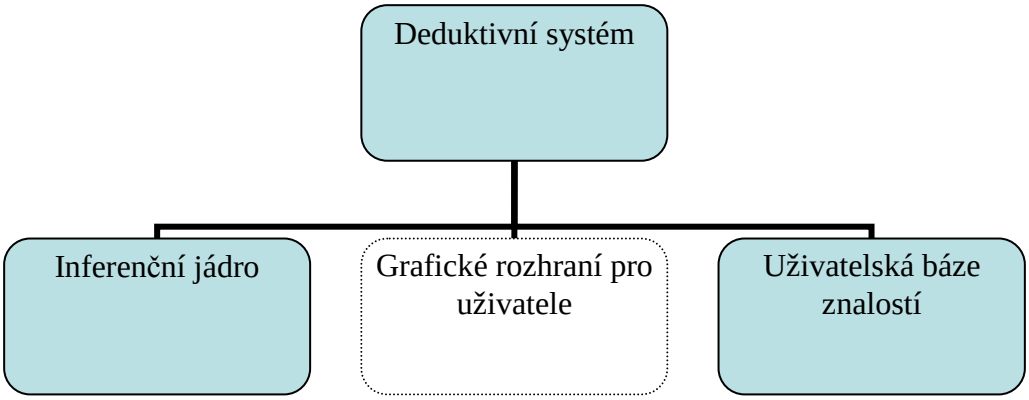
Doba potřebná ke studiu: 12 hodin

Průvodce studiem

Studium této kapitoly je poměrně náročné zejména pro ty z Vás, kteří dosud nemají žádné znalosti z logiky. V takovém případě Vám zřejmě některé příklady budou připadat obtížně pochopitelné, ovšem nenechte se tím odradit, neboť pochopením této části se Vám usnadní studium následujících kapitol. Pochopení tvorby a struktury deduktivního systému je klíčové pro celé studium. Na studium této části si vyhraďte alespoň 12 hodin. Doporučujeme studovat s přestávkami vždy po pochopení jednotlivých podkapitol.

3.1 Struktura deduktivního systému

Deduktivní systém má svou zvláštní strukturu, kde je především nutno oddělit od sebe inferenční jádro a bázi znalostí. V tom se podobá PROLOGu a expertním systémů. Systémy mají často také pohodlné grafické rozhraní, které je dnes již v uživatelských aplikacích standardem. Ovšem mnoho systémů má také jen holé jádro, které pro vstup dat používá obyčejný souborový systém. Zjednodušeně řečeno jde o konzolovou aplikaci bez GUI (Graphical User Interface), která pro svůj běh potřebuje v souboru zadanou bázi znalostí, případně nastavení vlastností inference apod. Do výstupního souboru pak aplikace vytvoří výsledky inference případně statistiky. Většinou se také jedná o soubory textově orientované, tedy lehce editovatelné s pomocí jakéhokoliv editoru textu.

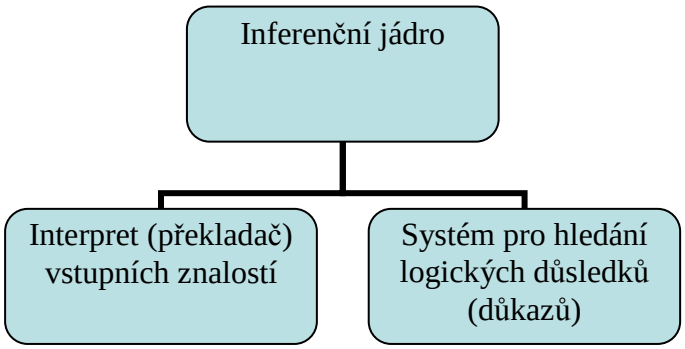


Obrázek 6: Deduktivní systém

3.2 Inferenční jádro

Inferenční jádro má dvě hlavní komponenty:

2. Interpret (překladač) vstupní báze znalostí
3. Systém pro hledání logických důsledků a jejich důkazy.



Obrázek 7: Inferenční jádro

První část vypadá na první pohled jednoduše, ale najít vhodný způsob reprezentace těchto znalostí pro hledání důkazů, není nijak jednoduchý. Jednak je potřeba nejprve provést kontrolu, zda vyžadované vstupní soubory jsou v souladu s jazykem pro popis znalostní báze a dotazů. V pozitivním případě je potřeba převést je do vnitřní reprezentace, kterou bude schopen systém pro hledání důkazů využít co nejefektivněji.

3.3 Definice jazyka

Aby byl schopen počítač pracovat s logickým výrazem (resp. s jakýmkoliv jiným syntaktickým elementem v libovolném zdrojovém kódu) musíme nějakým vhodným formálním prostředkem zapsat definici tohoto výrazu. Tyto výrazy vlastně tvoří specifický jazyk. Pro definice struktur programovacích jazyků se velmi dobře hodí takzvaná Backusova-Naurova forma. Je podobná bezkontextovým gramatikám (má terminální symboly, neterminální symboly, pravidla pro přepis neterminálů), ale navíc zjednodušuje zápis obvyklých technik jako je opakování nějakého řetězce (třeba příčítaní podvýrazů v aritmetických výrazech se může dělat opakovaně - u bezkontextových gramatik bychom toto museli řešit krkolomně pomocí jakési "rekurze"). Bezkontextová gramatika (BKG) je jedním z velmi vhodných způsobu zápisu syntaxe jazyků. Syntaxí zde rozumíme jejich jazykovou strukturu. Umožňuje totiž vyjádřit většinu technik, které například u programovacích jazyků používáme. Jde o alternativu několika možností, opakování stejného jazykového výrazu a hlavně vnořování celých rozvětvených struktur mezi sebou. Poslední zmiňovaná technika je právě onou technikou, kterou neumíme vyjádřit pomocí jazyků regulárních, ale teprve pomocí bezkontextových jazyků. Zkusme si představit velmi omezenou část nějakého programovacího jazyka - například strukturu aritmetického výrazu. Principiálně je většina jiných struktur velmi podobných (např. sekvence příkazů je analogická opakovanému sčítání podvýrazů!).

Příklad 9.

Sestrojme BKG pro jazyk složený z aritmetických výrazů s operandem x, operacemi +,* a umožňující vnořovat další podvýrazy stejného typu pomocí symbolů závorek (,). Kupříkladu se může jednat o výraz: x * (x + x + x)
Gramatiku sestrojíme hierarchicky - tedy aby byla rozlišena priorita operátorů a využijeme "rekurzivní" vlastnosti přepisu neterminálu, abychom docílili možnosti generovat opakovaně sčítání a násobení.
G = (S, A, B,x, *, +, (,), S, P)
P: $S \rightarrow^1 A + S$, $S \rightarrow^2 A$ (opakovaný přepis na S nám umožní generovat libovolně mnoho sčítání struktury A)
$A \rightarrow^3 B * A$, $A \rightarrow^4 B$ (opakovaný přepis na A nám umožní generovat libovolně mnoho násobení struktury B)
$B \rightarrow^5 (S)$, $B \rightarrow^6 x$ (rekurzivním přepisem na S můžeme vnořit libovolně mnoho podvýrazů zcela stejné struktury jako výraz sám do závorek anebo ukončit generování operandem x) (Pozn.: index u symbolu určuje pomocné číslo pravidla)
V této gramatice pak lze snadno generovat například výše uvedený výraz x * (x + x + x):
$S \Rightarrow^2 A \Rightarrow^3 B * A \Rightarrow^6 x * A \Rightarrow^4 x * B \Rightarrow^5 x * (S) \Rightarrow^1 x * (A + S) \Rightarrow^4 x * (B + S) \Rightarrow^6 x * (x + S) \Rightarrow^1 x * (x + A + S) \Rightarrow^4 x * (x + B + S) \Rightarrow^6 x * (x + x + S) \Rightarrow^2 x * (x + x + A) \Rightarrow^4 x * (x + x + B) \Rightarrow^6 x * (x + x + x)$
(Pozn.: index u symbolu $\Rightarrow^i$ určuje pomocné číslo pravidla)
Dalším přehledným a hlavně v praxi ještě více využívaným způsobem zápisu syntaxe jazyka je takzvaná Backusova-Naurova forma (BNF). Jde o zápis podobný bezkontextové gramatice, ale přitom bližší spíše programátorům, resp. praxi.

BNF obsahuje podobně jako BKG neterminály, které se uvádějí do úhlových závorek a přepisují skrze symbol := na řetězce terminálních a neterminálních symbolů. Jde tedy o pravidla tvaru:
$\langle X \rangle :: = \alpha_1 \ldots \alpha_n$
Pro přehlednější zápis je však ještě lepší modifikace BNF zvaná EBNF (Extended BNF) - rozšířená BNF, která zjednodušuje zápis opakovaně používaných, příp. podmíněně vyskytujících se výrazů. Umožňuje následující zápisy:
$\{\alpha\}$ - znamená, že výraz se vyskytuje v libovolném počtu (ekvivalent operace iterace)
$\{\alpha\}_n^m$ - znamená, že výraz se vyskytuje v počtu nejméně n a nejvýše m (ekvivalent operace mocniny od n do m)
$[\ \alpha]$ - znamená, že výraz se může a nemusí na daném místě vyskytnout - je to ekvivalentní zápisu $\{\alpha\}_0^1$

Příklad 10.

Gramatika z předchozího příkladu by v BNF mohla být zapsána například takto:

⟨aritmeticky vyraz+⟩:: = ⟨aritmeticky vyraz*⟩{+⟨aritmeticky vyraz*⟩}
⟨aritmeticky vyraz*⟩:: = ⟨operand/podvyraz⟩{*⟨operand/podvyraz⟩}
⟨operand/podvyraz⟩:: = (⟨aritmeticky vyraz+⟩) | x

BNF umožňuje přehledný zápis a navíc i jednoduchý přechod k některým typům SA, S pomocí BNF je zapsána například celá gramatika jazyka Pascal v učebnici. Příkladem může být deklarace podmíněného příkazu:

⟨podmineny prikaz⟩:: = if ⟨booleovsky vyraz⟩then ⟨prikaz⟩[else ⟨prikaz⟩]

Pro definici našeho logického výrazu použijeme následující gramatiku:

⟨Exp1⟩:: = ⟨Exp2⟩{ = ⟨Exp2⟩}
⟨Exp2⟩:: = ⟨Exp3⟩{ > ⟨Exp3⟩}
⟨Exp3⟩:: = ⟨Exp4⟩{| ⟨Exp4⟩}
⟨Exp4⟩:: = ⟨ICE⟩{& ⟨ICE⟩}
⟨ICE⟩:: = ~ ⟨ICE⟩| (⟨Exp1⟩) | 0 | 1 | | A | …| Z | a | …| z

Význam symbolů:

0 - nepravda , 1 - pravda , A ... Z, a ... z - atomy (logické proměnné), ~ - negace , = - ekvivalence , > - implikace , | - disjunkce , & - konjunkce

3.4 Syntaktická analýza a překlad

Prvním důležitým úkolem při překladu z nějakého zdrojového jazyka do cílového je především syntaktická analýza - tedy kontrola, zda je text ve zdrojovém jazyce správně zapsán. V nejjednodušším případě pouhé kontroly typu ANO/NE (program je správně/není správně syntakticky zapsán) se jedná o takzvanou syntaktickou analýzu (dále budeme zkracovat SA). V anglicky psaných zdrojích se setkáte spíše s jednoslovným označením "parsing". O daném postupu - algoritmu jak tuto SA provést, pak hovoříme jako syntaktickém analyzátoru (anglicky "parser").
Velice oblíbenou, jednoduchou a přehlednou metodou vedoucí přímo k hotovému a dobře čitelnému analyzátoru v libovolném strukturovaném programovacím jazyce je metoda rekurzivního sestupu (Recursive Descent Parsing). Metoda je založena na principu analýzy ßhora dolů", tedy se snažíme hledat odvození slova podle dané gramatiky od počátečního neterminálu v hierarchii směrem "dolů".
Metoda rekurzivního sestupu spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar (načítající vždy následující symbol slova) před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen rekurzivně voláním příslušné procedury. Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován tzv. LL(k) gramatikou. Z hlediska časové složitosti (obecně) jde opět o obecně neefektivní metodu s exponenciální časovou složitostí, nicméně pro jednoduché gramatiky z praxe je použitelná a zejména je její výhodnou vysoká čitelnost kódu ve vztahu k výchozí gramatice. Navíc existuje modifikace tzv. packrat parser, která pro omezenou třídu takzvaných parsing expression grammars pracuje v **lineární čase**. Také je tento postup flexibilní, protože umožňuje kdykoliv změnit a přidat syntaktický element bez nutnosti měnit celý kód, ale pouze dotčenou část gramatiky (například změna struktury číslo z celého čísla na reálné znamená pouze změnu procedury reprezentující tento element). Podívejme se nyní na příklad gramatiky pro generování aritmetických výrazů se sčítáním, násobením, číslicemi a vnořenými závorkovanými strukturami. Vyhodnocování takového výrazu strojově je pak velmi jednoduché pomocí tzv. postfixové notace (reverzní polská notace), kde vždy platí, že operátory se vyskytují až za jeho operandy. Pro vyhodnocení takové notace nám postačí jen datová struktura typu zásobník a nijak to neovlivní lineární časovou složitost celého procesu včetně syntaktické analýzy.

Příklad 11.

Nejprve sestrojme mírně modifikovanou gramatiku (oproti příkladu z minulé kapitoly) v Backusově-Naurově formě:

⟨Vyraz⟩:: = ⟨Term⟩{+⟨Term⟩}
⟨Term⟩:: = ⟨Faktor⟩{*⟨Faktor⟩}
⟨Faktor⟩:: = (⟨Vyraz⟩)|0|1|2|…|9

Nyní se schématicky pokusíme ukázat (nejde o zcela hotový kód, ale o jeho fragmenty po částech, které byly dohromady úspěšně testovány), jak bychom sestrojili SA metodou rekurzivního sestupu pro tuto gramatiku v jazyce Pascal. Tento kód, pak umožňuje nejen SA, ale i detekci možných chyb. Nejprve sestrojíme proceduru, která zapouzdřuje celou činnost SA. Její hlavička může vypadat například takto:

```
program Preklad;
var ch:char;             {aktualni zpracovavany znak}

    infixProg, postfixProg:string;     {globalni prom. unitu
pro uchovani vyrazu}
    errProg, posProg, infixpos:word;
{globalni prom. cisla chyby}

procedure SyntaktickaAnalyza(infix:string;var err,pos:word; var
postfix:string);
          {procedura analyzuje aritmetický výraz infix, postfix
obsahuje
           postfixovou notaci vhodnou pro vyhodnoceni
zasobnikem
           err obsahuje číslo chyby, pos obsahuje pozici ,kde
analýza skončila}

procedure Term;forward;
procedure Faktor;forward;
```

Používají se proměnné infixpos (pozice aktuálně čteného znaku ze vstupu), ch (aktuální znak). Analyzátor dále obsahuje nezbytný lexikální analyzátor pro načítaní jednotlivých symbolů (v našem zjednodušeném případě jde o jednoznakové symboly). Lexikální analýza je realizována procedurou GetChar, která ukládá znak do proměnné ch a případně provede detekci chybové situace err=2, pokud načteme zcela nepřípustný znak. V rámci SA je pak otázka jen přidat několik vhodně umístěných přiřazení do výstupní proměnné postfix, kde je přeložený výraz do postfixové notace, kterou lze velmi jednoduše algoritmicky vyhodnotit.

```
procedure Getchar;       {čte znak z infixu do proměnné ch}
begin
  if err=0 then
    begin
      Inc(infixpos);
      if infixpos<=Length(infix) then ch:=infix[infixpos] else
ch:=#0;
      ch:=Upcase(ch);
      if not((ch in ['0'..'9'])or(ch in ['(',')','*','+',#0]))
then err:=2;
         {ošetření nežádoucích znaků}
    end;
end;
```

Jádrem analyzátoru jsou jednotlivé rekurzivní procedury Výraz (sčítaní), Term (násobení), Faktor (číslice, vnořený závorkovaný výraz). Výraz přesně podle BNF buď volá podřízený Faktor nebo čte terminální symboly.

```
procedure Vyraz;               {výraz s nižší prioritou}
begin
  if err=0 then
    begin
      Term;
      while (ch='+') do
        begin
                 Getchar;       {sčítání}
                 Term;
         postfix := postfix + '+';
        end;
    end;
end;

procedure Term;               {výraz s vyšší prioritou}
begin
  if err=0 then
    begin
      Faktor;
      while (ch='*') do
        begin
                 Getchar;           {násobení}
                 Faktor;
         postfix := postfix + '*';
        end;
    end;
end;
```

A dále musíme sestrojit proceduru pro Faktor, která bude mít vzhledem k jinému charakteru přepisovaného řetězce i jiný kód.

```
procedure Faktor;
  {synt. analýza operandu}
begin
  if err=0 then
    begin
      case ch of
      '0'..'9':
          begin
            postfix:=postfix+ch;                  {anal.
číslic}
            Getchar;
          end;
      '(':begin
            Getchar;
                        {analýza výrazu se závorkou}
            Vyraz;
            if (ch<>')')and(err=0) then err:=4   {chyba- není
ukončen závorkou}
            else if err=0 then
              begin
                Getchar;
              end;
          end;
```

```
      else if err=0 then err:=5;    {nebyl detekován ani
vyraz,ani číslice}
      end;
    end;
end;
```

Faktor tedy rozlišuje dvě situace - buď jde o číslici nebo jde o výraz začínající závorkou a ukončený opačnou závorkou. Logicky tedy můžeme odhalit další dvě chyby (err=4, když chybí závorka, err=5, když není detekován ani výraz ani číslice). Postfixová notace se generuje postupně - každá číslice se vloží okamžitě po načtení, znak operátoru se přidá, až po zpracování podstromu.
Pozn. Nebyl by problém se zcela vyhnout tvorbě postfixu a stejným rekurzivním principem přímo vyčíslovat postupně hodnotu výrazu (procedury Faktor, Term a Výraz by pak měly návratovou hodnotu rovnou výsledku po aplikaci všech operací na dané úrovni podvýrazu).
Pozn. Samozřejmě, že chybové detekce by mohly odhalit ještě další problematické konstrukce - např. skončení nejvyššího volání procedury Výraz před přečtením posledního znaku apod. Vlastní tělo procedury pro SA, pak provede volání počátečního neterminálu a odhalí některé chyby dodatečně po přečtení výrazu např. že sice skončila nejvyšší procedura Vyraz, ale ještě ve vstupu jsou nějaké znaky.

```
begin
  err:=0;           {inicializace prom.}
  infixpos:=0;
  postfix:='';
  Getchar;
  Vyraz;            {zavolani syntakticke analyzy vyrazu}
  if (ch=')') and (err<>5) then
    begin
      err:=6;
      pos:=0;
    end;     {osetreni prebytecne prave zavorky}
  if  (ch<>#0)  and  (err=0)  then  err:=1;    {osetreni  konce
kompilace-neni konec
                                       infixu}
  pos:=infixpos;               {nastaveni navratovych hodnot}
end;
```

Nakonec můžeme v hlavním programu tuto proceduru použít.

```
begin
  infixProg := '5+3*2';
  SyntaktickaAnalyza(infixProg, errProg, posProg, postfixProg);
  Writeln('Doslo k chybe:', errProg);
  if err<>0 then exit;
  Writeln('Postfixova notace:', postfixProg);
  readln;
end;
```

Ilustrujme nyní průběh výpočtu procedury Výraz na výrazu 5 + 3 * 2.

Infixová notace	Aktuální znak	Aktuální procedura	Návrat do procedury
5 + 3 * 2	5	Výraz	
5 + 3 * 2	5	Term	
+ 3 * 2	+	Faktor	Term
+ 3 * 2	+	Term	Výraz
3 * 2	3	Výraz (+)	
3 * 2	3	Term	
* 2	*	Faktor	Term
2	2	Term (*)	
		Faktor	Term (*)
		Term (*)	Výraz (+)
		Vyraz (+)	

Obrázek 8: Průběh rekurzivního sestupu

Rozeberme nyní jádro vyhodnocení výrazu v infixní formě. Toto jádro provádí kompilaci aritmetického výrazu v infixové (tedy přirozené) notaci do notace postfixové. Ta je vhodná pro zpracování pomocí počítače např. vyhodnocení pomocí zásobníku. Aritmetický výraz v infixové (přirozené) notaci 5 + 3 * 2 lze pomocí této procedury přeložit na výraz v postfixové notaci 5 3 2 * +. Ta je konstruována tak, že místo tvaru, kde je operátor mezi operandy, má operátor až za oběma operandy.
Tento výraz lze pak pomocí zásobníku vyhodnotit tak, že čteme jednotlivé symboly a provádíme s nimi tyto dvě operace:

- Je-li čtený symbol operandem, pak ulož operand na zásobník.
- Je-li čtený symbol operátorem, pak vyber ze zásobníku posledních n operandů (kde n je arita operátoru; např. pro + je n = 2). Proveď operaci dle operátoru s vybranými operandy a výsledek ulož na zásobník.

Pro výraz 5 + 3 * 2 vezměme jeho postfixovou notaci 5 3 2 * + a vyhodnoťme jej s pomocí zásobníku.

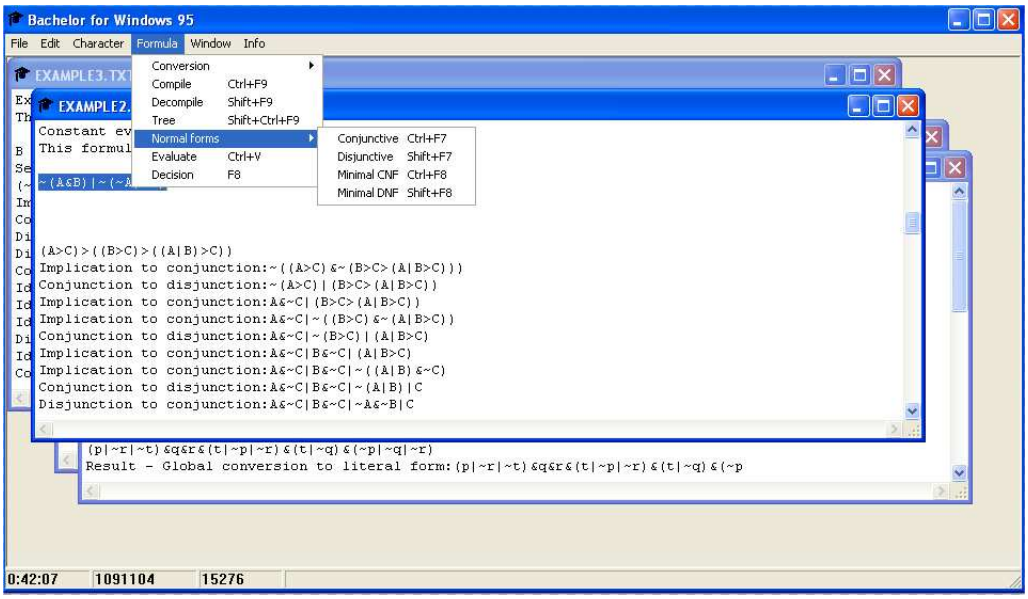
Nepřečtená část	Aktuální znak	Zásobník	Vybírané symboly	Operace
5 3 2 * +	5			
3 2 * +	3	5		
2 * +	2	3 5		
* +	*	2 3 5	2 3	2 * 3 = 6
+	+	6 5	6 5	6 + 5 = 11
		11		

Obrázek 9: Vyhodnocení postfixu

Obsah zásobníku po přečtení slova je roven hodnotě výrazu. Postup by samozřejmě bylo možno zobecnit na složitější čísla nebo proměnné, ale vyžadovalo by to složitější struktury zásobníku.

3.5 Příklad stromové reprezentace znalostí

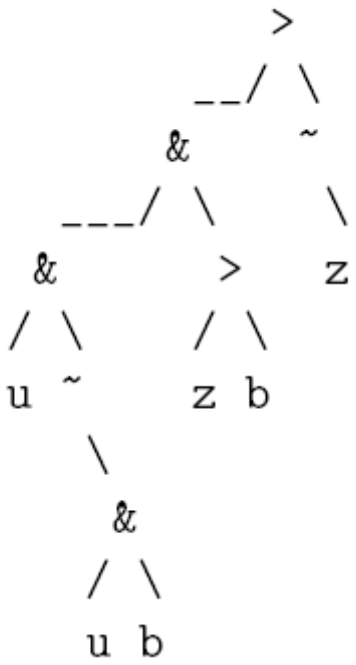
Jak jsme již pochopili v předchozí kapitole, není zápis v klasickém infixním formátu pro počítač příliš výhodný, i když my lidé jsme na něj zvyklí a umíme s ním díky výuce matematiky dobře pracovat. To platí o výrazech i strukturovaných kódech všeho druhu - programy, texty, aritmetické a logické výrazy. Ukázali jsme si způsob, jak lze jednoduše vytvořit jinou notaci (v podstatě jde jen o pořadí operátorů a operandů), kterou už počítač dokáže relativně lehce zpracovat (s pomocí zásobníku). Existuje však mnohem univerzálnější a pro algoritmizaci složitých procesů velmi vhodná metoda a to je reprezentace pomocí syntaktických stromů. Syntaktický strom je v podstatě orientovaným grafem s ohodnocenými uzly neobsahujícím cykly a u kterého, lze určit kořen stromu (jakýsi nejvyšší prvek). Každý uzel, pak může mít své potomky k nimž vedou hrany a ty uzly, které potomky nemají se nazývají listy stromu (graf tedy opravdu připomíná živý strom). Pro jednoduchost budeme stromy reprezentovat textově - jde o výstupy z již zmiňovaného doplňku vytvořeného pro čtenáře v rámci aplikace Bachelor (Habiballa, 09a). Ovšem pozor, tento doplněk je obsažen v nabídce Formula, Tree jen ve verzi pro Windows 32-bit, tedy WinBch95.exe.



Obrázek 10: Program Bachelor

Příklad 12.

Mějme logickou formuli - u&~(u&b)&(z>b)>~z.



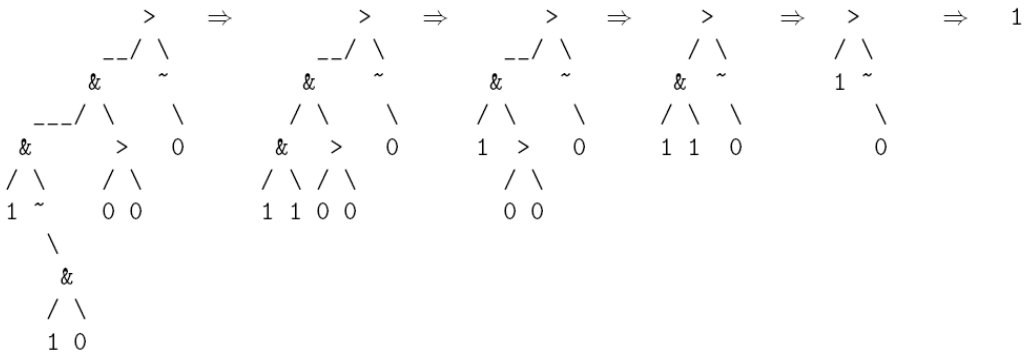
Obrázek 11: Strom formule u&~(u&b)&(z>b)>~z.

Tento strom výstižně a hlavně pro počítač dostatečně jednoduše postihuje celou hierarchii formule. V kořeni stromu je spojka implikace, což by počítač z infixní formy určoval velmi složitě (závorky, priority operátorů), zatímco pracuje-li se stromem, vše je z hierarchické struktury stromu jasné. Implikace má jako podstromy (spojené znaky / a \ a pomocnými znaky _) konjunkci složené formule a negace atomu z. Strom vůbec nemusí obsahovat závorky, neboť priorita operátorů je obsažena už v samotném stromu a není důvod ji narušovat nějakým speciálním symbolem jako jsou závorky. Algoritmy pro manipulace, změny a výpisy stromu mohou být navíc velmi chytře popsány rekurzivními algoritmy, které jsou pak velmi jednoduché (krátké).

Například pokud bychom znali hodnoty logických proměnných u, z a b, pak spočtení hodnoty výrazu se dá popsat jednoduchým rekurzivním pravidlem P:
Pokud je uzel listem vrať hodnotu proměnné,

Pokud uzel není listem, spočítej hodnoty všech potomků uzlu pravidlem P a pak tyto hodnoty zpracuj operací, která je obsažena v uzlu, a vrať výsledek (negaci logické konstanty spočítej přímo).

Podívejme se na to ilustrativně na samotném stromu. Vezměme stejný strom, pouze dosaďme hodnoty proměnných do listů u=1,z=0,b=0.



Tento příklad ilustruje operace se stromem, který nakonec vede k logické konstantě true (tudíž dedukce je správná).

3.6 Konstrukce stromu

Viděli jsme, že práce se stromem je pro počítač velmi příjemná a dá se formulovat takřka triviálními algoritmy (a bude platit i u mnohem složitějších operacích, jak uvidíme dále). Otázkou zůstává, jak s pomocí již známého rekurzivního sestupu a postfixového způsobu zápisu výrazu sestrojit syntaktický strom. Principiálně to není příliš složité, pouze technické zpracování vyžaduje dovednosti při programování dynamických datových struktur a práci s ukazateli. Využijeme stejného algoritmu se zásobníkem,

jakým bychom vyhodnocovali aritmetický výraz, pouze změníme vykonávané akce. Místo vložení číslice do zásobníku vytvoříme dynamickou proměnnou typu záznam, kterou vložíme do zásobníku. Tento záznam reprezentuje list stromu. Pokud v nějakém místě programu bychom podle starého postupu vkládali operátor, pak namísto toho vytvoříme opět dynamický záznam. Ten obsahuje odkaz na své nejvýše dva potomky (podvýrazy spojené logickou spojkou) - těmito odkazy ukážeme na poslední dva prvky v zásobníku, které předtím vybereme. Místo nich pak vložíme hotový záznam se symbolem daného operátoru. Na konci budeme mít ze správně utvořeného logického výrazu mít provázaný syntaktický strom a jeho kořen bude k dispozici na vrcholu zásobníku. Následující fragment kódu aplikace Bachelor ukazuje, jak vypadá datová struktura pro uložení jednoho uzlu stromu TSTreeNode a dále jak vypadá zapouzdřovací záznam pro dočasné uložení do zásobníku (jakýsi öbal", který potřebujeme pro uchování v zásobníkové paměti).

```
type

     PSTreeNode=^TSTreeNode;   {uzel stromu}
     TSTreeNode=record
      character:char;          {symbol : logická spojka,
                                proměnná      nebo      logická
konstanta}
      prior:byte;               {priorita symbolu - potřebujeme
ji pouze
                ke zpětnému výpisu do infixní podoby - tvorba
závorek}
      neg:boolean;             {příznak zda je uzel negován
                - nevytváříme zbytečně v programu další větev}
      left:PSTreeNode;              {ukazatel na levý podstrom
(podvýraz) - potomka}
      right:PSTreeNode;            {ukazatel na pravý podstrom
(podvýraz) - potomka}
     end;


     PSStackNode=^TSStackNode; {obal pro uchování v zásobníku}
     TSStackNode=record        {během překladu do stromu}
      node:PSTreeNode;         {ukazatel na vložený uzel}
      next:PSStackNode;         {ukazatel na následující obal v
zásobníku}
     end;
```

Princip tvorby stromu během rekurzivního sestupu si ukážeme na fragmentech analyzátoru. Popsat celý analyzátor by zabralo mnoho stran a principiálně nejde o odlišný případ, jaký jsme již předvedli na aritmetickém výrazu. Čtenář má navíc možnost se podívat do zdrojových kódů aplikace Bachelor. Ke všem neterminálům jsou v souladu s definicí BNF našeho jazyka sestrojeny procedury rekurzivního sestupu.

```
...
procedure Exp3;
 begin
  if Error=OK then
    begin
```

```
      Exp4;
      while (ch='|')and(error=OK) do
        begin
         Getchar;
         Exp4;

         if error=OK then Action('|',3);

        end;
    end;
 end;
...
procedure ICE;
 begin
 if error=OK then
  begin
   case ch of
    '~':begin
          Getchar;
          ICE;
          VPSStack^.node^.neg:=not(VPSStack^.node^.neg);
        end;
    'a'..'z','A'..'Z','0'..'1':
        begin
          Put(ch,6);
          Getchar;
        end;
    '(':begin
          Getchar;
          Exp1;
          if (ch<>')')and(error=OK) then Error:=missbra;
          Getchar;
        end
    else Error:=missexp;
    end;
  end;
 end;
```

Vidíme, že vložení listu do zásobníku nám realizuje procedura Put, která vytvoří daný uzel a jeho obal s příslušným znakem a prioritou. Pokud nám přijde znak negace, změníme u posledního vloženého atomu negaci na opačnou (teoreticky může obsahovat libovolné, množství negací za sebou). V programu je definováno mnoho chyb, které může výraz obsahovat - zde například chyba missexp vyjadřuje chybějící podvýraz a missexp chybějící uzavírací párovou závorku. Navázání vytvoření a navázání potomků uzlu, který není list (operátor), nám zajistí procedura Action.

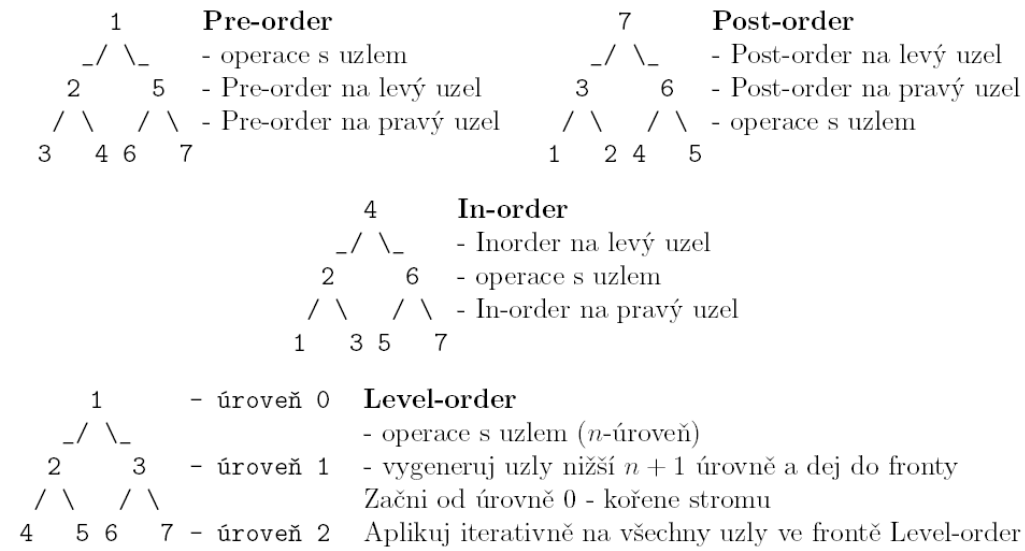
```
procedure Put(var chp:char;pr:byte);    {vytvoření a vložení
atomu}
    begin
      new(pom); {alokace nového dynamického záznamu - obalu}
      pom^.next:=VPSStack;  {původní vrchol zásobníku musíme
navázat na nový}
      VPSStack:=pom; {nový obal je první v zásobníku - vrchol}
      new(pom^.node); {v obalu vytvoříme uzel stromu}
      VPSStack^.node^.character:=chp; {uzel má požadovaný znak}
      pom^.node^.left:=nil;
```

```
      pom^.node^.right:=nil;  {levý  i  pravý  podstrom  atom  -
proměnná
                    nebo konstanta - nemá! tedy ukazuje na nil}
      pom^.node^.prior:=pr;    {nastavíme    prioritu    podle
požadavku}
      pom^.node^.neg:=false; {prvotní vytvoření nedělá negaci}

    end;
...
procedure  Action(chr:char;pr:byte);       {vlož  operátor  do
zásobníku}
 begin
   Put(chr,pr); {vlož do zásobníku nový obal s uzlem
                 o požadovaném znaku operace a prioritě}
   pom:=Cut;  {vyber  operátor  ze  zásobníku  -  budeme  s  ním
pracovat}
   pom^.node^.right:=VPSStack^.node;  {navaž  první  uzel  na
zásobníku
                    jako pravý podstrom - pozor v zásobníku je
pořadí obrácené}
   garbage:=Cut; {vyber tento uzel ze zásobníku a zničjeho
obal}
   dispose(garbage);
   pom^.node^.left:=VPSStack^.node; {navaž nyní první uzel
                              na zásobníku jako levý podstrom}
   garbage:=Cut; {Vyjmi jej ze zásobníku}
   dispose(garbage); {zničobal}
   SInsert(pom); {vlož obal a uzel operátoru do zásobníku}
 end;
```

3.7 Optimalizace výrazu

Vytvořený syntaktický strom nám dává možnost pracovat algoritmicky se strukturou formule. Strom můžeme procházet principiálně různými způsoby podle toho, co je cílem algoritmu. Existují 4 základní možnosti průchodu pre-order, in-order, post-order a level-order. Anglické slovíčko „order" jasně říká, co se myslí průchodem - jde o pořadí v jakém pracuje s uzly. Na následujících schématech lze pozorovat v jakém pořadí budou danou metodou zpracovány uzly (číslo je pořadí). Pravidlo se vždy aplikuje rekurzivně (s mírnou výjimkou u Level-order) - tedy používá samo sebe na potomky uzlu. Na uzel aplikujeme vybranou operaci (třeba výpis znaku uzlu). Trochu výjimečná je procedura průchodu Level-order, kde musíme implementovat frontu s ještě nezpracovanými uzly a ty postupně obsluhovat.



Využití různých způsobu průchodu se dá ilustrovat na následujících příkladech: Výpis výrazu do infixní formy - použijeme In-order, protože chceme mít vždy binární spojky v pořadí operand operátor operand.

Vyhodnocení výrazu - použijeme Post-order, neboť výraz se vyhodnocuje "zdola", tj. než začneme vyhodnocovat spojku, musíme oba podvýrazy již mít vyčíslené

Odstranění negace směrem na proměnné - použijeme Pre-order, jedná se o často prováděnou operaci v logice, chceme aby negace byla jen u logických proměnných, musíme tedy negaci postupně shora od kořene tlačit směrem dolů pomocí logických zákonů jako jsou De-Morganovy zákony, např.

$$\neg(A \wedge B) \Leftrightarrow (\neg A \vee \neg B).$$

Podívejme se blíže na algoritmus pro výpis formule v infixním tvaru čitelném pro uživatele. Rekurzivní procedura typu In-order Dive provede zejména volání sama sebe pro levý podstrom pro logickou spojku, pak se vždy vypíše znak atomu a nakonec se zavolá Dive pro pravý podstrom u spojky. Samozřejmě je třeba vyřešit závorkování (infixní forma, na kterou jsou lidé zvyklí, se bez závorek neobejde narozdíl od postfixu a stromu). Detaily jsou popsány v komentářích zdrojového kódu.

```
procedure TEditForm.Dive; {vypisuje rekurzivně formuli v
infixním tvaru}
   begin
         if (node^.neg=true) then
           begin   {pokud je uzel negován musíme vložit znak
negace}
             InsertString('~');
             if node^.prior<>6 then
             {ale pokud to není atom jsou nutné závorky}
               begin InsertString('('); end;
           end;
         if (node^.prior <> 6) then
         {není-li atom budeme vypisovat
```

```
                        levý podstrom}
          if
(node^.prior>node^.left^.prior)and(not(node^.left^.neg)) then

          {pokud má operátor levého podvýrazu nižší prioritu
musíme
          vložit závorku, jinak by se provedl nejprve uzel,
který právě řešíme}

             begin
               InsertString('(');

               {volá se Dive pro levý podstrom}
               Dive(node^.left); InsertString(')');
             end else Dive(node^.left); {volání bez závorek}
         InsertString(node^.character);                {vložíme
znak spojky uzlu}
         if (node^.prior <> 6) then {není-li atom vypíšeme
pravý podstrom}
          if
((node^.prior>node^.right^.prior)or((node^.prior=2)

and(node^.right^.prior=2)))and(not(node^.right^.neg)) then

 {pokud má operátor pravého podvýrazu nižší prioritu musíme
vložit závorku, jinak
 by se provedl nejprve uzel, který právě řešíme Pozor! Pro
pravý podstrom je třeba
 závorka, pokud je uzel i jeho pravý podvýraz implikace - není
asociativní}

             begin
               InsertString('(');

               {volá se Dive pro pravý podstrom}
               Dive(node^.right); InsertString(')');
             end
           else Dive(node^.right); {volání bez závorek}
         if (node^.prior<>6)and(node^.neg=true) then
           begin
             InsertString(')');{ukončení závorky pro počáteční
negaci}
           end;
  end;
```

Ve výuce se logika na SŠ většinou vyučuje jen v rámci matematiky a to způsobem, který odpovídá jejímu využití v matematice - tedy především jako formální aparát pro vyjadřování matematických vztahů. Většinou se spokojíme s výukou sémantiky logických spojek, případně s pojmem tautologie, ale existují také jednoduché a čistě formální metody na zjednodušování formulí a případně i určování platnosti, splnitelnosti a ověřování správnosti dedukce. Důležité je, že daná pravidla jsou poměrně jednoduchá a pracují výlučně se syntaxí formule, kterou nyní máme zapsánu velmi vhodně formou stromu. Formální pravidla známe i z jiných oblastí, například z teorie množin, kde třeba známe a často používáme vztah mezi dvěmi množinami a operacemi rozdílu a průniku:

$$A - B = A \cap \overline{B}$$

Podobně lze upravovat nejen výrazy s rovnicemi, ale i logické výrazy. Nejprve se podívejme na vyhodnocení logických konstant.

Pro vyhodnocení logických konstant může využít následující zákony (ekvivalence):

$$A \wedge B \Leftrightarrow B \wedge A, A \vee B \Leftrightarrow B \vee A, A \leftrightarrow B \Leftrightarrow B \leftrightarrow A$$

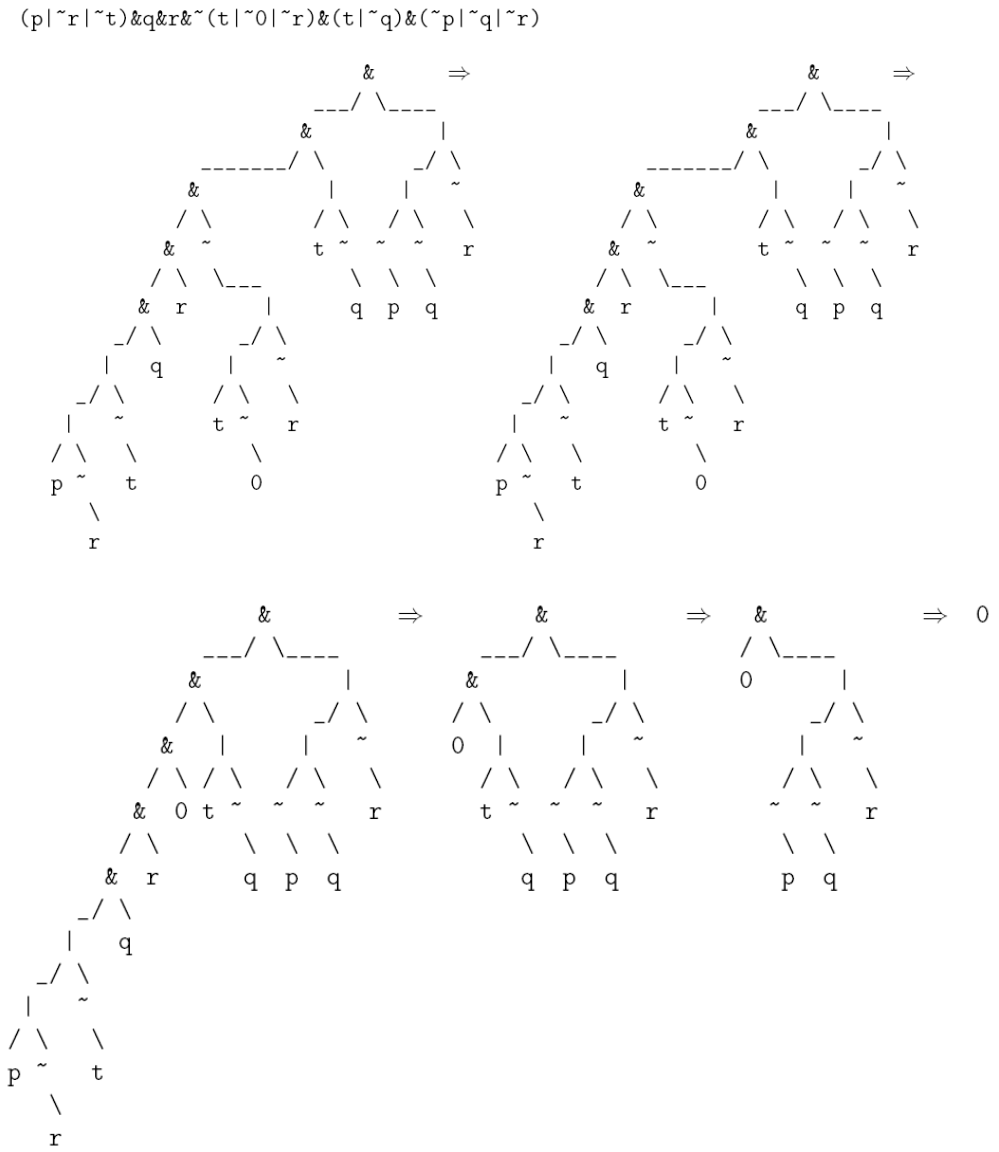
$$\neg 0 \Leftrightarrow 1, \neg 1 \Leftrightarrow 0, A \wedge 0 \Leftrightarrow 0, A \wedge 1 \Leftrightarrow A, A \vee 0 \Leftrightarrow A, A \vee 1 \Leftrightarrow 1, A \leftrightarrow 0 \Leftrightarrow \neg A, A \leftrightarrow 1 \Leftrightarrow A$$

$$A \rightarrow 1 \Leftrightarrow 1, A \rightarrow 0 \Leftrightarrow \neg A, 1 \rightarrow A \Leftrightarrow A, 0 \rightarrow A \Leftrightarrow 1$$

Tyto ekvivalence může zcela přirozeně naimplementovat do systému, který pracuje se syntaktickým stromem a pak už jen stačí napsat algoritmus, který vhodně prochází strom a aplikuje je. Vlastní procházení typu Post-order realizuje procedure RConstEval (uvádíme jen fragment) a každý uzel ja pak testován procedurou , zda se v něm díky logické konstantě nemůže zjednodušovat podle výše uvedených zákonů.

```
procedure TEditForm.RConstEval;
    begin
      if  (tr^.prior <> 6) then
        begin
          RConstEval(tr^.left);          RConstEval(tr^.right);
EvlConst1(tr);
        end; {levý podstrom, pravý podstrom, pak teprve uzel}
    end;
...
procedure TEditForm.EvlConst1;
 {pokus o vyhodnocení logické pravdy nebo nepravdy}
 ...
 begin
 ...
 case pchar of
      '=':begin
            if            pompt2^.character='0'            then
pompt1^.neg:=not(pompt1^.neg);
            p:=pompt1;RdisposeTree(pompt2);
          end;{ekvivalence se vyhodnotí přesně dle ekvivalencí}
                { ( A = 1 ) = A
                  ( A = 0 ) = ~A }
 ...
 end;
```

Podívejme se také na příklad takového zjednodušení, které může být velmi účinnou optimalizací a ušetřit nám zbytečné vyhodnocování velké části podmínky. Mějme poměrně složitou podmínku -logický výraz - který obsahuje jednu konstantu 0 (třeba bychom ji dostali po vyhodnocení jednoho porovnání v nějakém programovacím jazyce a nějakém programu v něm vytvořeném).



Další zajímavou transformací logického výrazu je převod na tzv. funkčně úplnou množinu spojek. V logice existují kombinace spojek, pomocí kterých můžeme vyjádřit všechny formule jako ekvivalentní. To by mohlo být zajímavé v případě, že stroj, který pro zpracování našich logických výrazů použijeme, umí velmi rychle oproti jiným spojkám, řešit nějakou konkrétní spojku (procesory počítačů i jiné logické obvody mohou tuto vlastnost mít). Druhou možností jsou například některé důkazové metody (například metoda "reductio ad absurdum", nepřímý důkaz, vyžaduje formule ve formě implikací). Funkčně úplné množiny spojek jsou například:

$$\{\neg, \longrightarrow\}, \{\longrightarrow\}$$

- negace a implikace, či dokonce pouze implikace (ovšem s pomocí logických konstant)!

$$\{\neg, \wedge, \vee\}, \{\neg, \wedge\}, \{\neg, \vee\}$$

- negace, konjunkce, disjunkce či dokonce negace pouze s jednou těchto spojek; u první množiny lze dostat tvar tzv. negační normální formy - kdy negace se vyskytuje pouze u výrokových proměnný (lze ji vytlačit směrem dolů ve stromě až na listy)

Při převodu výrazu na tyto množiny spojek můžeme využít například tyto zákony (přepis mezi implikací a disjunkcí a konjunkcí, odstranění ekvivalence, vytlačení negace - De Morganovy zákony):

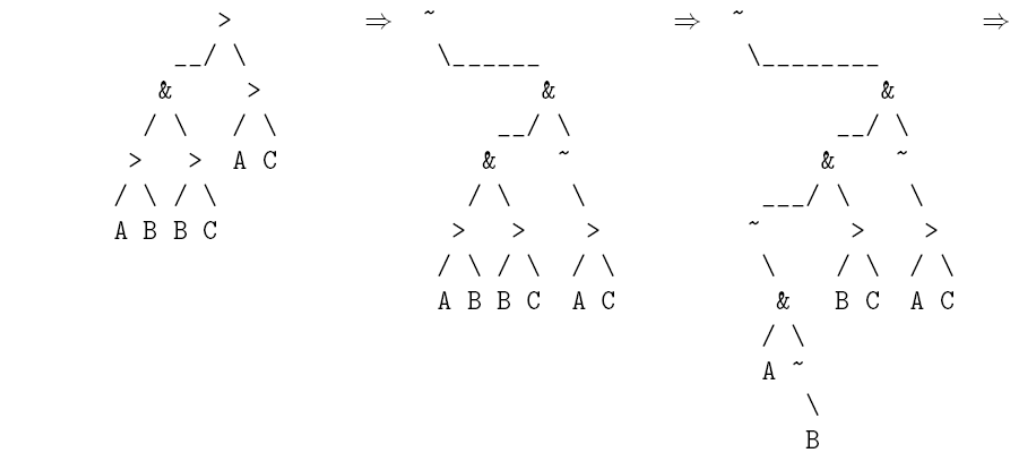
$$A \wedge B \Leftrightarrow \neg(A \rightarrow \neg B), A \vee B \Leftrightarrow \neg A \rightarrow B, A \leftrightarrow B \Leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A), \neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B, \neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B, \neg\neg A \Leftrightarrow A$$

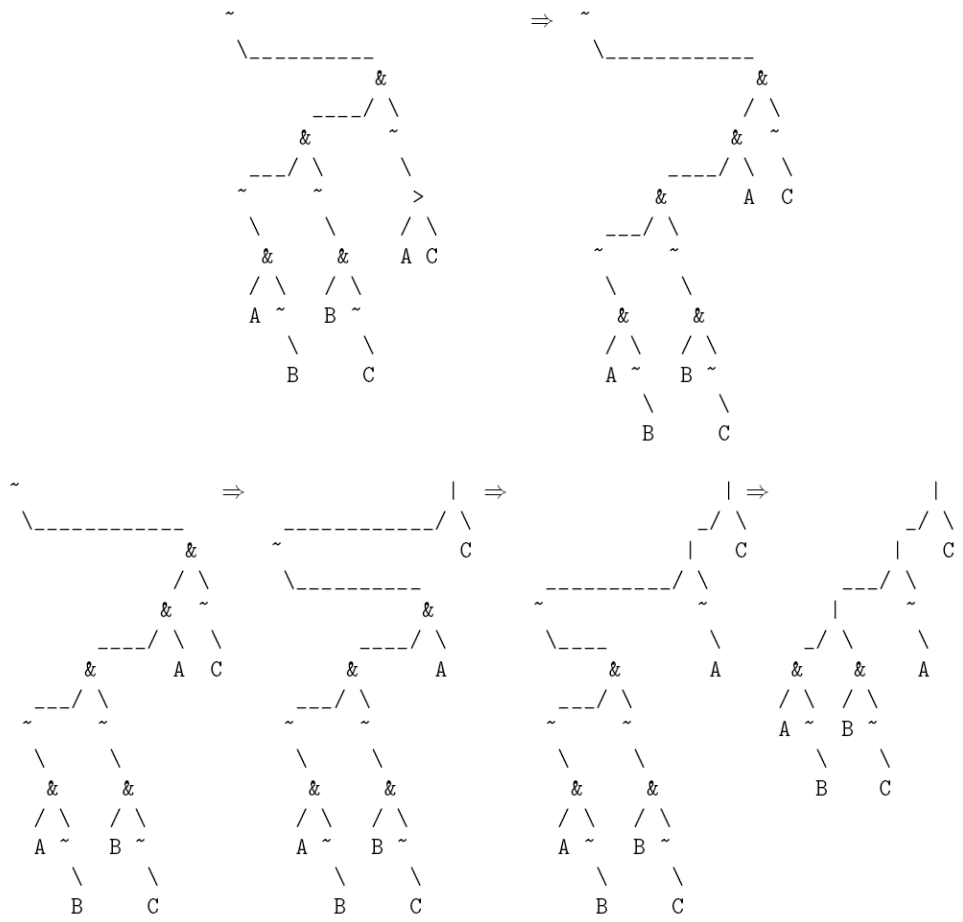
Tyto zákony lze implementovat průchodem typu Pre-order (nejprve se musí změnit spojka na vyšší úrovni a pak se může pokračovat dále na nižší). Podívejme se pro příklad na fragment algoritmu na převod na konjunkce a negace. Procedura Conjunction provede rekurzivně změny spojek a příslušné změny na negované formule dle ekvivalencí.

```
procedure TEditForm.Conjunction;{konverze na konjukce a negace}
...
     else if p^.character = '>' then
         begin {implikaci změníme na konjunkci}
           ChangeOper(p,pchar);p^.neg:=not(p^.neg);
           {pozor nová konjukce je negovaná}
           p^.right^.neg:=not(p^.right^.neg);
           {pozor pravý podstrom je také negovaný}
           ...
         end;
         { ( A > B ) =  ~ ( A & ~ B )  }
     end;
  end;
```

A nyní příklad, jak lze nejprve formuli převést pouze na konjunkce, negace a pak na konjunkce, disjunkce a negace, kde negace budou jen na proměnných. Výchozí formule je následující (vlastně jde o známé pravidlo tranzitivity):

```
(((A>B)&(B>C))>(A>C))
```





Nyní si oba převody ještě okomentujeme v infixní formě:

(((A>B)&(B>C))>(A>C))
Implikace na konjunkci:~((A>B)&(B>C)&~(A>C))
Implikace na konjunkci:~(~(A&~B)&(B>C)&~(A>C))
Implikace na konjunkci:~(~(A&~B)&~(B&~C)&~(A>C))
Implikace na konjunkci:~(~(A&~B)&~(B&~C)&A&~C)
~(~(A&~B)&~(B&~C)&A&~C)

Conjunction to disjunction:~(~(A&~B)&~(B&~C)&A)|C
Konjunkce na disjunkci:~(~(A&~B)&~(B&~C))|~A|C
Konjunkce na disjunkci:A&~B|B&~C|~A|C
A&~B|B&~C|~A|C

Nakonec se seznámíme s nejdůležitějším pojmem tzv. konjunktivní normální formy výrazu. Ten nám také umožní dokonce formálně prověřovat dedukci.

Konjunktivní normální forma (KNF) výrazu se skládá z tzv. konjunktů spojených spojkou konjunkce (konečný počet). Konjunkt je tvořen z výrokových proměnných bez nebo s negací spojených výhradně spojkou disjunkce. Tento tvar tedy z hlediska stromu obsahuje směrem od kořene dolů konjunkce a platí, že od první disjunkce se už směrem dolů vyskytují jen disjunkce nebo atomy s negací a bez.

Disjunktivní normální forma (DNF) výrazu je duální ke KNF a skládá se z tzv. disjunktů spojených spojkou disjunkce (konečný počet). Disjunkt je tvořen z výrokových proměnných bez nebo s negací spojených výhradně spojkou konjunkce. Tento tvar tedy z hlediska stromu obsahuje směrem od kořene dolů disjunkce a platí, že od první konjunkce se už směrem dolů vyskytují jen konjunkce nebo atomy s negací a bez. Příkladem DNF je vygenerovaná formule z minulého příkladu.

A&~B|B&~C|~A|C

KNF se používá především v oblasti automatizovaného dokazování při použití klauzulární rezoluční metody a DNF se dá využít pro zjištění platnosti formule (zda je tautologie) - DNF tautologie by po zjednodušení měla degradovat na 1 (pravdu). DNF je navíc velmi expresivní, pokud chceme určit, za jakých pravdivostních hodnot výchozích proměnných je výraz pravdivý. Druhá stránka věci je také pro praxi velice užitečná, neboť jednoduchými pravidly lze DNF i KNF minimalizovat, což může významně zjednodušit a zmenšit formuli (samozřejmě menší formule, znamená mnohem méně vyhodnocování při výpočtu výrazu). Nejprve si projděme pravidla pro přepis do normálních forem pomocí ekvivalencí (existují samozřejmě i sémantické metody, kdy z tabulky můžeme vytvořit tzv. úplné normální formy). Ekvivalence, kromě již výše použitých pravidel, zahrnují především dvě pravidla, která umožňují změnit hierarchii negace mezi spojkami konjunkce a disjunkce (jde o distributivní zákon), dále zákony absorpce, absorpce negace, idempotence a komplementarita, a nakonec rozšíření.

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

Pozn. V automaticky vyhodnocených příkladech se používá v algoritmech někdy pro jednodušší manipulaci (např. u zákona rozšíření) přiřazení logických konstant za A a B, tak aby výsledek po vyhodnocení odpovídal ekvivalenci.

Příklad 13.

Mějme formuli vyjadřující, že existuje k implikaci i její obrácená implikace, pak lze odvodit i ekvivalenci obou formulí, které jsou argumenty implikace.

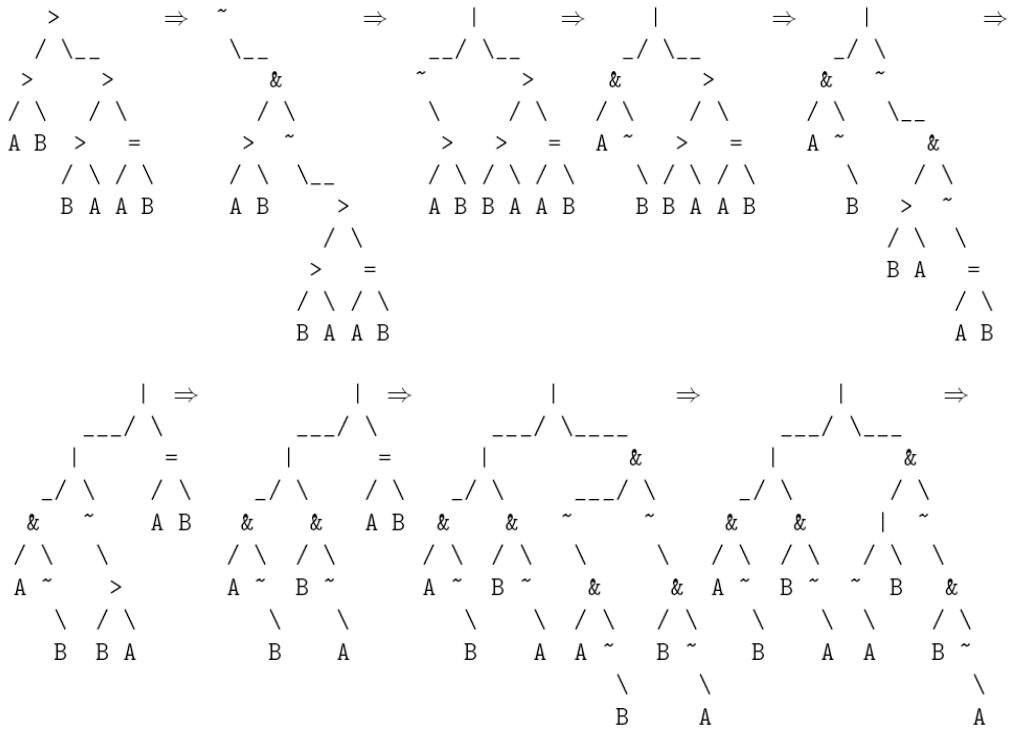
(A>B)>((B>A)>(A=B))

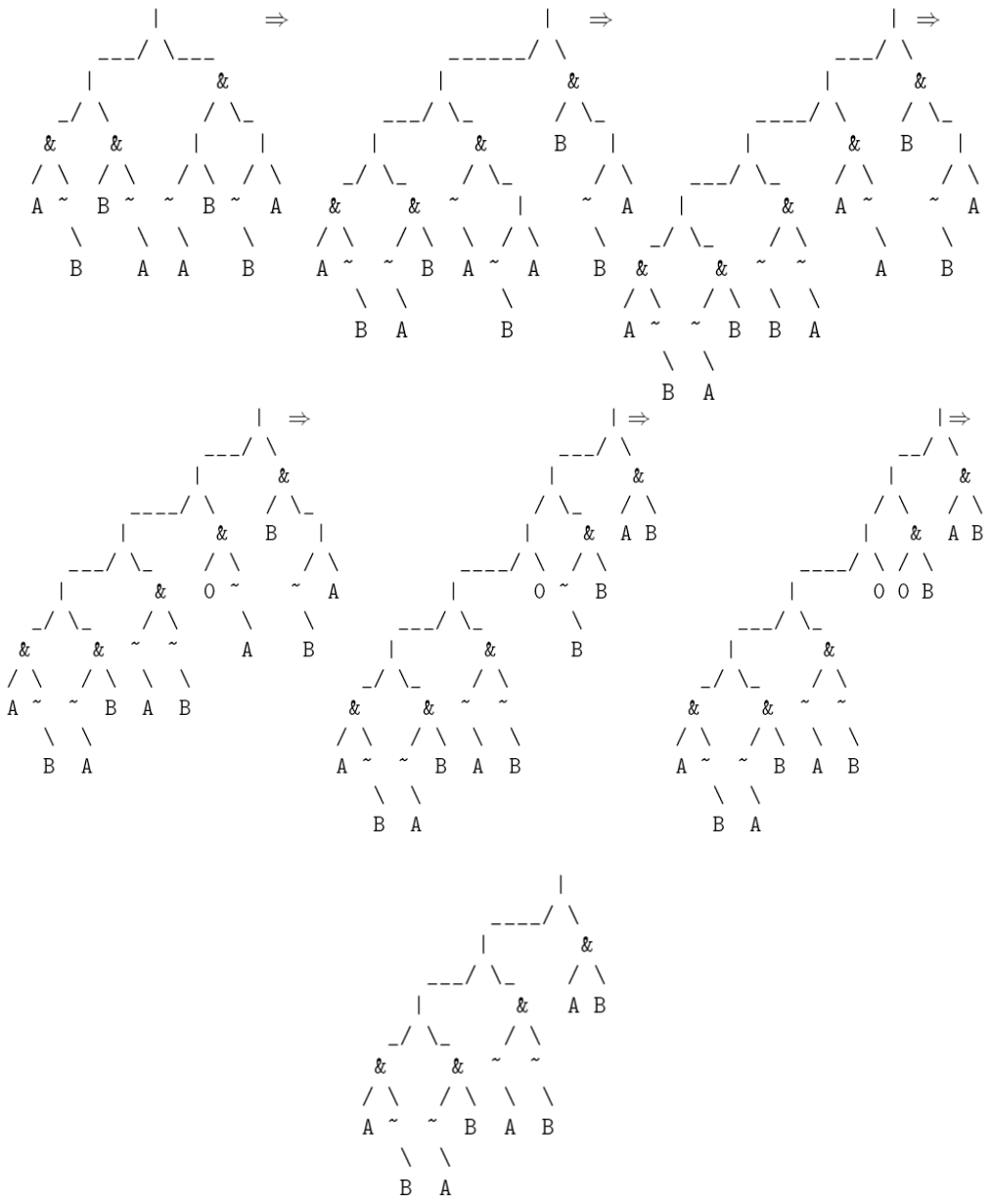
Nejprve se podívejme, které operace musíme udělat, abychom dospěli k DNF. Na této DNF je vidět, za jakých podmínek je platná. Obsahuje 4 disjunkty a můžeme z nich i vyčíst, při jakých hodnotách dosazených za logické proměnné bude výraz pravdivý. Disjunkt 1 říká, že výraz platí, pokud A platí (tedy A = 1) a zároveň platí ŘB (tedy B = 0), disjunkt 2 - A = 0, B = 1, disjunkt 3 - A = 0, B = 0, disjunkt 4 - A = 1, B = 1. V tomto případě jednoduchou úvahou vidíme, že v podstatě to jsou všechny možnosti, jaké hodnoty mohou být A a B dosazeny a tudíž formule je tautologie. Minimalizace nám však toto zajistí algoritmicky.

(A>B)>((B>A)>(A=B))
Implikace na konjunkci:~((A>B)&~(B>A>(A=B)))
Konjunkce na disjunkci:~(A>B)|(B>A>(A=B))
Implikace na konjunkci:A&~B|(B>A>(A=B))
Implikace na konjunkci:A&~B|~((B>A)&~(A=B))
Konjunkce na disjunkci:A&~B|~(B>A)|(A=B)
Implikace na konjunkci:A&~B|B&~A|(A=B)
Ekvivalence na konjunkci:A&~B|B&~A|~(A&~B)&~(B&~A)
Konjunkce na disjunkci:A&~B|B&~A|(~A|B)&~(B&~A)
Konjunkce na disjunkci:A&~B|B&~A|(~A|B)&(~B|A)
Distribuce:A&~B|~A&B|~A&(~B|A)|B&(~B|A)
Distribuce:A&~B|~A&B|~B&~A|A&~A|B&(~B|A)
Komplementarita:A&~B|~A&B|~A&~B|0&~A|B&(~B|A)
Distribuce:A&~B|~A&B|~A&~B|0|~B&B|A&B
Komplementarita:A&~B|~A&B|~A&~B|0|0&B|A&B

DNF: A&~B|~A&B|~A&~B|A&B

Nebo výstižněji pomocí stromu.





Nyní můžeme zkusit aplikovat pravidla pro minimalizaci formule a dostaneme logickou konstantu 1. Tudíž formule je opravdu logicky platná (tautologie - zákon). Takovou podmínku bychom bez problému mohli zcela vynechat a považovat ji za vždy splněnou.

A&~B|~A&B|~A&~B|A&B

Rozšíření:0&~B|~A&B|1&~B|A&B
Rozšíření:0&~B|0&B|1&~B|1&B
Rozšíření:0|1
Minimální DNF: 1

Příklad:

Na dalším příkladě můžeme vidět, jak drasticky se zjednoduší pomocí minimalizace formule - tedy ušetříme mnoho potenciálních vyhodnocení. Mějme následující formuli:

(~p>(q&~r))>(~q|r)

Nejprve ji po krocích převedeme na DNF.
Implikace na konjunkci:~((~p>q&~r)&~(~q|r))
Konjunkce na disjunkci:~(~p>q&~r)|~q|r
Implikace na konjunkci:~p&~(q&~r)|~q|r
Konjunkce na disjunkci:~p&(~q|r)|~q|r
Distribuce:~q&~p|r&~p|~q|r
DNF: ~p&~q|~p&r|~q|r

Minimalizací dostaneme překvapivě jednoduchou formuli a to pouze aplikací dvou absorpcí.
Absorpce:0&~q|~p&r|~q|r
Absorpce:0&~q|0&r|~q|r
Minimální DNF: ~q|r

Posledním použitím, které sice úzce nesouvisí s cílem našeho článku, je možnost minimalizací DNF zjišťovat, zda závěr vyplývá z daných předpokladů. Ukažme si to pouze na příkladu.

Příklad 14.

Nechť jsou daná tři tvrzení - předpoklady:
1. Jan je učitel.
2. Neplatí, že Jan je učitel a zároveň je bohatý.
3. Je-li Jan rockový zpěvák, pak je bohatý.
Chtěli bychom prověřit závěr - Z. Jan není rockový zpěvák.
Nejprve musíme zvolit logické proměnné.
u - Jan je učitel.
b - Jan je bohatý.
z - Jan je rockový zpěvák.
Nyní musíme sestrojit výrazy pomocí spojek pro tvrzení 1., 2., 3. a spojit je všechny konjunkcí (platí současně), tuto konjunkci pak dáme jako první podvýraz implikace a druhým bude závěr Z. (Implikace vyjadřuje, že závěr vyplývá z předpokladu). Formuli následně převedeme na minimální DNF a vyjde-li 1, pak se jedná o správnou dedukci (systém Bachelor toto přímo umožňuje).

u&~(u&b)&(z>b)>~z

Implikace na konjunkci:~(u&~(u&b)&(z>b)&z)
Konjunkce na disjunkci:~(u&~(u&b)&(z>b))|~z
Konjunkce na disjunkci:~(u&~(u&b))|~(z>b)|~z

Konjunkce na disjunkci:~u|u&b|~(z>b)|~z
Implikace na konjunkci:~u|u&b|z&~b|~z
Absorpce negace:~u|b&1|~b&z|~z
Absorpce negace:~u|b&1|~b&1|~z
Rozšíření:~u|0|1|~z
Výraz je platný. {Dedukce je správná}

Druhý zajímavý příklad umožňuje pomocí DNF zjistit, kdy dává sada výrazů smysl - v tomto případě tak chceme odhalit viníka trestného činu (předpokládáme, že mluví pravdu).

Příklad 15.

Brown, Jones a Smith jsou podezřelí z podvodu. Svědčili pod přísahou takto:
1. Brown: Jones je vinen a Smith je nevinen.
2. Jones: Je-li vinen Brown, pak je vinen i Smith.
3. Smith: Já jsem nevinen, ale alespoň jeden ze zbývajících obviněných je vinen.
Nejprve musíme zvolit logické proměnné.
u - Brown je vinen.
b - Jones je vinen.
z - Smith je vinen.
Nyní musíme sestrojit výrazy pomocí spojek pro tvrzení 1., 2., 3. a spojit je všechny konjunkcí (platí současně). Výraz následně převedeme na minimální DNF a disjunkty nám ukáží informace, kdy je konjunkce tvrzení pravdivá - dává smysl a také z nich přímo vyčteme, jaká musí vina a nevina jednotlivých obviněných.

(J&~S)&(B>S)&(~S&(J|B))
Implikace na konjunkci:J&~S&~(B&~S)&~S&(J|B)
Konjunkce na disjunkci:J&~S&(~B|S)&~S&(J|B)
Distribuce:~B&J&~S&~S&(J|B)|S&J&~S&~S&(J|B)
Distribuce:J&J&~S&~S&~B|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Idempotence:~B&1&J&~S&~S|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Idempotence:~B&1&J&1&~S|B&J&~S&~S&~B|S&J&~S&~S&(J|B)
Komplementarita:~B&1&1&J&~S|0&~B&J&~S&~S|S&J&~S&~S&(J|B)
Idempotence:~B&1&1&J&~S|0&~B&J&1&~S|S&J&~S&~S&(J|B)
Distribuce:~B&J&~S|0|J&J&~S&~S&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&J&~S&~S&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&J&1&~S&S|B&J&~S&~S&S
Komplementarita:~B&J&~S|0|1&J&1&0&S|B&J&~S&~S&S
Idempotence:~B&J&~S|0|1&0&1&J&S|B&J&1&~S&S
Komplementarita:~B&J&~S|0|1&0&1&J&S|B&J&1&0&S
Formule je konzistentní.Její modely vyjadřuje DNF:
~B&J&~S {Tedy vinen je Jones a ostatní jsou nevinní.}

- Význam dedukce pro praxi
- Formální vs. Sémantická dedukce

4 Strategie pro automatizovanou dedukci

V této kapitole se dozvíte:

- Strategie pro automatizovanou rezoluci (dedukci)
- Standardní strategie pro vyšší efektivitu
- Autorem navržené strategie

Po jejím prostudování byste měli být schopni:

- Naprogramovat jednoduché inferenční jádro postavené na standardních strategiích

Klíčová slova této kapitoly:

Symbolická logika, logická dedukce, formální dedukce

Doba potřebná ke studiu: 8 hodin

Průvodce studiem

Studium této kapitoly je poměrně zajímavé pro ty, kteří pochopili principy formální dedukce a její časové složitosti. V takovém případě Vám zřejmě některé příklady nebudou dělat problémy, avšak uvědomte si, že tyto strategie je nutné používat nejen teoreticky, ale i v konkrétní implementaci, což není zcela jednoduché.
Na studium této části si vyhraďte alespoň 8 hodin. Doporučujeme studovat s přestávkami vždy po pochopení jednotlivých podkapitol.

Jistě jste si v minulé kapitole všimli, že pracujeme jen s logikou výrokovou. Na této logice jsme si vysvětlili způsob výstavby deduktivního systému. Z výuky logiky je vám však jasné, že výroková logika je aparát natolik jednoduchý, že nás stěží může uspokojit při modelování úsudku. Z hlediska vyčíslitelnosti je dokonce výroková logika rozhodnutelná, což jste viděli rovněž v minulé kapitole v praxi. Pro automatizaci dedukce se však mnohem lépe hodí logika predikátová, resp. její vhodně zjednodušený fragment. I když to vypadá jednoduše vyměnit výrokovou logiku za predikátovou (PL), při automatizaci dedukce to má pro nás fatální důsledky. Logika predikátová totiž není obecně rozhodnutelná (pouze částečně rozhodnutelná). Navíc nás samozřejmě nezajímá jen vyčíslitelnost takové logiky, ale také jakou složitost budou mít algoritmy dedukce. Jako příklad může sloužit například tolik úspěšné logické programování. Jazyk PROLOG jako jeho typický zástupce nám ukazuje velmi efektivní automatizovanou dedukci – odpovědi na dotazy na znalostní bázi dostanete v řádu zlomků sekund. Ovšem to vše je zaplaceno například absencí existenční kvantifikace a další drastických omezení PL. Proto se nyní soustředíme na systém schopný dedukce i v logice predikátové. Zde je vzhledem k časové složitosti potřeba používat strategie pro hledání důkazu, které zajistí rovněž „rozumnou“ časovou složitost procesu.

4.1 Základní pravidla pro logickou dedukci

Jako nejúspěšnější pravidlo pro automatizovanou logickou dedukci se bezpochyby bere již zmíněný rezoluční princip. Můžeme se s ním setkat v různých podobách – např. v klauzulární logice je formulován jako pravidlo řezu. Přesto to je to jakási mutace rezolučního pravidla, podobně jako třeba známé pravidlo Modus-Ponens z Hilbertovských axiomatických systémů je také v podstatě pouze zjednodušenou rezolucí. Rezoluční princip slouží k dokázání splnitelnosti množiny formulí, ale také jak už bylo uvedeno, k automatickému dokazování vět. Rezoluce může být aplikována na formule ve výrokové i predikátové logice a to přímým nebo nepřímým postupem. Ve výrokové logice se můžeme setkat s rezoluční metodou, jako nástrojem pro dokázání splnitelnosti či nesplnitelnosti formulí, v konjunktivní normální formě, častěji se však používá pojem klauzulární formě. Klauzulí máme na mysli takovou formuli, která obsahuje pouze binární spojku disjunkce a atomy této formule jsou pozitivní či negativní literály výrokových proměnných.
Rezoluční odvozovací pravidlo má tvar:

$$\frac{(A \vee l) \wedge (B \vee \neg l)}{(A \vee B)}$$

Kde l a $\neg l$ jsou pozitivním a negativním literálem, z nichž pomocí rezolučního odvozovacího pravidla dokážeme vyvodit výslednou klauzuli vynecháním literálů, a tak dostaneme výslednou rezolventu.
Tento postup lze použít pro dokázání toho, co vyplývá z daných formulí a nazývá se logická dedukce. Pro důkaz splnitelnosti (nesplnitelnosti) formule můžeme použít nepřímý rezoluční důkaz, kdy popřeme závěr logického důsledku.
Např. Mějme formuli Z, která je logickým důsledkem dané množiny formulí, platí tedy že:

$$P_1, P_2, \ldots, P_n \models Z.$$

Tuto množinu převedeme na důkaz nesplnitelnosti formule:

$$P_1 \& P_2 \& \ldots \& P_n \& \neg Z.$$

Jestliže se budeme rozhodovat, mezi přímou či nepřímou metodu, bude jednodušší použít nepřímou metodu v případě, kdy máme stanoven závěr. Pokud nám po použití rezoluční metody vyjde prázdná klauzule, dokážeme tím splnitelnost celého logického důsledku.
Použití rezoluční metody v predikátové logice je složitější než v logice výrokové, jelikož zde může být nekonečně mnoho předpokladů a závěrů pro dokazování. Z těchto důvodů byly zavedeny syntaktické metody dedukce, mezi nejznámější a nejdůležitější patří ***Obecná rezoluční metoda***, která se stala základem pro logické programování a programovací jazyk PROLOG. Jde o zobecnění rezoluční metody výrokové logiky, aplikovatelné na formule v konjunktivní normální formě, která se, na rozdíl od formulí ve výrokové logice, nazývá Skolemovou klauzulární formou (viz. další text).
Zobecněné rezoluční pravidlo má tvar:

$$\frac{(A \vee l_1) \wedge (B \vee \neg l_2)}{(A \vee B)\sigma}$$

Kde σ je nejobecnější unifikací literálů l_1 a l_2 , tzn. $l_1\sigma = l_2\sigma$. Klauzule v čitateli se nazývají rodičovské klauzule, ze kterých dostaneme ve jmenovateli rezolventu těchto klauzulí. Unifikací nazveme nejobecnější substituci termů, pro složitější úkoly budeme využívat unifikační algoritmy, které nám unifikaci zjednoduší. Mezi nejznámější algoritmy patří ***Herbrandovo universum*** a ***Robinsonův unifikační algoritmus***, který zobecňuje rezoluční odvozovací pravidlo, tak jak je uvedeno výše.

Pomocí této metody můžeme řešit logickou dedukci či ověřování, zda je daná formule tautologií. Stejně jako u logiky výrokové používáme přímé i nepřímé postupy.

Automatické dokazování vět jsou procedury, které mohou být použity k ověření, zda daná formule *F* (cíl) je ***logickým důsledkem*** množiny formulí *N* (teorií). Popření dokazování teorémů se zabývá stejným problémem, chce ukázat, že daná množina $N \cup \{\neg F\}$ je v rozporu (je sporem). Sporná množina *N* může být stanovena buď sémantickým tablem, nebo formálním důkazem $\perp$ z *N*, kde důkazy jsou stopy deduktivních odvození definovaných pomocí kolekce odvozovacích pravidel. Pro naše účely jde o odvozovací pravidlo n+1 relace na obecné klauzule. Prvky takové relace jsou obvykle zapisovány takto:

$$\frac{C_1 \quad \ldots \; C_n}{C}$$

Výše uvedená relace se pak nazývá ***odvozením (inferencí)***. Klauzule $C_1, \ldots, C_n$ jsou premisy (předpoklady) a *C* nazveme ***závěrem*** daného odvození.

Odvozovací systém **Γ** je souborem odvozovacích pravidel. Máme-li **I** jako odvození nebo množinu odvození, označíme **C** (**I**) jako závěr nebo množinu závěrů. Můžeme říct, že jde o odvození z *N*, pokud jsou všechny premisy prvky z množiny *N*.

Odvození se nazývá ***platným***, pokud je závěr ***logickým důsledkem*** daných premis $C_1, \ldots, C_n \models C$.

Platnost je často minimálním požadavkem pro odvozovací systém. V našem systému popření je dostačující, když odvozování zachovává konzistenci neboli splnitelnost. Odvozovací systém **Γ** zachovává splnitelnost pro všechny množiny klauzulí *N*, množina $N \cup C$ (**Γ**(*N*)) je splnitelná pro kterékoliv splnitelné *N*. Platný odvozovací systém sice zachovává splnitelnost, obecně to obráceně ale neplatí. Budeme tedy uvažovat pouze odvozovací systém zachovávající splnitelnost.

Uspořádání premis při odvozování je důležité, rozlišujeme premisy hlavní, které redukují závěr v rámci jiných premis (vedlejších). Neurčíme-li jinak, je poslední premisa, v odvození, hlavní premisou a další premisy jsou považovány za vedlejší.

Důkaz klauzule *C* z množiny klauzulí (teorií) *N*, v odvozovacím systému **Γ**, je posloupnost klauzulí $C_1, \ldots, C_m$, takových že $C = C_m$ a každá klauzule C_i je buď prvkem z *N* nebo je závěrem odvození pomocí **Γ** z $N \cup \{C_1, \ldots, C_{i-1}\}$.

Klauzule v *N* nazýváme předpoklady (domněnkami). Zapisujeme N $|\text{-}_\Gamma$ *C* pokud existuje důkaz klauzule *C* z *N* pomocí odvozovacího systému **Γ**.
Je-li *C* ***kontradikcí***, znamená to, že ***popíráme*** *N*. Odvozovací systém **Γ** je ***popíratelně úplný***, pokud je popření pomocí **Γ** z jakékoliv ***nesplnitelné*** množiny klauzulí *N*.

Množinu klauzulí *N* nazveme ***nasycenou***, s ohledem na **Γ**, pokud závěr, jakéhokoliv odvození pomocí **Γ** z *N*, je prvkem N. Jestliže je odvozovací systém ***popíratelně úplný*** a množina *N* je nasycená s ohledem na **Γ**, pak *N* je buď ***splnitelná***, nebo obsahuje ***kontradikci***.

Následující definice rezoluce ***je platná***:

$$\frac{F\,[G] \quad F'[G]}{F'[G/\bot] \vee F'[G/\mathrm{T}]}$$

Jde o rezoluci na *G*, kde závěr odvození nazveme ***rezolventou*** dvou premis. *F* a *F´* jsou pozitivní a negativní předpoklady. *G* je vyřešenou podformulí. Rezoluce je ***platné*** odvození. Ve skutečnosti předpokládejme, že *I* je interpretace, ve které jsou oba předpoklady platné. V *I* je výroková formule *G* pravdivá či nepravdivá (True nebo False). Pokud je tedy G [¬G] true v I, pak je F[G/T] [F[G⊥]] a odtud získáme naši rezolventu.

Jak už bylo řečeno v úvodu, je rezoluce jednou z hlavních výpočetních metod logického programování, ale také obecnou metodou automatického dokazování vět, založenou J. A. Robinsonem roku 1965. Od následujícího odvozovacího pravidla se odvíjí další rezoluční metod.

Rezoluční odvozovací pravidlo má tvar:

$$\frac{(A \vee l) \wedge (B \vee \neg l)}{(A \vee B)}$$

Kde *l* a $\neg l$ jsou pozitivním a negativním literálem, z nichž pomocí rezolučního odvozovacího pravidla dokážeme vyvodit výslednou klauzuli vynecháním literálů, a tak dostaneme výslednou rezolventu.

Následující metody rezoluce pracují s běžnými klauzulemi, které mají volné proměnné. Taková klauzule je množinou literálů. Máme-li klauzuli *C* a atom *A* je v ní pozitivním literálem, zatímco atom ¬*A*, je v ní negativním literálem.

Binární Rezoluce

$$\frac{C \vee A \quad D \vee \neg A}{(C \vee D)\sigma}$$

kde σ je nejobecnější unifikací atomických formulí *A* a $\neg A$.

Pozitivní faktoring

$$\frac{C \vee A \vee B}{(C \vee A)\sigma}$$

kde σ je nejobecnější unifikací atomických formulí *A* a B.

4.2 Neklauzulární rezoluce v predikátové logice

Snaha o formulaci rezolučního pravidla aplikovatelného na zcela obecné formule predikátové logiky (prvního řádu) není příliš rozšířená. Asi nejstarším kvalitní prací v této oblasti je starší práce Murraye „Completely non-clausal theorem proving“ (Murray, 1982), avšak teprve práce Bachmaira a Ganzingera (Bachmair, 1997) přinesla jasně definovaný princip formálního dokazování založeného na neklauzulárním tvaru.
Rezoluce našla své uplatnění v mnoha aplikacích z oblasti umělé inteligence a logiky jakými jsou např. Prolog, expertní systémy založené na pravidlových systémech
a teoretických metodách axiomatického systému – pravidlo řezu v klauzulárním axiomatickém systémů. U všech těchto aplikací se využívá rezoluční pravidlo na formule, které jsou v klauzulární formě, jak ve výrokové, tak i v predikátové logice. Formule v klauzulární formě jsou s původními formulemi ekvivalentní, ale mají zcela jiný tvar. Existují však i přístupy, kde je možné použít rezoluci i na formule, které nejsou v klauzulární formě.

Pravidlo lze formulovat ve výrokové verzi následovně.
Neklauzulární rezoluce

$$\frac{F[G] \quad F'[G]}{F[G/\bot] \vee F'[G/\top]}$$

kde F a F′ jsou premisy (pozitivní a negativní), G je výskyt podformule v premisách (toto pravidlo tedy může dokonce pracovat s celými podformulemi a je-li G atom, jde o specifický případ). Výsledná rezolventa má výskyty G v pozitivní premise nahrazeny logickou konstantou false a v negativní premise konstantou true. Korektnost tohoto pravidla je lehce dokazatelná, stačí si uvědomit předpoklady – obě premisy jsou platné v interpretaci I a tudíž v této interpretaci I musí být platná i jejich disjunkce, kde v jedné nahradíme G konstantou false a v druhé true. Pohybujeme se totiž ve dvouhodnotové logice, kde neexistuje jiná možnost interpretace G než pravda a nepravda. V disjunkci bude tudíž alespoň jedna její část opět interpretována jako true.

Premise1	Premise2	Resolvent	Simplified	Comments
$a \vee b$	$b \vee c$	$(a \vee \bot) \vee (\top \vee c)$	$\top$	no complemental pair
$a \vee \neg b$	$b \vee c$	$(a \vee \top) \vee (\top \vee c)$	$\top$	wrong selection of premises
$a \vee b$	$\neg b \vee c$	$(a \vee \bot) \vee (\bot \vee c)$	$a \vee c$	right clausal resolution
$a \vee \neg b$	$\neg b \vee c$	$(a \vee \top) \vee (\bot \vee c)$	$\top$	no complemental pair

Obrázek 12: Klauzulární rezoluce v kontextu neklazulární

Zajímavý pohled nám pak dává obrázek, který ukazuje obecnost neklauzulární rezoluce vůči rezoluci klauzulární. Také to evokuje další problémy, které při klauzulární rezoluci nemáme - to je správný výběr pozitivní a negativní premisy (jejich pořadí). Pokud zvolíme nesprávné pořadí premis nedostaneme

sice sémanticky nekorektní rezolventu, ale může jít o rezolventou „nadbytečnou“ – redundantní. To například ve smyslu nepřímého dokazování (rezolučního popření) znamená, že vygenerujeme tautologii. Ta samozřejmě pro další průběh rezolučního popření není produktivní. U klauzulární rezoluce to je triviální problém, protože rezoluci lze aplikovat pouze na navzájem negativní pár literálů. Tento problém pak řeší **strukturální analýza formule** pomocí pojmu takzvané **polarity podformule**. Polarita v podstatě u atomů simuluje jejich pozitivitu resp. negativitu ve smyslu negovaného atomu, pokud bychom provedli převod do klauzulární normální formy. Samozřejmě není potřeba tento převod vůbec realizovat – pokud máme k dispozici syntaktický formační strom formule je výpočet polarity z hlediska časové složitosti logaritmický ve vztahu k velikosti formule (při implementacích se navíc polarita vypočítává už při konstrukci stromu, který musíme mít stejně k dispozici). Výpočet polarity lze formálně realizovat rekurzivním výpočtem – tedy například máme-li implikaci dvou formulí a tato implikace je pozitivní, pak antecedent (jako celá podformule) je negativní a celý konsekvent je pozitivní.

Strukturální analýza formule je poměrně jednoduchá a to i z hlediska časové složitosti., jak již bylo uvedeno. U klauzulární rezoluce je problém polarity literálu bezpředmětný – podle toho zda je atom negovaný resp. není negován, pak je z tohoto pohledu negativní resp. pozitivní. Jelikož ale pracujeme s obecnými formulemi, musíme vzít v úvahu, jak logické spojky mění polaritu literálu. To lze docílit pomocí následující definice (podrobnosti v práci A).

Definice 1. Strukturální pojmy FOL formule
Nechť F je formule FOL pak Sub (subformule), Sup (superformule), Pol (polarita) a Lev (stupeň) jsou definovány následovně:

$F = G \wedge H$ nebo $F = G \vee H$	Sub(F) = {G,H}, Sup(G) = F, Sup(H) = F Pol(G) = Pol(F), Pol(H) = Pol(F)
$F = G \rightarrow H$	Sub(F) = {G,H}, Sup(G) = F, Sup(H) = F Pol(G) = -Pol(F), Pol(H) = Pol(F)
$F = \neg G$	Sub(F) = {G}, Sup(G) = F Pol(G) = -Pol(F)
$F = \exists \alpha G$ nebo $F = \forall \alpha G$ (α je proměnná)	Sub(F) = {G}, Sup(G) = F Pol(G) = Pol(F)

$Sup(F) = \emptyset \Rightarrow Lev(F) = 0, Pol(F) = 1$
$Sup(F) \neq \emptyset \Rightarrow Lev(F) = Lev(Sup(F)) |+ 1$
Pro Sub a Sup jsou definována reflexivní a tranzitivní uzavření Sub* a Sup* rekurzivně takto:

1. $Sub^*(F) \supseteq \{F\}$, $Sup^*(F) \supseteq \{F\}$

2. $Sub^*(F) \supseteq \{H | G \in Sub^*(F) \wedge H \in Sub(G)\}$,
   $Sup^*(F) \supseteq \{H | G \in Sup^*(F) \wedge H \in Sup(G)\}$

Tato strukturální mapování poskytují rámec pro přiřazování kvantifikátorů výskytům proměnné. To je potřeba pro korektní simulaci skolemizace (informace o kvantifikaci proměnné v prenexní formě). Mapování subformulí a superformulí a jejich uzavření zapouzdřují základní hierarchickou informaci o struktuře formule. Stupeň dává uspořádání vzhledem k oblasti platnosti proměnné (což je také podstatné pro simulaci skolemizace – unifikace je vyhrazená pro existenciální proměnné). Polarita dovoluje rozhodnout globální význam proměnné (např. Globálně je existenciální proměnná univerzální jestliže její kvantifikační subformule má negativní polaritu). Platná unifikace potřebuje další definice kvantifikace proměnných. Uvedeme pojmy souvisejícího kvantifikátoru pro výskyt proměnných, mapování substituce a mapování důležitosti (musíme rozhodnout mezi původními výskyty proměnných ve speciálních axiomech a nově uvedenými v průběhu dokazování).

Definice 2. ***Přiřazování proměnných, substituce a důležitost***

Nechť F je formule FOL, $G = p(t_1,\ldots,t_n) \in Sub^*(F)$ atom z F a α je proměnná vyskytující se v t_i. Mapování proměnných Qnt(přiřazování kvantifikátorů), Sbt(substituce proměnných) a Sig(význam) jsou definovány následovně:

$Qnt(\alpha) = Q\alpha H$, kde $Q = \exists \vee Q = \forall$, H, $I \in Sub^*(F)$, $Q\alpha H \in Sup^*(G)$, $\forall Q\alpha I \in Sup^*(G) \Rightarrow Lev(Q\alpha I) < Lev(Q\alpha H)$.

$F[\alpha / t']$ je substituce termu t' za α v $F \Rightarrow Sbt(\alpha) = t'$.

Proměnná α vyskytující se v $F \in LAx \cup SAx$ je významná *w.r.t.* existenční substituce, $Sig(\alpha) = 1$ jestliže proměnná je významná, $Sig(\alpha) = 0$ jinak.
Všimněte si, že *Qnt* mapování provádí přiřazení první proměnné v kvantifikátoru v hierarchii (od atomů), která odpovídá svým symbolickým jménem. Jsme schopni rozhodnout mezi proměnnými stejného jména a není potřeba přejmenovat žádnou proměnnou. *Sbt* mapování drží substituované termy v kvantifikátoru a není nutné přepisovat všechny výskyty proměnné, když se pracuje s tímto mapováním s unifikací. Je také jasné, že jestliže *Qnt*(α) = Ø pak α je volná proměnná. Tyto proměnné mohou být jednoduše vynechány vložením nových univerzálních kvantifikátorů do F. Významové mapování je důležité pro rozlišování mezi původními univerzálními proměnnými formule a mezi nově zavedenými během hledání důkazu (existenční proměnné s ním nemohou být spojeny).

Předtím, než může být uveden standardní unifikační algoritmus, bychom měli formulovat pojem globální univerzální a globální existenciální proměnné (simuluje konverzi do prenexní normální formy).

Definice 3. Globální kvantifikace

Nechť F je formule bez volných proměnných a α je výskyt proměnné v termu z F.

1. α je globální univerzální proměnná $(\alpha \in Var_{\forall}(F))$ jestliže $(Qnt(\alpha) = \forall \alpha H \wedge Pol(Qnt(\alpha)) = 1)$ nebo $(Qnt(\alpha) = \exists \alpha H \wedge Pol(Qnt(\alpha)) = -1$

2. α je globální existenciální proměnná $(\alpha \in Var_{\forall}(F))$ jestliže $(Qnt(\alpha) = \exists \alpha H \wedge Pol(Qnt(\alpha)) = 1)$ nebo $(Qnt(\alpha) = \forall \alpha H \wedge Pol(Qnt(\alpha)) = -1$

$Var_{\forall}(F)$ a $Var_{\exists}(F)$ jsou množiny globálních univerzálních a existenciálních proměnných.

Je jasné, s respektováním postupu skolemizace, že existenciální proměnná může být substituována za univerzální pouze když všechny globální univerzální proměnné nad rámec existenciální byly již substituovány termem. Funkce skolemových funktorů zrovna tak. To znamená, že substituce existenciální proměnné na univerzální produkuje logický důsledek formule.Nyní je možno definovat nejobecnější unifikační algoritmus založený na rekurzivním principu (rozšiřuje unifikaci v kontrastu k standardu MGU).

Definice 4. ***Nejobecnější unifikační algoritmus***

Nechť $G = p(t_1,\ldots,t_n)$ a $G' = r(u_1,\ldots,u_n)$ jsou atomy. Nejobecnější unifikátor (substituční mapování) $MGU(G,G') = \sigma$ je získán následujícími kroky unifikace atomů a termů, nebo algoritmus vrací chybový stav pro unifikaci. Pro účely algoritmu definujeme následující omezení unifikace proměnných (VUR).

Omezení unifikace proměnných

Nechť F_1 je formule a α je proměnná vyskytující se v F_1, F_2 je formule, t je term vyskytující se v F_2 a β je proměnná vyskytující se v F_2. Omezení unifikace proměnných (VUR Variable Unification Restriction) pro (α,t) platí, jestliže platí jeden ze stavů:

1. α je globální univerzální proměnná a t ≠ β, kde β je globální existenciální proměnná a α se nevyskytuje v t (ne-existenční substituce)

2. α je globální univerzální proměnná a t = β, kde β je globální existenciální proměnná a $\forall F \in Sup^*(Qnt(\beta))$, $F = Q\gamma G$, $Q \in \{\forall, \exists\}$, γ je globální univerzální proměnná, $\mathrm{Sig}(\gamma) = 1 \Rightarrow (Sbt(\gamma) = r') \in \sigma$ r' je term (existenční substituce).

Atomická unifikace

1. Jestliže n = 0 a p = r pak $\sigma = \emptyset$ a unifikátor existuje (úspěšný stav).

2. jestliže n > 0 a p = r pak se provede unifikace termů pro páry (t_1, u_1),…,(t_n, u_n); Jestliže unifikátor existuje pro každý pár, pak MGU(G,G') = σ je získáno během unifikace termů (úspěšný stav).

3. V každém jiném případě unifikátor neexistuje (neúspěšný stav)

Unifikace termů (t',u')

1. jestliže u' = α, t' = β jsou proměnné a Qnt(α) = Qnt(β) pak unifikátor existuje pro (t',u') (úspěšný stav) (výskyt totožné proměné).

2. jestliže t' = α je proměnná a $(Sbt(\alpha) = v') \in \sigma$ pak se použije unifikace termů pro (t',v'); Unifikátor pro (t',u') existuje jestliže existuje pro (v',u') (úspěšný stav pro už substituované proměnné).

3. jestliže u' = α je proměnná a $(Sbt(\alpha) = v') \in \sigma$ pak se použije unifikace termů pro (t',v'); Unifikátor pro (t',u') existuje jestliže existuje pro (t',v') (úspěšný stav pro už substituované proměnné).

4. Jestliže t' = a, u' = b jsou individuové konstanty a a = b pak pro (t',u') unifikátor existuje (úspěšný stav)

5. jestliže t' = f(t_1',…, t_m'), u' = g(u_1',…,u_n') jsou funkční symboly s argumenty a f = g pak unifikátor pro (t',u') existuje jestliže existuje pro každý pár (t_1',u_1'),…, (t_n',u_n') (úspěšný stav).

6. jestliže platí, že t' = α je proměnná a VUR pro (t',u'), pak unifikátor existuje pro (t',u') a $\sigma = \sigma \cup (Sbt(\alpha) = u')$ (úspěšný stav).

7. jestliže platí, že u' = α je proměnná a VUR pro (u',t'), pak unifikátor existuje pro (t',u') a $\sigma = \sigma \cup (Sbt(\alpha) = t')$ (úspěšný stav).

8. Ve všech ostatních případech unifikátor neexistuje (neúspěšný stav)

MGU(A) = σ pro množinu atomů A = {G_1, G_k} je vypočtený atomární unifikací pro (G_1, G_i), σ_i = MGU(G_1, G_i), $\forall i$, $\sigma_i = \emptyset$, kde před každou atomární unifikací (G_1, G_i), σ je množina pro σ_{i-1}.

S dříve definovanými pojmy je jednoduché uvést obecné rezoluční pravidlo pro FOL (bez spojky ekvivalence).

Příklad použití unifikace:

Můžeme se pokusit unifikovat a rezolvovat následující formule:

∀X∃Y p(X, Y) ├ ∃Y∀X p(X,Y)

Budeme rezolvovat nepřímým postupem, tudíž popřeme závěr a přidáme ho k předpokladu a pokusíme se najít popření (prázdnou formuli, false):

F0 : ∀X∃Y p(X, Y).

F1 (negovaný závěr) : ¬∃Y∀X p(X,Y)

Existují dvě triviální a dvě netriviální možnosti kombinace formulí F0 a F1. Samozřejmě můžeme vyloučit 2 triviální kombinace, které povedou na true, což je pro důkaz popřením bezpředmětné. Další dvě kombinace jsou zajímavější:

[F0 & F1] :není možné rezolvovat, unifikátor nedovolí, neboť dosazovaná existenční proměnná má nadřazenou univerzální X a ta zatím není ničím substituována. Jde o správný výsledek, existence specifického Y pro každý prvek univerza nezaručuje naplnění závěru – musí existovat jeden objekt Y stejný pro každý prvek dosazený za X.

Můžeme se ale pokusit vyzkoušet správnou funkci unifikátoru na duálním případě: ∃Y∀X p(X, Y) ├ ∀X∃Y p(X, Y).

F0: ∃Y∀X p(X, Y)

F1: ¬∀X∃Y p(X, Y)

Pro tento případ najdeme lehce popření, neboť tyto výrazy jsou unifikovatelné. Nejprve je možné unifikovat za univerzální proměnnou X v F0, existenční X

z F1 (na ničem nezávisí, tudíž unifikátor vrací danou substituci a dále opačně lze za univerzální Y v F1 ;berme v úvahu, že negace u F1 obrací globální význam a končí úspěchem).

Definice 5. ***Obecná rezoluce pro logiku prvního řádu*** **(GR_{FOL})**

$$\frac{F[G_1,\ldots,G_k]F'[G'_1,\ldots,G'_n]}{F\sigma[G/\perp]\vee F'\sigma[G/T]}$$

Kde $\sigma = MGU(A)$ je nejobecnější unifikátor (MGU) z množiny atomů $A=\{G_1,\ldots,G_k,G'_1,\ldots,G'_n\}$, $G = G_1\sigma$. Pro každou proměnnou α v F nebo F', $(Sbt(\gamma)=\alpha)\cap\sigma=\emptyset\Rightarrow Sig(\alpha)=1$ v F nebo F' jestliže $Sig(\alpha)=1$ v $F\sigma[G/\perp]\vee F'\sigma[G/\mathrm{T}]$. F se nazývá pozitivní a F' se nazývá negativní premisa, G reprezentuje výskyt atomu. Výraz $F\sigma[G/\perp]\vee F'\sigma[G/\mathrm{T}]$ je rezolventa premis G.

Všimněte si, že s Qnt mapováním je možné rozeznávat proměnné ne jen podle jejich jména (které nemusí být unikátní), ale také pomocí mapování (které unikátní je). Sig vlastnost umožňuje oddělit proměnné, které původně nebyly v rozsahu existenční proměnné. Při použití pravidla by mělo být nastaveno Sig mapování pro každou proměnnou v axiomech a negovaný cíl na 1.

Řešení je popsáno nejprve na intuitivní úrovni a dále bylo vylepšeno na formální úroveň zavedením několika pojmů, které na sebe vzájemně navazují. Zjednodušeně řečeno rozšíříme unifikaci termů o možnost unifikovat nejen (univerzální) proměnnou s termem, ale jelikož zachováváme všechny kvantifikátory, zavedeme i možnost unifikace univerzální proměnné s univerzální proměnnou (což odpovídá klasickému unifikátoru) a pak možnost unifikace univerzální proměnné s existenční proměnnou, ovšem to jen za splnění podmínek, které bychom jinak očekávali i u příslušného skolemova funktoru (pokud by skolemizace proběhla). Tou podmínkou je, že všechny univerzální proměnné v jejichž rozsahu je daná existenční proměnná již mají přiřazený term (tím by vlastně skolemův funktor degradoval na element reprezentující konstantní objekt). Samozřejmě z povahy neklauzulární rezoluční metody je potřeba brát také v úvahu, že proměnná se stejným symbolickým jménem se může vyskytovat ve formuli vícekrát, přičemž nemusí jít o sémanticky stejnou proměnnou (záleží na použití kvantifikátorů pro tyto proměnné – význam proměnné může být „zastíněn“ v lokálním výskytu symbolicky stejné proměnné). To lze jednoduše vyřešit přiřazením nejbližšího nadřazeného výskytu (z hlediska struktury formule) kvantifikátoru pro proměnnou s daným jménem pro každý výskyt proměnné v termech. Volné proměnné se eliminují zavedením univerzálního kvantifikátoru na nejvyšší úrovni formule.
Nyní okomentujme jednotlivé notace vedoucí k unifikátoru. Prvním krokem je zavedení strukturálních pojmů na formuli. Jde o klasická zobrazení umožňující definovat formační strom formule – Sub (Podformule), Sup (Nadformule), definované pro podformuli a vracející v případě Sub množinu podformulí o kardinalitě n odpovídající n-aritě logické spojky. Sup pro podfurmuli A vrací vždy jednu podformuli B, která má vlastnost, že A náleží do Sub(B). Zobrazení

Pol přiřazuje podformuli její polaritu (hodnota 1 nebo -1) ve smyslu, který jsme diskutovali v kapitole o neklauzulární rezoluci ve výrokové logice. Dalším potřebným zobrazením je Lev (úroveň), která dává možnost uspořádat podformule podle úrovně jejich vnoření (tj. samotná formule F má Lev(F)=0, její přímé podformule mají 1, atd… Dalším nutným zobrazením jsou také reflexivní a tranzitivní uzávěry množin Sub a Sup (potřebujeme například znát všechny kvantifikátory nad úrovní podformule a pak je pohodlné aplikovat uzávěr Sup s omezující podmínkou). Další definice se vztahuje k proměnným. Nejdůležitější je zobrazení Qnt, které přiřazuje výskytu proměnné v termu jeho kvantifikátor – tedy mezi prvky uzávěru množiny Sup jde o kvantifikátor se stejným jménem proměnné a největší hodnotou funkce Lev. Jinak řečeno jde o nejbližší kvantifikátor pro tuto proměnnou, pokud uvažujeme průchod formačním stromem od listů ke kořeni. Další pomocné definice zavádějí zobrazení substituce termu za proměnnou – Sbt (jde pouze uniformizaci zápisů do tvaru ostatních zobrazení) a zobrazení Sig rozlišuje významné a nevýznamné proměnné ve smyslu jejich přítomnosti ve formuli – axiomu (během odvozování mohou vznikat nové proměnné při vytvoření rezolventy s unifikací) a na těchto proměnných pak nesmí substituce existenční proměnné do univerzální záviset. Definice zavádí klíčový pojem tzv. globálních univerzálních a globálních existenčních proměnných. Pokud se chceme vyhnout skolemizaci je potřeba nasimulovat proces, který by jinak provedl převod do prenexní formy. Během tohoto převodu by se všechny kvantifikátory dostaly před otevřené jádro, přičemž by mohly změnit svůj význam mezi univerzalitou a existencí (na základě přepisovacích pravidel to umožňuje například negace podformule). A právě konečný význam kvantifikátoru před otevřeným jádrem (tedy „globální“ význam v rámci celé formule) potřebujeme znát. K řešení tohoto problému lze opět velice jednoduše použít polaritu podformule, která určuje, zda je význam původní nebo duální v případě, že je polarita kvantifikační podformule negativní. Definice dále přináší vlastní unifikační algoritmus (poměrně rozsáhlý), který zahrnuje rovněž definici Variable Unification Restriction. Specifikuje omezení na unifikaci proměnné, jak jsme ho již diskutovali výše. To znamená, že unifikovat může pouze proměnná globální univerzální proměnná a to buď s objektem, který není globální existenční proměnnou nebo je globální existenční proměnnou, ale všechny její nadřazené globální univerzální proměnné jsou substituovány termem (s výjimkou nevýznamných proměnných).

Dále již následuje specifikace algoritmu pro unifikaci atomů a termů. Oproti standardním algoritmům jsou přidány kroky, které umožňují unifikaci totožných proměnných (nikoliv proměnných se stejným symbolickým názvem, protože těch může být v jedné formuli více). Nakonec je rovněž definován nejobecnější unifikátor pro množinu atomů. V práci C jsou tyto definice uvedeny na intuitivní úrovni.

Na základě unifikátoru je pak možno definovat samotné pravidlo neklauzulární rezoluce pro predikátovou logiku. Jde o systém nepřímého dokazování, tedy formálně prokazujeme nesplnitelnost množiny axiomů a negace dotazu. Je zde potřeba rozlišit již diskutované významné proměnné (tedy ty které jsou původně obsaženy v axiomech a dotazu) od nevýznamných.

4.3 Neklauzulární rezoluce pro fuzzy logiku

Neklauzulární rezoluční metoda se může zdát v tradiční logice poněkud redundantním formalismem. Existuje klauzulární rezoluce s mnoha různorodými efektivními strategiemi, které značně zvyšují efektivitu inference. I v samotné tradiční logice najdeme speciální případy kdy neklauzulární rezoluce má prokazatelně výhody – např. u složitých formulí s mnoha úrovněmi vnoření podformulí (neklauzulární rezoluce u nich může rychle zcela eliminovat ty části formule, které by jinak v několika klauzulích zatěžovaly systém rezolučního popření). Zcela principiální význam má však neklauzulární rezoluční technika pro fuzzy (vícehodnotové) logiky. Existuje samozřejmě velké množství variant, jak fuzzy logiky definovat. Pokud se uvažuje klasický model, kdy formule je interpretována prvkem některý ze zavedených algeber založených na t-normách, pak je v podstatě možný výběr ze tří možností definice chování fuzzy logiky. Je možno použít Gődelovu, Goguenovu nebo Lukasiewiczovu algebru. S ohledem na možnost aplikovat navržený unifikační algoritmus pro tradiční (crisp) termy a navíc možnost používat zákon dvojí negace volba logicky padá na poslední zmiňovanou Lukasiewiczovu algebru.
Ve fuzzy logice (logikách) je problém rezolučního dokazování nesrovnatelně složitější než v logice tradiční. Problém u tradiční klauzulární rezoluce spočívá především v nemožnosti obecně vygenerovat klauzulární formu formule, a to ani ve fuzzy logikách výrokových. V některých logikách, například ve fuzzy predikátové logice (Lukasiewiczově) s ohodnocenou syntaxí ($Ev_L\forall$) (Novák, 1999), platí alespoň logická pravidla vedoucí obecně k prenexní formě ohodnocené formule. Proto byla právě tato fuzzy logika vybrána pro aplikaci neklauzulárního systému rezolučního popření. Logika $Ev_L\forall$ pracuje s velmi praktickým pojmem (syntaktického) ohodnocení formule. Každá formule může tedy v rámci důkazu mít syntaktickou hodnotu, která pak u vyvozené formule reprezentuje hodnotu důkazu. Propracovaná teorie pak dává univerzální nástroje pro práci s odvozovacími pravidly a důkazy jejich korektnosti, které lze provést mnohem jednodušeji než při nutnosti konstruovat celý důkaz pro dané pravidlo zcela od počátku. Díky této flexibilitě se důkaz redukuje na důkaz platnosti nerovnice dle citované monografie.
První pokus o aplikaci neklauzulární rezoluce do fuzzy výrokové logiky s ohodnocenou syntaxí byl publikován v roce 2002. Šlo prozatím o přístup nepracující s unifikací a také umožňoval pouze přímé důkazy. Jeho myšlenka vychází z analogie s fuzzy pravidlem Modus-ponens. Odvozovací pravidlo pracuje obecně s několika premisami a ty vedou k výsledné formuli. Pravidlo má svou syntaktickou a sémantickou část – syntaktickou je ona výsledná formule, ale navíc je potřeba přiřadit nové ohodnocení této formuli, což zajišťuje operace r^{evl}. Klíčovým je pojem ohodnoceného důkazu, kde je ke klasické posloupnosti navíc u každého prvku (formule) jeho ohodnocení. Ohodnocení poslední formule v důkazu je pak hodnotou důkazu. Pojem pravdivosti a dokazatelnosti se od tradičních pojmů liší v tom, že formule není prostě pravdivá a dokazatelná, ale může být pravdivá nebo dokazatelná ve stupni a (kde a je prvek množiny pravdivostních hodnot Lukasiewiczovy algebry, resp. jeho syntaktický protějšek). Pravidlo fuzzy Modus-Ponens přiřazuje výsledné formuli ze dvou premis novou hodnotu operací Lukasiewiczovy konjunkce.

Dokazatelnost

Nechť T je fuzzy teorie a $A \in F_J$ je formule. $T \vdash_a A$ je definováno jako A je teorémem ve stupni a, nebo dokazatelná ve stupni a ve fuzzy teorii T.

$$T \vdash_a A \text{ právě tehdy když } a = \vee \{Val(w) \mid w \text{ je důkaz A z T}\}$$

$T \models_a A$ znamená A je pravdivá ve stupni a v teorii T.

Fuzzy modus ponens

$$r_{MP}: \frac{a / A,\ b / A \Rightarrow B}{a \otimes b / B}$$

V klasické logice je možné nahlížet na pravidlo modus ponens jako na speciální případ rezoluce. To lze aplikovat i pro fuzzy logiku.

Neklazulární rezoluce pro fuzzy predikátovou logiku (GR_{FPL})

$$r_{GR}: \frac{a / F[G_1,,...,G_k],\ b / F'[G'_1,...,G'_n]}{a \otimes b / F\sigma[G / \perp] \nabla F'\sigma[G / T]}$$

Příklad 16.

Common proof members (axioms):

1. $0.8/\forall X[\exists Y[child(X,Y)\ \&\ female(Y)]$
   $\&\ \exists Y[child(X,Y)\ \&\ male(Y)] \Rightarrow happy(X)]$ *(happy with parents - 0.8)*
2. $0.5/\forall X[toys(X) \Rightarrow happy(X)]$ *(happy with toys - 0.5)*
3. $1/child(johana, hashim)$ *(clear crisp fact)*
4. $1/child(johana, lucie)$ *(clear crisp fact)*
5. $1/male(hashim)$ *(clear crisp fact)*
6. $1/female(lucie)$ *(clear crisp fact)*
7. $0.9/toys(johana)$ *(johana has a lot of toys - 0.9)*
8. $1/\neg happy(johana)$ *(negated goal - is johana happy?)*

Proof 1:

9. $0.9 \otimes 0.5/\bot\nabla[\top \Rightarrow happy(johana)]$
9a. $0.4/happy(johana)$ *(r_{GR} on 7.,2., $Sbt(X) = johana$)*
10. $1 \otimes 0.4/\bot\nabla\neg\top$
10a. $0.4/\bot$ *(r_{GR} on 9.,8.)*

(happy(johana) is provable in 0.4)

Nově definované pravidlo „General Fuzzy Resolution“pak ze dvou premis s hodnotou a a b vytváří novou rezolventu klasickým způsobem jako v případě klasické logiky ovšem pomocí spojky Lukasiewiczovy disjunkce a hodnota nové rezolventy je rovna a⊗b. Důkaz korektnosti resp. úplnosti formule je proveden pomocí induktivního resp. konstruktivního přístupu. Pro důkaz úplnosti pak postačí dvě zjednodušovací pravidla pro redukci formulí s logickými konstantami a samotný důkaz je pak konstruktivní a provádí nahrazení sekvence s pravidlem Modus-ponens pomocí nově zavedeného pravidla General Fuzzy Resolution. Tedy toto pravidlo může beze zbytku nahradit Fuzzy Modus-ponens, se kterým je logika úplná.

Toto však neřeší dva základní problémy, které jsou pro praktickou realizaci navrhovaného rezolučního principu podstatné. Jde o nepřímé dokazování a o rozšíření na fuzzy predikátovou logiku. Tyto otázky jsou vyřešeny až následně a spočívají v rozšíření standardní definice pojmu ohodnoceného důkazu o pojem ohodnoceného nepřímého důkazu („evaluated refutational formal proof“) a příslušného pojmu hodnoty nepřímého důkazu – stupně popření („refutation degree“). Nepřímý důkaz a stupeň popření jsou dodefinovány a rozšiřují klasickou definici. Samozřejmě je potřeba najít ekvivalenci mezi dokazatelností formule a supremem stupňů popření nepřímých rezolučních důkazů a důkaz je opět konstruktivní. Druhou otázkou je realizace unifikace v případě nevýrokových formulích. Jelikož zvolená fuzzy logika pracuje s crisp termy (a nesnižuje to expresivitu, neboť fuzzy termy lze simulovat pomocí fuzzy predikátů), lze využít shodnou unifikační proceduru jako u klasické predikátové logiky. Uvedený příklad dedukce pak ukazuje, že ve fuzzy logice jde o dimenzi složitější problém, protože může existovat mnoho různých důkazů s odlišnými stupni příslušnosti. To v klasické logice není, protože stačí najít jeden možný důkaz (formule buď logicky vyplývá nebo ne), zatímco ve fuzzy logice musíme potenciálně hledat několik důkazů (i nekonečně mnoho) a z nich vybrat nejlepší (viz definice dokazatelnosti formule).

Důkaz s vyšším stupněm dokazatelnosti:

Proof 2:

9. $0.8 \otimes 1/[\exists Y[child(johana, Y) \,\&\, female(Y)]$
$\&\, \exists Y[child(johana, Y) \,\&\, male(Y)] \Rightarrow \bot]\nabla\neg\top$

9a. $0.8/\neg[\exists Y[child(johana, Y) \,\&\, female(Y)]$
$\&\, \exists Y[child(johana, Y) \,\&\, male(Y)]]$ (r_{GR} *on 1.,8.,* $Sbt(X) = johana$)

10. $0.8 \otimes 1/\neg[[child(johana, lucie) \,\&\, \top]$
$\&\, \exists Y[child(johana, Y) \,\&\, male(Y)]]\nabla\bot$

10a. $0.8/\neg[child(johana, lucie)$
$\&\, \exists Y[child(johana, Y) \,\&\, male(Y)]]$ (r_{GR} *on 6.,9.,* $Sbt(Y) = lucie$)

11. $0.8 \otimes 1/\neg[child(johana, lucie)$
$\&[child(johana, hashim) \,\&\, \top]]\nabla\bot$

11a. $0.8/\neg[child(johana, lucie)$
$\&\, child(johana, hashim)]$ (r_{GR} *on 5.,10.,* $Sbt(Y) = hashim$)

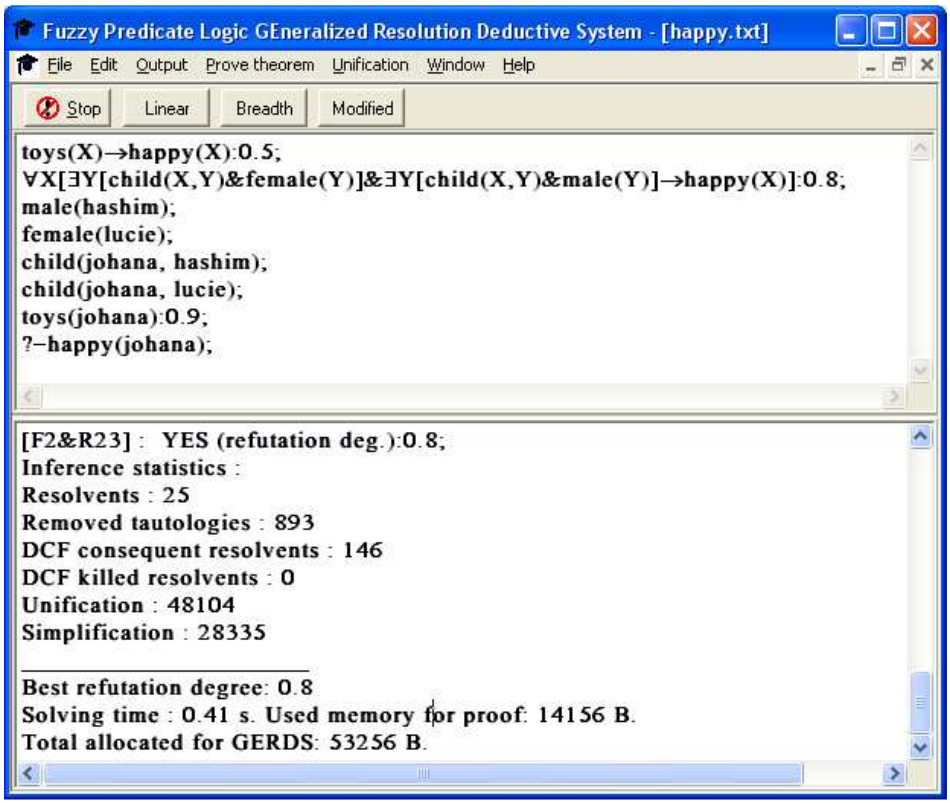
12. $0.8 \otimes 1/\neg[\top \,\&\, child(johana, hashim)]\nabla\bot$

12a. $0.8/\neg[child(johana, hashim)]$ (r_{GR} *on 4.,11.*)

13. $0.8 \otimes 1/\neg\top\nabla\bot$

13a. $0.8/\bot$ (r_{GR} *on 3.,12.*)

(happy(johana) is provable in 0.8)



Obrázek 13: Fuzzy Predicate Logic GEneralized Deductive System

Samozřejmě chceme podobně jako u klasické logiky mít k dispozici také experimentální nástroj pro nepřímé rezoluční dokazování ve fuzzy logice. Jelikož unifikační procedura (nejsložitější prvek systému) je shodná s klasickou logikou, nebylo nijak dimenzionálně obtížnější sestavit novou počítačovou aplikaci pojmenovanou FPLGERDS (Fuzzy Predicate Logic GERDS). Použité techniky tvorby syntaktického formačního stromu jsou totožné, ovšem bylo třeba obohatit formule také o ohodnocení prvkem množiny pravdivostních hodnot a realizovat nové interpretace logických spojek. Další problém se skrývá v hledání „nejlepšího“ nepřímého důkazu. Nelze tedy používat dobře známé inferenční strategie, jejichž myšlenka je většinou spojena s nejrychlejším vyhledáním jakéhokoliv (jediného) důkazu, což je ve fuzzy logice nepoužitelný přístup. Proto byla snaha nalézt jistou variantu již probírané metody DCF. Ani tato strategie není ovšem ve fuzzy logice použitelná obecně. Může se použít jen v případě, že nově vygenerovaná rezolventa vyplývá z některé z předchozích a zároveň je její ohodnocení menší nebo rovno předchozí. To samozřejmě zejména u redundantních znalostních bází bude způsobovat menší účinnost než v klasické logice, přesto jde o velmi efektivní strategii.

4.4 Klasické rezoluční strategie ve dvouhodnotové logice

Existuje několik strategií generování rezolvent. Z pohledu způsobu řazení rezolvent a jejich následného využití v procesu inference existují v principu dva přístupy:

- Prohledávání do šířky (rezoluční uzávěr).
- Prohledávání do hloubky.

Prohledávání do šířky

Generováním do šířky generujeme všechny rezolventy, které je možno vytvořit z množiny vstupních formulí (proto je tato metoda také nazývána rezolučním uzávěrem), ty jsou pak označovány jako rezolventy prvního stupně. V dalším kroku je generována množina rezolvent druhého stupně, které se generují z rezolventy některého předcházejícího stupně a alespoň jedné rezolventy bezprostředně předchozího. Tímto způsobem se pokračuje generováním rezolvent dalších stupňů dokud není nalezeno rezoluční popření či nevyprší daný limit. Následuje příklad důkazu logického důsledku pomocí strategie prohledávání do šířky.

Příklad 17.

Výchozí formule (axiomy) :

F0 : $[a(X) \wedge g(X) \rightarrow b(X)]$.

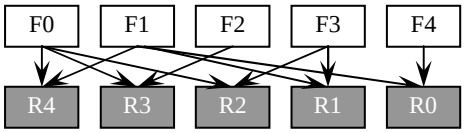
F1 : $[b(X) \wedge g(X) \rightarrow c(X)]$.

F2 : $a(a)$.

F3 : g(a).
F4 (¬dotaz) : ¬c(Y).

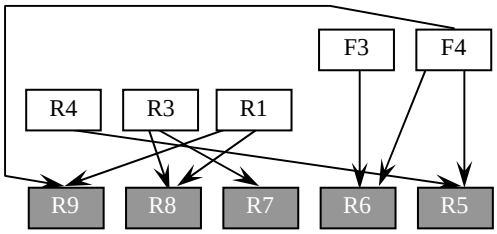
1. Fáze – generování rezolvent prvního stupně z množiny vstupních formulí:

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [F3&F1] : [b(a)→c(a)].
R2 [F3&F0] : [a(a)→b(a)].
R3 [F2&F0] : [g(a)→b(a)].
R4[F1&F0]: [[g(X)→c(X)]∨¬[a(X)∧g(X)]].



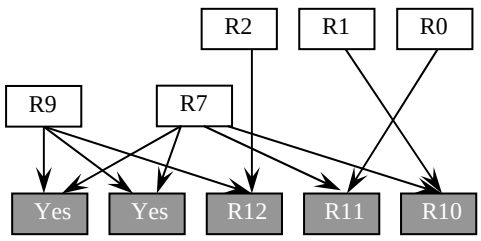
2. Fáze – generování rezolvent druhého stupně z alespoň jedné rezolventy prvního stupně a druhá rezolventa může být buďto výchozí formule nebo rezolventa prvního stupně.

R5 [R4&F4] : [¬g(X)∨¬[a(X)∧g(X)]].
R6 [R4&F3] : [c(a)∨¬a(a)].
R7 [R3&F3] : b(a).
R8 [R3&R1] : [¬g(a)∨c(a)].
R9 [R1&F4] : ¬b(a).



3. Fáze – generování rezolvent třetího stupně z alespoň jedné rezolventy druhého stupně a druhá rezolventa může být výchozí formule, nebo rezolventa 1. i 2. stupně.

R10 [R1&R7] : c(a).
R11 [R0&R7] : ¬g(a).
R12 [R9&R2] : ¬a(a).
[R7&R9] : YES.
Y = a.
[R9&R7] : YES.
Y = a.



4. Fáze - generování rezolvent čtvrtého stupně z alespoň jedné rezolventy třetího stupně a druhá rezolventa může být výchozí formule, nebo rezolventa 1., 2. i 3. stupně.

[R10&F4] : YES.

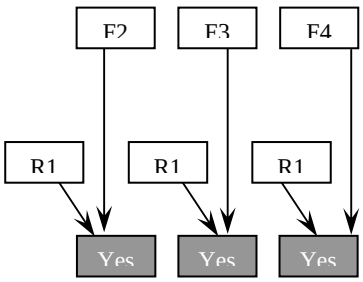
Y = a.

[R11&F3] : YES.

Y = a.

[R12&F2] : YES.

Y = a.



Jednotlivé fáze prohledávání do šířky postupně vedou k důkazu závěru c(Y). Již ve třetí fázi tato strategie našla dvě možné cesty k popření, které bychom nejspíše rovnou použili při dokazování bez použití automatizace. I v následující fázi bylo nalezeno popření pro zbývající tři cesty. Můžeme říci že tato strategie je úplná a pokud popření existuje tak bude nalezeno. Nevýhodou této strategie pro použití u automatizace je generovaní velkého počtu rezolvent, zejména těch, které k popření vůbec nepotřebujeme, což může být implementačně velmi neefektivní, zejména kvůli zbytečnému plýtvání paměťovými prostředky.

Prohledávání do hloubky

Pro automatizaci v logickém programování se používá strategie generování do hloubky, která generuje rezolventu ze dvou premis ze vstupní množiny a pak opakovaně aplikuje rezoluci na výslednou rezolventu a další rezolventu nebo axiom rekurzivně. Dojde-li algoritmus k rezolventě, ze které již nelze další rezolventu vygenerovat, vrací se o úroveň výše a pokračuje s další možnou rezolventou. Tato strategie není úplná, protože může uvíznout v nekonečné větvi, i když řešení existuje. Oproti tomu je její velkou výhodou, že vede často rychleji k nalezení popření než strategie kde se používá prohledávání do šířky a je tedy vhodnější k automatizaci. Ukažme si tedy příklad důkazu logického důsledku za použití prohledávání do hloubky.

Příklad 18.

Výchozí formule (axiomy) :

F0 : [a(X)∧g(X)→b(X)].

F1 : [b(X)∧g(X)→c(X)].

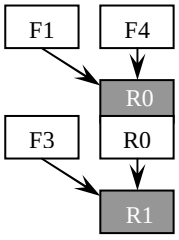
F2 : a(a).

F3 : g(a).

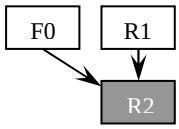
F4 (¬dotaz) : ¬c(Y).

R0 [F4&F1] : ¬[b(Y)∧g(Y)].

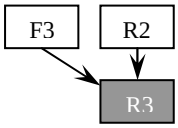
R1 [R0&F3] : ¬b(a).



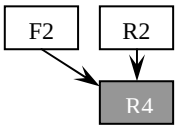
R2 [R1&F0] : ¬[a(a)∧g(a)].



R3 [R2&F3] : ¬a(a).

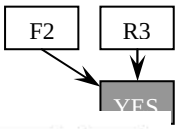


R4 [R2&F2] : ¬g(a).



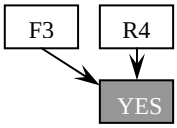
[R3&F2] : YES.

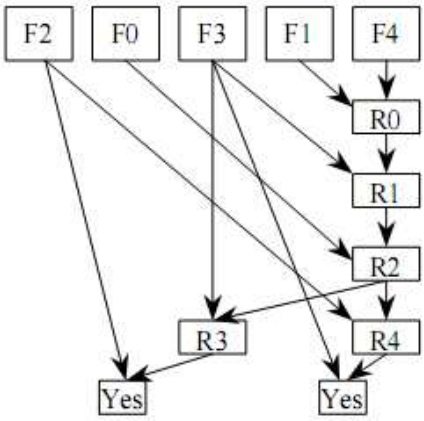
Y = a.



[R4&F3] : YES.

Y = a.





Je vidět že pro důkaz prohledáváním do hloubky stačilo pouze 5 rezolvent, je tedy mnohem kratší a navíc se pro automatizaci nabízí využití rekurze. Dalším vylepšením prohledávání do hloubky je široce používaná (např. PROLOG) lineární strategie. I když není úplná pro obecné formule, při používání omezeného jazyka predikátové logiky máme úplnost zaručenu a její efektivita je již dostatečná pro praktickou realizaci inference. Metoda je jednoduchá, generujeme rezolventy od prověřovaného závěru a výsledek rezoluce vždy použijeme v dalším kroku, jako jednu z premis.

4.5 Detekce konsekventních formulí (DCF algoritmus)

Pokud chceme použít neklauzulární rezoluční pravidlo na zcela obecné formule a mít zaručenu úplnost metody, pak je řešením použít strategii prohledávání do šířky, která ovšem vede ke kombinatorické explozi. Autor navrhl algoritmus, kterým však lze tuto neefektivitu částečně odstranit a díky tomu je i odvozování pro plnou predikátovou logiku prakticky realizovatelné.
Myšlenka detekce konsekventních formulí (DCF) je zajímavá také díky tomu, že její součástí je neklauzulární rezoluce a to její speciální případ nazývaný „self-resolution“ – jde o aplikaci na stejnou formuli jako pozitivní i negativní premisu. To usnadňuje implementaci, neboť hotové postupy pro neklauzulární

rezoluci se mohou použít i pro DCF. DCF lze použít pouze pro nepřímé dokazování, které je ale mnohem účelnější než přímé generování důsledků.
DCF spočívá ve filtrování potenciální rezolvent vzniklých rezolucí. Z pohledu nepřímého dokazování, kdy se snažíme najít popření – false, jsou dva typy formulí zbytečné:

- tautologie (přidáním tautologie k množině formulí se interpretace této množiny nijak nezmění),
- logické důsledky již zařazených rezolvent (je-li prázdná formule odvoditelná z množiny původních rezolvent a axiómů, pak je z pohledu modelů takovéto množiny logický důsledek neproduktivní, neboť zachovává všechny modely původní množiny).

Výhodnost použitých unifikačních technik a celé neklauzulární rezoluce nemusí být z teoretických pasáží zcela zřejmá pro klasickou logiku (u deskripční logiky, fuzzy logiky či fuzzy deskripční logiky je důvodu hned několik, které byly diskutovány). Bylo by možné namítat, že u klasické logiky nám klauzulární rezoluce s mnoha prověřenými strategiemi dává v praxi lepší výsledky. Pokud se však zamyslíme nad možnostmi implementace, které dává hierarchická syntaktická struktura formule, najdeme mnoho výhod i pro rezoluci neklauzulární.
Myšlenka aplikovat neklauzulární rezoluční princip ve fuzzy logice samozřejmě přímo vybízela také k implementaci. Implementace je založena na základech popsaných v kapitole věnované fuzzy logice. Velkým rozdílem je oproti klasické logice nutnost hledat nepřímý důkaz s nejvyšším stupněm popření. To tedy znamená prohledávat potenciálně všechny možné důkazy. Rovněž strategie DCF musela být významně změněna – záleží totiž na stupni dokazatelnosti dané rezolventy a nelze jednoduše odfiltrovat všechny vyplývající rezolventy, ale pouze ty s nižší nebo stejným stupněm příslušnosti. Obecně není žádným překvapením, že nepřímé dokazování (a to nejen rezoluční) je ve fuzzy logice principiálně mnohem časově náročnější problém a tudíž tu efektivní inferenční strategie hrají ještě důležitější úlohu, aby automatizovaná dedukce ve fuzzy prostředí mohla mít úspěch na praktických úlohách. Jedná se v podstatě o rodinu strategií, které v kombinaci s klasickými strategiemi tvoří 9 možných způsobů hledání důkazů.

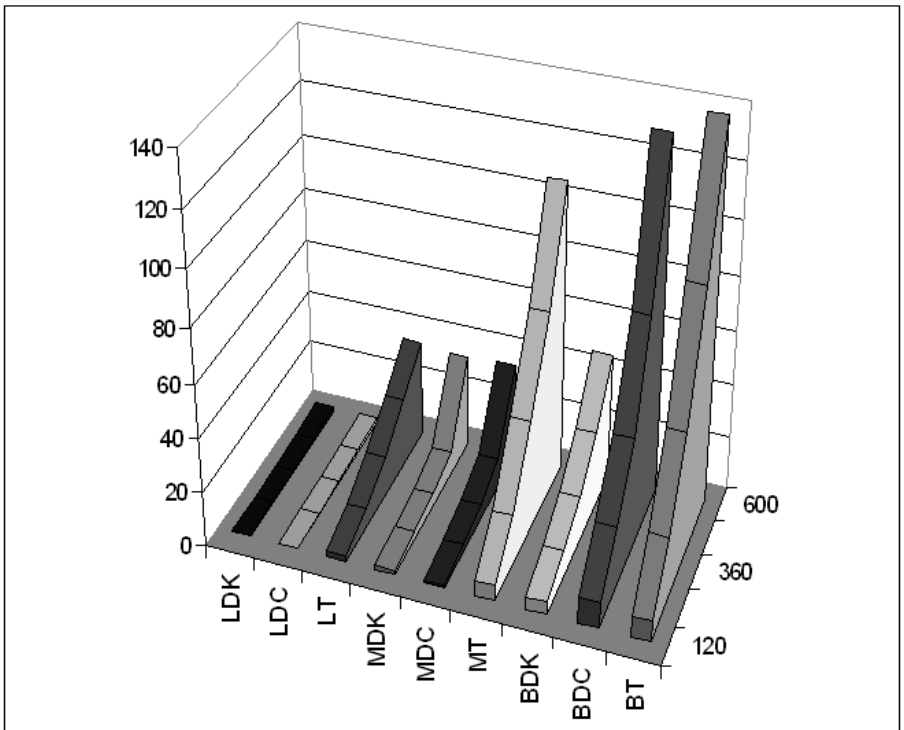
Search method		**Description**
Breadth	B	Level order generation, start - special axioms + goal
Linear	L	Resolvent ⇒ premise, start - goal
Modified-Linear	M	Resolvent ⇒ premise, start - goal + special axioms

Obrázek 14: Strategie hledání důkazu

DCF Method		Description
Trivial	T	Exact symbolic comparison
DCF	DC	Potential resolvent is consequent (no addition)
DCF Kill	DK	DCF + remove all consequent resolvents

Obrázek 15: Strategie DCF

Pro tyto kombinace pak byly provedeny experimenty časové a prostorové náročnosti pro různě velké znalostní báze s různým počtem formulí. Příklad lze najít na obrázku.



	120	240	360	480	600
BT	7,44	25,09	50,55	90,91	138,67
BDC	9,52	23,73	43,76	76,63	129,94
BDK	4,68	10,1	17,64	29,54	47,31
MT	5,86	18,85	41,46	70,01	106,68
MDC	0,93	3,09	5,14	10,54	35,14
MDK	1,45	3,2	5,03	8,84	34,32
LT	2,17	7,24	14,76	24,98	35,77
LDC	0,26	0,52	0,57	0,8	1,96
LDK	0,28	0,5	0,57	0,78	1,99

Obrázek 16: Porovnání časové efektivity strategií ve fuzzy logice

- Rezoluční strategie

5 Modelové deduktivní systémy

V této kapitole se dozvíte:

- Příklady deduktivních systémů od autora opory i jiných autorů
- Podrobné ukázky tvorby systému
- Ukázky funkčnosti systému

Po jejím prostudování byste měli být schopni:

- Pracovat s již hortovými systémy
- Navrhnout a implementovat vlastní jednoduchý deduktivní systém

Klíčová slova této kapitoly:

Symbolická logika, logická dedukce, formální dedukce, rezoluční strategie

Doba potřebná ke studiu: 6 hodin

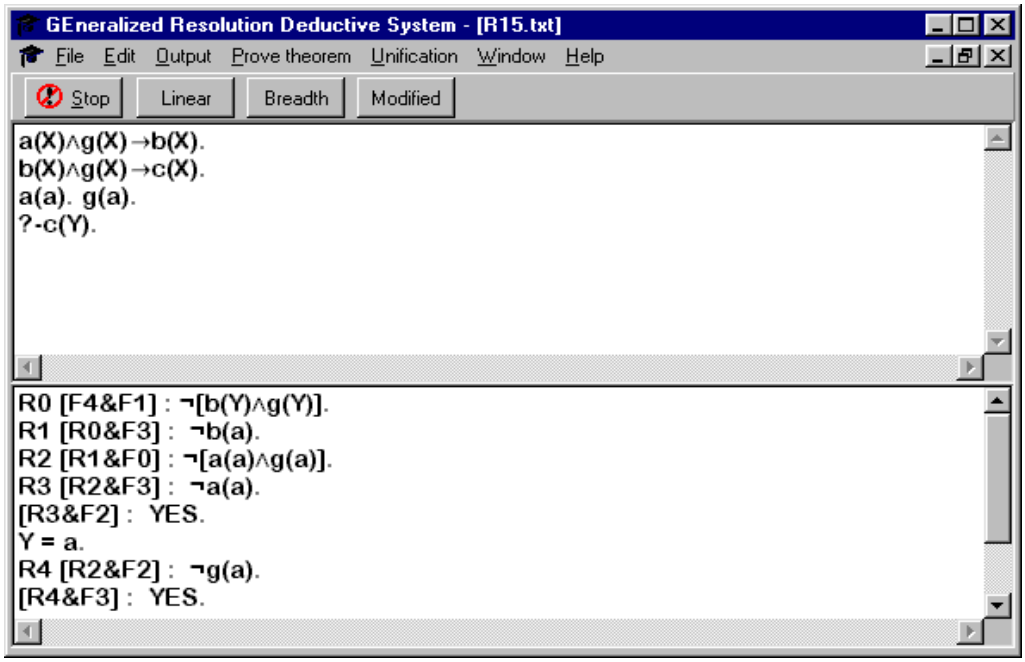
Průvodce studiem

Studium této kapitoly by mělo být vyvrcholením vašich snah o pochopení automatické dedukce. Pokud jste pochopili předchozí kapitoly, pak je tato kapitola pro vás již jednoduchá.
Na studium této části si vyhraďte alespoň 8 hodin. Doporučujeme studovat s přestávkami vždy po pochopení jednotlivých podkapitol. Po celkovém prostudování a vyřešení všech příkladů doporučujeme dát si pauzu, třeba 1 den, a pak se pusťte do vypracování korespondenčních úkolů.

V této kapitole si ukážeme podrobný rozbor algoritmů, datových struktur i uživatelského rozhraní již hotových deduktivních systémů.

5.1 Generalized Resolution Deductive System

Jako jedna z nejlepších ukázek (to neznamená, že je to nejlepší deduktivní systém) aplikaci vytvořenou autorem pro oblast automatizované dedukce (konkrétně zobecněné rezoluční metody) (Bachmair, 1997) a logiky obecně. V předcházejících kapitolách jsme se zmiňovali o počítačové aplikaci – prostředku, který se nyní pokusíme detailně popsat. Je realizována na 32-bitové platformě Windows (95,98,NT atd.).



Obrázek 17: Uživatelské rozhraní aplikace GERDS

Autor v rámci diplomové práce využil teoretických východisek automatizace zobecněného pravidla rezoluce a navrhl a implementoval počítačovou aplikaci (Generalized Resolution Deductive System – GERDS), která je použitelná jako didaktický prostředek pro demonstraci nejen nových směrů ve výzkumu automatizované dedukce, ale také pro pochopení klasických přístupů inferenčních postupů.

Uživatelské rozhraní aplikace je velmi jednoduché. Jde o MDI (Multiple document interface) aplikaci, která dává možnost pracovat s více otevřenými příklady najednou. Každé nezávislé okno příkladu má dvě části – *Editor* pro vstupní formule (axiomy) a *Výstup* pro sledování procesu inference a jejích výsledků. Na obrázku můžete vidět příklad otevřeného okna.

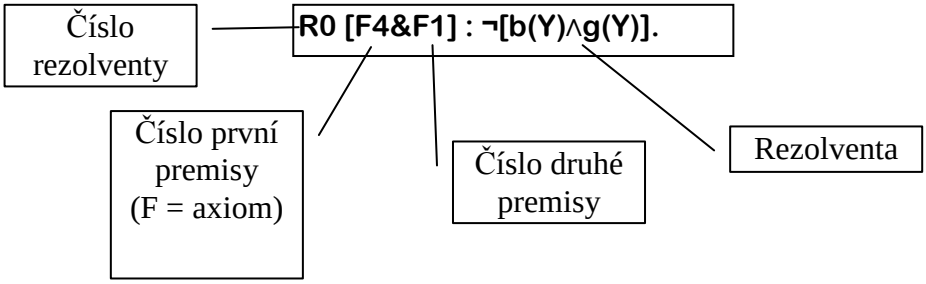
Editor dovoluje vkládat výchozí formule (axiomy) a cíle (dotazy), jejichž platnost chceme prověřit. Dotaz je uvozen řetězcem ?- a může se vyskytovat vícekrát, systém pak vyhodnotí všechny dotazy, které se v *Editoru* vyskytují. Každý dotaz se dokazuje (odvozuje) pouze s pomocí formulí, které jsou uvedeny před ním v okně *Editoru*. Podívejme se na následující příklad.

Příklad 19.

a(X)∧g(X)→b(X).
b(X)∧g(X)→c(X).
a(a). ?-g(a).
g(a).
?-c(Y).

V tomto příkladu máme tři výchozí formule uvedené před první dotazem. Jak uvidíte, pokud si zkusíte jej vyhodnotit, tento první dotaz nebude vyhodnocen jako platný, protože výchozí tři formule nedávají dostatečné znalosti pro jeho

odvození. Následuje čtvrtý axiom a za ním další dotaz, který by již v tomto případě byl dokázán jako platný, neboť poslední axiom umožní jeho odvození.
Okno *Výstupu* může obsahovat seznam výchozích formulí a cílů, sekvenci rezolvent, nezjednodušených forem rezolvent a statistiku inference. Položky sekvence formulí jsou v následující podobě na obrázku.



Obrázek 18: Struktura sekvence rezolvent

Podívejme se na příklad takovéto sekvence.

Příklad 20.

Výchozí formule (axiomy) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 (¬dotaz) : ¬g(a).

Výchozí formule (axiomy) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 : g(a).
F4 (¬dotaz) : ¬c(Y).

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [R0&F3] : ¬b(a).
R2 [R1&F0] : ¬[a(a)∧g(a)].
R3 [R2&F3] : ¬a(a).
[R3&F2] : YES.
Y = a.
R4 [R2&F2] : ¬g(a).
[R4&F3] : YES.
Y = a.

Rezolventa, která obsahuje místo formule řetězec YES, označuje rezolventu false, tedy popření množiny axiomů a znegovaného cíle. Pod touto rezolventou se vyskytuje výraz Y = a, což je přiřazení konstanty proměnné Y, které způsobilo nalezení důkazu. Pro práci s příklady je tato možnost nezbytná, aby se dalo hovořil o prakticky použitelném odvozovacím prostředku.
Podívejme se nyní na didaktických příkladech na některé vybrané strategie, jejichž vysvětlení a praktická ukázka s pomocí programu GERDS zlepšuje pochopení teoretických východisek automatizované dedukce. Použijeme k tomu příklady vygenerované s pomocí aplikace GERDS, kterou mohou

využívat sami studenti. Velmi dobrým zdrojem informací o strategiích je kniha (Mařík, 1993). Prohledávání stavového prostoru lze realizovat různými přístupu a tento přístup je pak univerzální pro různé typy úloh, což samozřejmě platí i o hledání rezolučního důkazu, generováním rezolvent.

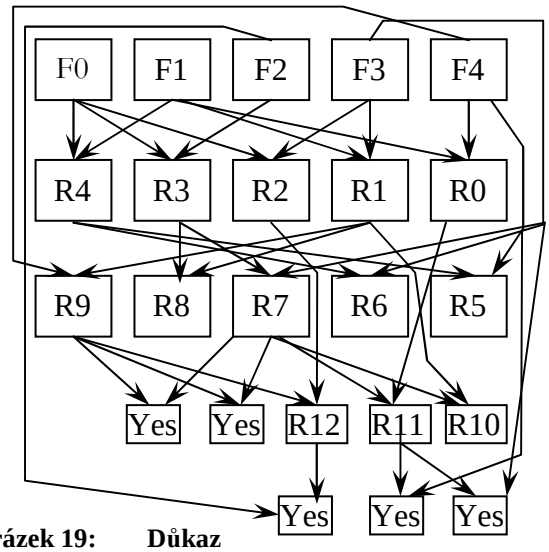
Strategie prohledávání do šířky spočívá v generování všech rezolvent, které lze vytvořit z množiny vstupních formulí (axiomů) a negovaného závěru (nultý stupeň) a ty se pak nazývají rezolventy prvního stupně. Potom se pokračuje v generování všech možných rezolvent druhého stupně, které vznikají z libovolné rezolventy nižšího stupně a alespoň jedné rezolventy bezprostředně předchozího tedy prvního stupně. Pokračuje v generování rezolvent dalších stupňů podle stejného pravidla až je nalezeno rezoluční popření nebo do nekonečna či určitého limitu, pokud takové popření neexistuje. Podívejme se na následující příklad důkazu:

Příklad 21.

Výchozí formule (axiomy) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 : g(a).
F4 (¬dotaz) : ¬c(Y).

_____

R0 [F4&F1] : ¬[b(Y)∧g(Y)]. R1 [F3&F1] : [b(a)→c(a)].
R2 [F3&F0] : [a(a)→b(a)]. R3 [F2&F0] : [g(a)→b(a)].
R4[F1&F0]: [[g(X)→c(X)]∨¬[a(X)∧g(X)]].
R5 [R4&F4] : [¬g(X)∨¬[a(X)∧g(X)]].
R6 [R4&F3] : [c(a)∨¬a(a)].
R7 [R3&F3] : b(a).
R8 [R3&R1] : [¬g(a)∨c(a)].
R9 [R1&F4] : ¬b(a).
R10 [R1&R7] : c(a).
R11 [R0&R7] : ¬g(a).
[R11&F3] : YES.
Y = a.
[R10&F4] : YES.
Y = a.
[R9&R7] : YES.
Y = a.
R12 [R9&R2] : ¬a(a).
[R7&R9] : YES.
Y = a.
[R12&F2] : YES.
Y = a.



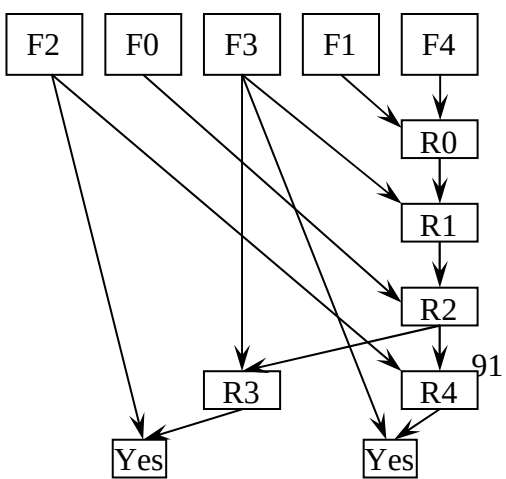
Obrázek 19: Důkaz pomocí strategie prohledávání „do šířky“

Obrázek a důkazová sekvence nám ukazuje, jak postupně probíhal důkaz závěru c(Y). Vidíme, že postupným generováním stupňů rezolvent, jsme nejprve odvodili antecedent implikace F1 jako rezolventu R0. Spolu s ní vznikly další rezolventy R1 – R4. V dalším stupni se vygenerovaly další rezolventy a již ve třetím stupni jsme našli dvě možné cesty k popření, z nichž právě první vznikla z tohoto antecedentu (jde zřejmě o nejjasnější cestu, kterou bychom použili při rezolučním popření při dokazování bez použití automatizace). I na další úrovni ještě bylo dosaženo popření třemi dalšími cestami. Vidíme, že tato strategie vedla k cíli (hned několika cestami), avšak vzniklo při ní mnoho rezolvent. Tato metoda je sice úplná (tedy pokud popření existuje, pak bude nalezeno), ale za cenu zpomalení důkazu, generováním značného množství rezolvent. Tomu se naopak vyhneme při použití další strategie.

Strategie prohledávání do hloubky generuje rezolventu ze dvou premis ze vstupní množiny a pak opakovaně aplikuje rezoluci na výslednou rezolventu a další rezolventu nebo axiom rekurzivně, až nelze žádnou rezolventu již dále vygenerovat. Poté se algoritmus vrátí na předchozí úroveň (provádí backtracking) a pokračuje s další možnou rezolventou. Tento typ strategie není úplný, neboť v případě nekonečné sekvence generovaných rezolvent uvízne v nekonečné smyčce a nemusí tak najít řešení, které by se vyskytlo v jiné větvi. Tato strategie vede často rychleji k cíli než předchozí ovšem právě s omezením, které již bylo zmíněno. Podívejme se na stejný příklad řešený s pomocí tohoto prohledávání.

Příklad 22.

Výchozí formule (axiomy) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 : g(a).
F4 (¬dotaz) : ¬c(Y).



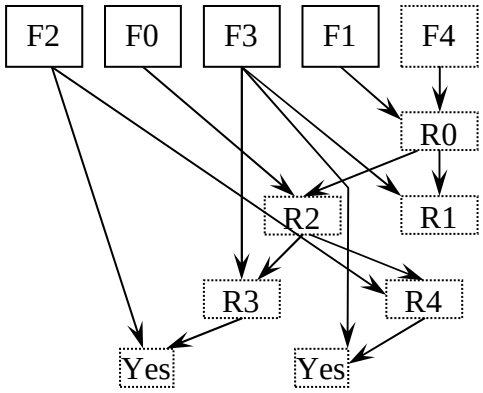
R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [R0&F3] : ¬b(a).
R2 [R1&F0] : ¬[a(a)∧g(a)].
R3 [R2&F3] : ¬a(a).
[R3&F2] : YES.
Y = a.
R4 [R2&F2] : ¬g(a).
[R4&F3] : YES.
Y = a.

Obrázek 20: Důkaz pomocí strategie prohledávání „do hloubky“

Vidíme, že důkaz s pomocí této strategie je mnohem kratší a stačí k němu vygenerování pouze 5 rezolvent.
Mezi další pomocné strategie, které zefektivňují inferenci patří také například lineární strategie spočívající v použití poslední generované klauzule při každé další rezoluci (vytváří se tak vlastně neporušený řetěz při generování). Jde o základní techniku používanou při logickém programování. V aplikaci GERDS se používají dva typy lineární strategie – lineární, která vychází při první rezoluci pouze z dotazu a modifikovaná lineární strategie, která není omezena pouze na dotaz. Strategie podpůrné množiny je jednoduchá a efektivní, ale bohužel také neúplná strategie. Vznikla na základě faktu, že v každé množině formulí existuje konsistentní podmnožina. Je pochopitelné, že rezolventy z této podmnožiny nemohou vést k rezolučnímu popření. Proto tato strategie umožní aplikovat rezoluci na takových formulích, které jsou buď dotazem nebo z dotazu vznikly. Podívejme se opět na příklad:

Příklad 23.

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [R0&F3] : ¬b(a).
R2 [R0&F0] : [¬g(Y)∨¬[a(Y)∧g(Y)]].
R3 [R2&F3] : ¬a(a).
R4 [R2&F2] : ¬g(a).
[R4&F3] : YES.
Y = a.
[R3&F2] : YES.
Y = a.



Obrázek 21: Strategie podpůrné množiny

Strategie podpůrné množiny umožňuje usměrňovat proces inference, aby byl mnohem efektivnější.
Filtrační strategie je další metodou redukce důkazu. Dvě premisy A,B mohou být použity do rezoluce tehdy a jen tehdy, pokud platí tyto dvě podmínky:

1. A a B jsou z množiny výchozích formulí

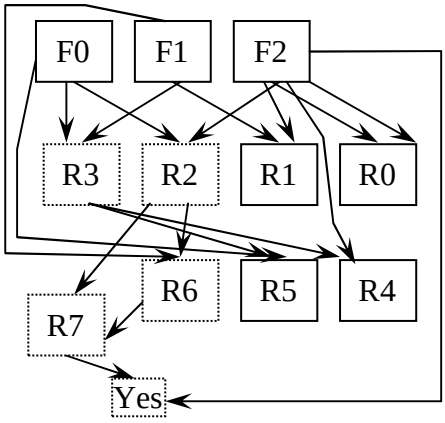
2. A je potomkem B nebo B je potomkem A (potomek je rezolventa vygenerovaná z rodičovské)

To zda je formule potomkem, lze zjistit ze stromu rezolučního popření. Tato strategie je úplná, i když není tak efektivní jako předchozí. Podívejme se na příklad použití:

Příklad 24.

Výchozí formule (axiomy) :
F0 : a→b∧g.
F1 : b∧g→c.
F2 (¬dotaz) : ¬[a→c].

R0 [F2&F2] : ¬c.
R1 [F2&F1] : ¬[b∧g].
R2 [F2&F0] : b∧g.
R3 [F1&F0] : [g→c]∨¬a.
R4 [R3&F2] : ¬g∨¬a.
R5 [R3&F0] : c∨¬a.
R6 [R2&F1] : g→c.
R7 [R2&R6] : c.
[R7&F2] : YES.



Obrázek 22: Filtrační strategie

Poslední strategií, která je ovšem nejúčinnější a byla vyvinuta speciálně pro tuto aplikaci, je detekce redundantních rezolvent. Podívejme se nyní na příklady jejího využití:

Příklad 25.

Uvažujme formuli **[a ∨ b] ∧ [¬b ∨ c]** dokažme, že **[a ∨ c]** je z ní vyplývající formulí. Provedeme rezoluci na negaci implikace výchozí a vyplývající formule:

¬[[a∨b]∧[¬b∨c]→a∨c]

¬[[⊥∨b]∧[¬b∨c]→ ⊥∨b]∨¬[[⊤∨b]∧[¬b∨c]→ ⊤∨b] ⇒ ⊥

Bylo použito zjednodušujících pravidel a dostali jsme tím logickou hodnotu false, což znamená že jsme dokázali, že jde skutečně o vyplývající formuli. Podívejme se ještě na jeden příklad:

Formule **a∧b** je konsekventní formulí **[a↔b]∧a.**

¬[[a↔b]∧a→a∧b]

[¬[[⊤↔b]∧ ⊤→ ⊤∧b]∨¬[[⊥↔b]∧ ⊥→ ⊥∧b]]∧[¬[[⊥↔b]∧ ⊥→ ⊥∧b]∨¬[[⊤↔b]∧ ⊤→ ⊤∧b]] ⇒ ⊥

Důležitým aspektem detekce konsekventních formulí je její jednoduchá implementace v systému GERDS, kde je již vytvořena procedura pro zobecněnou rezoluci. Aplikace umí s formulemi zacházet jako se syntaktickými stromy a proto je jednoduché vytvořit virtuální strom negace implikace z kandidátské a předchozí rezolventy a provést rezoluci, na jejímž základě po několika krocích zjistíme, zda dospějeme k false – tedy důkazu, že implikace je platná.

Porovnejme důkazy s použitím standardních strategií s detekcí konsekventních rezolvent.

Příklad 26.

F0 : a↔b∧g. F1 : b∧g↔c. F2 (¬dotaz) : ¬[a↔c].
R0 [F2&F1] : [¬a∨¬[b∧g]]∧[a∨b∧g]. **R1 [R0&F2] : ¬[b∧g]∨c.**
R2 [R1&F0] : ¬g∨c∨¬a. **R3 [R2&F2] : ¬g∨¬a.**
R4 [R3&F2] : ¬g∨c. **R5 [R4&F0] : c∨¬a.**
R6 [R5&F2] : ¬a. **R7 [R6&F2] : c.**
R8 [R7&F1] : b∧g. **R9 [R8&F1] : g↔c.**
R10 [R9&F2] : [g∨a]∧[¬g∨¬a]. **R11 [R10&F1] : a∨[b↔c].**
R12 [R11&R6] : b↔c. **R13 [R12&F2] : [b∨a]∧[¬b∨¬a].**
R14 [R13&F0] : a∨[a↔g]. **R15 [R14&F2] : ¬g∨¬c.**
R16 [R15&F1] : ¬c. **R17 [R16&F2] : a.**
[R17&R6] : YES.
Čas : 0.22 s.

Žádná ze standardních strategií nebyla schopna omezit velikost důkazu. Čtenář si sám může vyzkoušet, že bez kontroly konsekventních rezolvent jsou důkazy naprosto nekontrolované a vytvářejí v tomto případě během několika sekund více než 300 rezolvent.

Příklad 27.

F0 : a∧¬b∧c∧d∨a∧¬b∧¬c∧d∨¬a∧¬b∧c∧d∨a∧¬b∧¬c∧d∨a∧¬b∧c∧d.
F1 (¬dotaz) : ¬[¬a∧¬b∨a∧¬b].

R0 [F1&F1] : b.
R1 [F1&F0] : b∨¬b∧c∧d∨¬b∧¬c∧d∨¬b∧¬c∧d∨¬b∧c∧d.
R2 [F0&F1] : ¬b∧c∧d∨b.
R3 [F0&F0] : ¬b∧c∧d∨¬b∧c∧d∨¬b∧¬c∧d∨¬b∧¬c∧d∨¬b∧c∧d.
[R3&F1] : YES.
Čas: 0.05 s.
(bez restrikce)
R0 [F1&F1] : b.
R1 [F0&F0] : ¬b∧c∧d∨¬b∧c∧d∨¬b∧¬c∧d∨¬b∧¬c∧d∨¬b∧c∧d.
[R1&F1] : YES.
(s detekcí konsekventních formulí)

Příklad 28.

Výchozí formule (axiomy) :
F0 : a(X). **F1 : b(X).**
F2 (¬dotaz) : ¬[a(X)∧b(X)].

R0 [F2&F1] : ¬a(X).
[R0&F0] : YES.
R1 [F2&F0] : ¬b(X).
[R1&F1] : YES.

V dalším příkladu se pokusíme dokázat tvrzení, které zjevně nemůže vyplývat z F0 a F1, neboť neobsahuje hledaný atom c. Nesmíme však zapomenout, že pokud je množina výchozích tvrzení nekonzistentní, lze z ní odvodit cokoliv! Tento příklad jasně připomíná základní pravidlo dedukce – tedy že sporná teorie implikuje všechny formule. Ukáže jim, že musí být opatrní při formulaci výchozích tvrzení a dbát na jejich vzájemnou konzistenci.

Příklad 29.

Výchozí formule (axiomy) :
F0 : a∧¬b.
F1 : ¬a∧b.
F2 (¬dotaz) : ¬c.

Nejprve se pokusíme využít lineární strategie, která nevede k popření, neboť začíná od dotazu, který v tomto případě nemůže rezolvovat s žádným axiomem. Abychom mohli dospět k popření pomocí lineární strategie, musíme použít výše zmíněnou modifikovanou strategii. Nejpřímější cestou je však použití strategií do hloubky.

Příklad 30.

R0 [F1&F1] : b.
[F1&F0] : YES.

[a∧¬b]∨[¬a∧b].
?-¬a∧¬b.
Řešení: ?- ¬[¬a∧¬b].
[F1&F1] : ¬[⊤∧¬b]∨¬[⊥∧¬b].
[F1&F0] : ¬[⊤∧¬b]∨ ⊤∧¬b∨ ⊥∧b.
[F1&F0] : ¬[¬a∧ ⊤]∨a∧ ⊥∨¬a∧ ⊤.

Na detailním zobrazení nezjednodušených formulí můžeme vidět, že nelze pro tuto situaci vytvořit smysluplnou rezolventu.

Výchozí formule (axiomy) :
F0 : a∧¬b∧c∧d∨¬a∧b∧¬c∧d.
F1 (¬query) : ¬[¬a∧¬b].

_____

R0 [F1&F0] : b∨¬b∧c∧d.
R1 [F1&F0] : a∨¬a∧¬c∧d.
R2 [F0&F0] : b∧¬c∧d∨¬b∧c∧d.
R3 [F0&F0] : ¬a∧¬c∧d∨a∧c∧d.
R4 [F0&F0] : ¬a∧b∧d∨a∧¬b∧d.
R5 [F0&F0] : a∧¬b∧c∨¬a∧b∧¬c.

Další příklad opět ukazuje, jak bude vypadat generování rezolvent, pokud nelze najít popření – tedy z axiomů formule nevyplývá.

Výchozí formule (axiomy) :
F0 : a→b∧g. F1 : b∧g→c. F2 : b∧g→a. F3 : c→b∧g.

F4 (¬dotaz) : ¬[a↔c].

R0 [F4&F3] : a∨b∧g.
R1 [R0&F4] : b∧g∨¬c.
R2 [R1&F2] : ¬c∨[g→a].
R3 [R2&F1] : [g→a]∨¬[b∧g].
R4 [R3&F4] : ¬g∨¬[b∧g]∨¬c.
R5 [R4&F3] : ¬b∨¬c.
R6 [R5&F4] : ¬b∨a.
R7 [R6&F3] : a∨¬c.
R8 [R7&F4] : ¬c.
R9 [R8&F4] : a.
R10 [R9&F0] : b∧g.
R11 [R10&F1] : g→c.
R12 [R11&F4] : ¬g∨¬a.
R13 [R12&F0] : ¬a.
R14 [R13&F4] : c.
[R14&R8] : YES.
Čas : 0.48 s.

Výchozí formule (axiomy) :
F0 : a↔b∧g. F1 : b∧g↔c. F2 (¬dotaz) : ¬[a↔c].

R0 [F2&F1] : [¬a∨¬[b∧g]]∧[a∨b∧g].
R1 [R0&F2] : ¬[b∧g]∨c.
R2 [R1&F0] : ¬g∨c∨¬a.
R3 [R2&F2] : ¬g∨¬a.
R4 [R3&F2] : ¬g∨c.
R5 [R4&F0] : c∨¬a.
R6 [R5&F2] : ¬a.
R7 [R6&F2] : c.
R8 [R7&F1] : b∧g.
R9 [R8&F1] : g↔c.
R10 [R9&F2] : [g∨a]∧[¬g∨¬a].
R11 [R10&F1] : a∨[b↔c].
R12 [R11&F1] : a∨¬c∨¬g.
R13 [R12&F2] : ¬c∨¬g.
R14 [R13&F2] : ¬g∨a.
R15 [R14&F1] : a∨¬c.
R16 [R15&F2] : ¬c.
R17 [R16&F2] : a.
[R17&R6] : YES.
Čas: 0.54 s.

Příklad právě uvedený ukazuje silně neklauzulární rezoluci. Jak je uvedeno, důkaz formule **a↔c** může být proveden, jak implikativními axiomy, tak ekvivalenčními axiomy. Pokud bychom použili klasický důkaz s pomocí klauzulárního systému, vzniklo by nám obrovské množství formulí při převodu do klauzulární formy, které by značně zpomalily proces dokazování.

F0 : [a(X)∧g(X)→b(X)].
F1 : a(a).
F2 : g(c).
F3 (¬dotaz) : ¬b(a).

R0 [F3&F0] : ¬[a(a)∧g(a)].
R1 [R0&F1] : ¬g(a).
Čas : 0.06 s.

b(a) v tomto příkladě nelze dokázat, neboť g platí pouze pro konstantu c. Na tomto příkladě si studenti mohou uvědomit, jak důležitá je unifikace pro nevýrokové příklady. Uvidí, že je nejprve nutné substituovat do proměnných termy, které jsou stejné, aby mohla proběhnout rezoluce.

Výchozí formule (axiomy) :
F0 : [a(X)∧g(X)→b(X)].
F1 : a(a).
F2 : g(c).
F3 : g(a).
F4 (¬dotaz) : ¬b(a).

R0 [F4&F0] : ¬[a(a)∧g(a)].
R1 [F3&F0] : [a(a)→b(a)].
R2 [F2&F0] : [a(c)→b(c)].
R3 [F1&F0] : [g(a)→b(a)].

R4 [R3&F4] : ¬g(a). **R5 [R3&F3] : b(a).**
R6 [R1&F4] : ¬a(a). **[R6&F1] : YES.**
Čas : 0.27 s.

V předchozím příkladě jsou zajímavé rezolventy s implikací, které ukazují názornost důkazu s pomocí zobecněné rezoluce. Není nutné je převádět do normální formy a díky tomu si rezolventy zachovávají svou expresivitu.

Výchozí formule (axiomy) :
F0 : [a(X)∧g(X)→b(X)]. **F1 : [b(X)∧g(X)→c(X)].**
F2 : a(a). **F3 : g(a).**
F4 (¬dotaz) : ¬c(Y).

R0 [F4&F1] : ¬[b(Y)∧g(Y)]. **R1 [F3&F1] : [b(a)→c(a)].**
R2 [F3&F0] : [a(a)→b(a)]. **R3 [F2&F0] : [g(a)→b(a)].**
R4 [F1&F0] : [[g(X)→c(X)]∨¬[a(X)∧g(X)]].
R5 [R4&F4] : [¬g(X)∨¬[a(X)∧g(X)]]. **R6 [R4&F3] : [c(a)∨¬a(a)].**
R7 [R3&F3] : b(a). **R8 [R3&F1] : [¬g(a)∨c(a)].**
R9 [R3&R0] : ¬g(a). **R10 [R1&F4] : ¬b(a).**
R11 [R1&R7] : c(a).
[R11&F4] : YES. **Y = a.**
[R10&R7] : YES. **Y = a.**
R12 [R10&R2] : ¬a(a).
[R9&F3] : YES. **Y = a.**
[R7&R10] : YES. **Y = a.**
[R12&F2] : YES. **Y = a.**
Čas : 1.72 s.

Tento příklad je prvním, které v dotazu vyžadují přiřazení do proměnné. Je zde pět možných popření a všechny jsou možné pouze při přiřazení konstanty a. Tato možnost, kterou GERDS poskytuje je zásadní pro mnoho didaktických příkladů. Uvidíme v následujících ukázkách, že díky jí ukážeme zajímavé úlohy z matematiky.

Výchozí formule (axiomy) :
F0 : [a(X)→a(X+1)]. **F1 : a(0).**
F2 (¬dotaz) : ¬a(5).

R0 [F1&F0] : a(1). **R1 [R0&F0] : a(2).**
R2 [R1&F0] : a(3). **R3 [R2&F0] : a(4).**
R4 [R3&F0] : a(5). **[R4&F2] : YES.**
Čas : 0.05 s.

GERDS umožňuje i použití infixních operátorů, které se vyhodnocují. V příkladě se realizuje pomocí implikace jednoduchá inkrementace celočíselné hodnoty.

Výchozí formule (axiomy) :
F0 : 1<2. **F1 : 2<3.**
F2 : 3<4. **F3 : 4<5.**
F4 : 5<6. **F5 : [X<Y∧Y<Z→X<Z].**
F6 (¬dotaz) : ¬1<5.

R0 [F6&F5] : ¬[1<Y∧Y<5]. **R1 [R0&F3] : ¬1<4.**
R2 [R1&F5] : ¬[1<Y∧Y<4]. **R3 [R2&F2] : ¬1<3.**
R4 [R3&F5] : ¬[1<Y∧Y<3]. **R5 [R4&F1] : ¬1<2.**

[R5&F0] : YES.
Čas : 0.22 s.

Tento důkaz ilustruje možnosti modelování tranzitivity v systému GERDS – konkrétně jde o relační operátor <. Další příklad demonstruje použití různých typů datových typů.

Výchozí formule (axiomy) :
F0 : [a(X)∧g(Z)→b(X)]. **F1 : [b(X)∧g(Z)→c(X)].**
F2 : a(a). **F3 : g(30).**
F4 (¬dotaz) : ¬[c(Y)∨g(Y)∨b(Y)].

R0 [F4&F4] : ¬[g(Y)∨b(Y)]. **R1 [F4&F4] : ¬[c(Y)∨b(Y)].**
R2 [F4&F4] : ¬[c(Y)∨g(Y)].
[F4&F3] : YES. **Y = 30.**
[F3&F4] : YES. **Y = 30.**
R3 [F3&F0] : [a(X)→b(X)]. **R4 [R3&F4] : ¬a(X).**
R5 [R3&F2] : b(a).
[R2&F3] : YES. **Y = 30.**
[R1&R5] : YES. **Y = a.**
[R0&F3] : YES. **Y = 30.**
[R0&R5] : YES. **Y = a.**
[R5&F4] : YES. **Y = a.**
[R5&R1] : YES. **Y = a.**
[R5&R0] : YES. **Y = a.**
[R4&F2] : YES. **Y = a.**
Čas : 0.54 s.

Příklad, který lze nyní zkoumat se studenty, je pro ně velice vhodný a má motivační prvky. Ukazuje jim poměrně jednoduchou Fibonacciho posloupnost, modelovanou s pomocí predikátové logiky:

Výchozí formule (axiomy) :
F0 : [f(I, A) ∧ f(I+1, B) → f(I+2, A+B)].
F1 : f(0, 1). **F2 : f(1, 1).**
F3 (¬dotaz) : ¬f(15, X).

R0 [F2&F0] : [f(2, B)→f(3, 1+B)]. **R1 [F1&F0] : [f(1, B)→f(2, 1+B)].**
R2 [R1&F2] : f(2, 2). **R3 [R0&R2] : f(3, 3).**
R4 [R3&F0] : [f(4, B)→f(5, 3+B)]. **R5 [R2&F0] : [f(3, B)→f(4, 2+B)].**
R6 [R5&R3] : f(4, 5). **R7 [R4&R6] : f(5, 8).**
R8 [R7&F0] : [f(6, B)→f(7, 8+B)]. **R9 [R6&F0] : [f(5, B)→f(6, 5+B)].**
R10 [R9&R7] : f(6, 13). **R11 [R8&R10] : f(7, 21).**
R12 [R11&F0] : [f(8, B)→f(9, 21+B)].
R13 [R10&F0] : [f(7, B)→f(8, 13+B)].
R14 [R13&R11] : f(8, 34). **R15 [R12&R14] : f(9, 55).**
R16 [R15&F0] : [f(10, B)→f(11, 55+B)].
R17 [R14&F0] : [f(9, B)→f(10, 34+B)].
R18 [R17&R15] : f(10, 89). **R19 [R16&R18] : f(11, 144).**
R20 [R19&F0] : [f(12, B)→f(13, 144+B)].
R21 [R18&F0] : [f(11, B)→f(12, 89+B)].
R22 [R21&R19] : f(12, 233). **R23 [R20&R22] : f(13, 377).**
R24 [R23&F0] : [f(14, B)→f(15, 377+B)].
R25 [R22&F0] : [f(13, B)→f(14, 233+B)].
R26 [R25&R23] : f(14, 610). **R27 [R24&R26] : f(15, 987).**

[R27&F3] : YES. **X = 987.**
Čas : 1.29 s.

Nalezli jsme patnácté Fibonacciho číslo, které je přiřazeno do proměnné dotazu (987). Další motivační příklad modeluje funkci faktoriál, vypočetli jsme zde faktoriál čísla 10:

F0 : [f(X, Y)→f(X+1, Y*(X+1))]. **F1 : f(1, 1).**
F2 (¬dotaz) : ¬f(10, Y).

R0 [F1&F0] : f(2, 2). **R1 [R0&F0] : f(3, 6).**
R2 [R1&F0] : f(4, 24). **R3 [R2&F0] : f(5, 120).**
R4 [R3&F0] : f(6, 720). **R5 [R4&F0] : f(7, 5040).**
R6 [R5&F0] : f(8, 40320). **R7 [R6&F0] : f(9, 362880).**
R8 [R7&F0] : f(10, 3628800).
[R8&F2] : YES. **Y = 3628800.**

Zobecněná rezoluce má také velmi důležitý dopad na dokazování kvantifikovaných formulí. Problém převodu do klauzulární formy zahrnuje také skolemizaci existenčních proměnných. Systém GERDS tuto skolemizaci nevyžaduje, což je další výhoda.
Nejprve se podívejme na důkazy jednoduchých problémů záměny pořadí kvantifikátorů, které jiné metody řeší velice obtížně.

F0: ∀X∃Y p(X,Y).
?-∃Y∀X p(X,Y). ?-∃Y∃X p(X,Y). ?-∀X∃Y p(X,Y). ?-∀Y∀X p(X,Y).
Řešení #1.
Řešení #2.
[F1&F0] : YES.
Řešení #3.
[F1&F0] : YES.
Řešení #4.

Případy 1 a 4 nemají řešení, neboť nevyplývají z F0. Opačný případ již vyplývá, neboť případ 2 je speciální případ F0 a 3 je přímo F0. Příklad s existencí na první místě je opět správně vyřešen:

∃Y∀X p(X,Y).
?-∀X∃Y p(X,Y). ?-∃X∃Y p(X,Y). ?-∀X∀Y p(X,Y).
Řešení #1.
[F1&F0] : YES.
Řešení #2.
[F1&F0] : YES.
Řešení #3.

Případ 1 je dokazatelný, protože existuje-li jedno fixní Y pro X, pak triviálně je toto Y tímto pro vybrané X. Případ 2 je speciálním případem F0.

∃Y∃X p(X,Y).
?-∀X∃Y p(X,Y). ?-∃X∃Y p(X,Y). ?-∀X∀Y p(X,Y).
Řešení #1.
Řešení #2.
[F1&F0] : YES.
Řešení #3.

Poslední případ ukazuje nejjednodušší případ, kdy testujeme existenci obou proměnných. Je jasné, že jediný důsledek je druhý dotaz. Uvedené příklady s existencí je velice důležité studentům správně vysvětlit. Může to značně zlepšit jejich pojetí interpretace existence v predikátové logice.

∀Z∃Y p(X,Y,Z). **?- ∀Z∃Y p(X,Y,Z).**
[F1&F0] : YES.

Unifikační algoritmus je podle dalšího příkladu schopen pracovat i s více proměnnými v jednom predikátu v souladu se svázáním s ostatními proměnnými.

∀Z∃Y p(X,Y,Z). **?- ∃Y∀Z p(X,Y,Z).**
Řešení #1.

∃Y∀Z p(X,Y,Z). **?- ∀Z∃Y p(X,Y,Z).**
Řešení #1.
[F1&F0] : YES.

Pro správné fungování unifikace je také důležité, zda pracuje správně při dosazování za existenční proměnné.

∀X a(X) → ∃X b(X). ∀X a(X).
?- b(a).
Řešení ¬b(a).

∃X a(X)→∀X b(X). a(a).
?- b(c).
Řešení ¬b(c).
R0 [F2&F0] : ∀X ¬a(X).
[R0&F1] : YES.

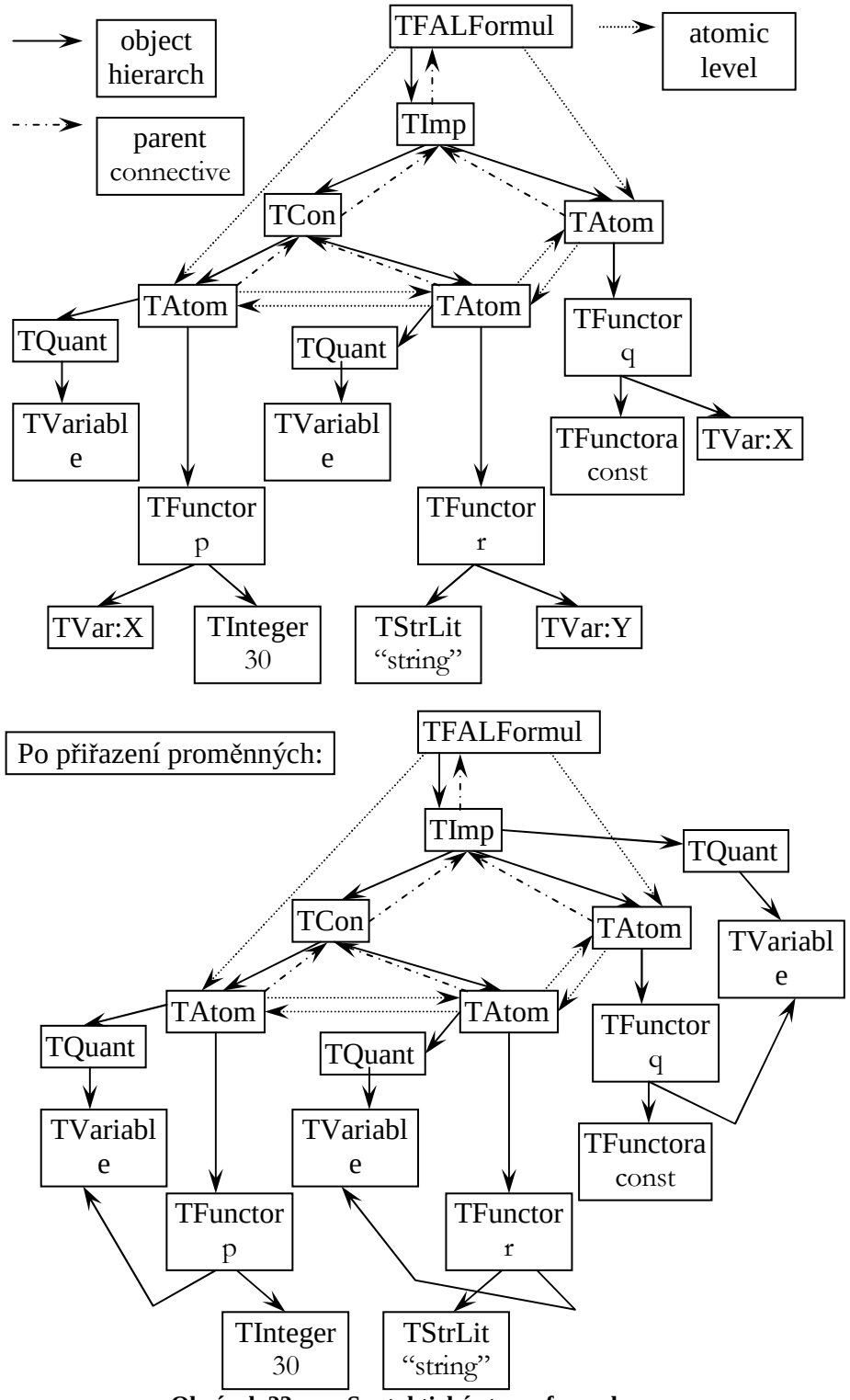
Pro vývoj systému GERDS bylo použito vývojového prostředí Borland Delphi 2 s jazykem Object Pascal. Právě GERDS je jako programátorská pomůcka nejvyvinutějším produktem. Nejde jen o možnost pracovat s ním interaktivně, ale pro výuku informatiků je neocenitelná možnost pracovat se zdrojovými kódy aplikace, které jim umožňují provést již zmíněný specifický a nespecifický transfer znalostí a dovedností. Jde nejen o znalosti teoretické z logiky a teorie formálních jazyků, ale také o znalosti a dovednosti z oblasti implementace překladačů, rezolučních strategií apod. Nyní rozebereme pouze ilustrační možnosti využití tohoto produktu jako pomůcky pro tuto výuku na zdrojových kódech aplikace.

Formule predikátové logiky v programu GERDS reprezentuje obecný objekt typu TSub (podformule).

```
TSub = class
   Neg : boolean;  { Flag, který označuje, že podformule je negována }
   Ev : shortint;  { Indikátor logické hodnoty formule }
   Pol : shortint; { Hodnota polarity formule }
   L : TSub;
   R : TSub;       { Odkaz na levý a pravý podstrom formule }
   Ac : TObject;   { Nadřazený objekt - formule }
   Q : TQuant;     { Kvantifikační část podformule – odkaz na objekt typu
kvantifikátor }
   ...
end;
```

TSub je základní třídou pro její potomky – TCon, TDis, TImp, TEqv a TAtom, které reprezentují podformule s konkrétní logickou spojkou nebo atomem. Každá podformule má referenci na levou a pravou podformuli. Ev může obsahovat tři hodnoty: -1 – false, 1 – true, 0 – pokud není formule prozatím evaluována. Ac obsahuje ukazatel na nadřízený uzel (kořen stromu formule odkazuje na objekt typu TFALFormula. TAtom má již speciální prvky, jelikož obsahuje složitější strukturu.
Studenti nejlépe pochopí celou hierarchickou strukturu objektů a potažmo i logických formulí z následujícího příkladu syntaktického stromu formule.
Uvažujme následující formuli

∀X p(X,30) ∧ ∃Y r("string",Y) → q(const,X).



Obrázek 23: Syntaktický strom formule

Příklad objektové hierarchie syntaktického stromu ukazuje, jak jsou proměnné přiřazeny příslušným výskytům. Po přiřazení vznikne nový kvantifikátor, jenž kvantifikuje volnou proměnnou X. Atomy jsou svázány pomocí ukazatelů, což umožňuje mnohem efektivnější práci při hledání možných rezolucí než by bylo procházení celého stromu rekurzivně.

Syntaktický analyzátor a kompilátor

Analyzovali jsme datové struktury, především syntaktický strom a třídy, které jej tvoří. Syntaktický analyzátor má za úkol rozpoznat, zda logický program –

seznam formulí a dotazů je správně zapsán, a má provést převod do formy stromu. Hlavní metodou je XComp.

```
procedure TPLProgram.XComp;
 begin
  ...
  StackInit;
  cpos := 0; dontcare := 0;Error := OK;Getchar;
  if (ch = '?')and(infix[cpos] = '-') then
  begin
   query := true;GetCharI; GetChar;
  end;
  if Error = OK then Eqv;  { Go to parser / generator. }
  ...
  if Error = endcomp then
   begin
     Error := OK;sub := TSub(VPSStack^.Node);dispose(VPSStack);
     if query then F := TQuery.Create(self)
     else F := TFALFormula.Create(self);
     F.Cont := sub;sub.Ac := F;F.MF := FirstMol;F.ML := LastMol;
     F.PostComp;GetChar;
     if ch <> #0 then LocalPos := cpos-1 else LocalPos := cpos;
   end
  else
    begin
      if VPSStack <> nil then DestroyNode(VPSStack^.node);
     DisposeStack;LocalPos := cpos-inum-1;
    end;
 err := error;
 case error of
  ...
 end;
end;
```

Na začátku tohoto algoritmu je třeba správně inicializovat zásobník, jenž se používá k vytváření stromu. Dokud nejsou prvky stromu přiřazeny správným nadřazeným objektům a podformulím, jsou zde uloženy. GetChar (resp. GetCharI) nastavuje správně aktuální lexikální element do proměnné ch, přičemž ignoruje „prázdné“ znaky (resp. bere v úvahu prázdné znaky, pokud je to u některých syntaktických struktur potřeba). Potom rozhoduje o tom, zda formule je dotazem nebo ne. Dalším krokem je rekurzivní sestup na základě gramatiky jazyka, o němž budeme detailně diskutovat v dalších odstavcích. Na konci procedura identifikuje chyby. Pokud k žádným nedošlo, je vytvořena formule ve formě objektu typu TFALFormula.

Metoda rekurzivního sestupu, kterou používá analyzátor, spočívá v konstrukci procedur strukturovaného programovacího jazyka přesně dle Backusovy-Naurovy formy (BNF), kde je každému neterminálu přiřazena jedna procedura a je volána procedura GetChar před každým terminálem. Výskyt neterminálu v pravidle je v proceduře nahrazen rekurzivně voláním příslušné procedury. Iterace a podmínka v Backus-Naurově formě je nahrazena jednoduše jejich programátorskými protějšky. Vyžaduje se, aby jazyk byl definován LL(k) gramatikou. V tomto případě jde o LL(1) gramatiku definovanou následovně:

<Formula> ::= <Imp> { ↔ <Imp> }
<Imp> ::= <Dis> { → <Dis> }
<Dis> ::= <Con> { ∨ <Con> }
<Con> ::= <Subformula> { ∧ <Subformula> }

<Subformula> ::= ¬ <Subformula> | <Quantifier section> <Subformula> | '[' <Formula> ']' | <Predicate>
<Quantifier section> ::= <Quantifier character> <Variable> { , <Variable> } {<Quantifier character> <Variable> { , <Variable> } }
<Predicate> ::= <Predicate name> <List of parameters> | <Term> <InfixPred> <Term>
<Term> ::= <Term2> { <+/- operator> <Term2> }
<Term2> ::= <Base> { <*// operator> <Term2> }
<Base> ::= <Variable> | <Function> | <StrLit> | <Number> | + <Number> | - <Number> | (<Term>)
<Functor> ::= <Lower case> {<Alphanumeric>}
<Predicate name> ::= <Lower case> {<Alphanumeric>}
<List of parameters> ::= { (<Term> { , <Term> }) }
<Function> ::= <Functor> <List of parameters>
<Variable> ::= <Upper case> {<Alphanumeric>}
<Number> ::= <Integer> {. <Integer>} { e <+/- operator> <Integer>}
<StrLit> ::= " {<Alphanumeric>} "
<InfixPred> ::= ≥ | ≤ | < | > | = | ≠
<Quantifier character> ::= ∀ | ∃
<+/- operator> ::= + | - , <*// operator> ::= * | /
<Lower case> ::= a | .. | z, <Upper case> ::= A | .. | Z | _
<Alphanumeric> ::= <Lower case> | <Upper case> | <Numeric> , <Integer> ::= <Numeric> { <Numeric> } , <Numeric> ::= 0 | .. | 9

```
procedure Eqv;
begin
 if Error = OK then
   begin
     Imp;
     while (ch = ceqv)and(error = OK) do
       begin
        Getchar;
        Imp;
        if error = OK then AssignSub(TEqv.Create);
       end;
   end;
end;
```

Procedura Eqv přísluší neterminálu <Formula>, který reprezentuje podformuli se spojkou ekvivalence (s nejmenší prioritou). Tato procedura volá Imp a pak pokračuje libovolněkrát opět voláním Imp podle toho kolikrát se vyskytuje znak ekvivalence, přesně podle definice gramatiky. Každá ekvivalence je tak vytvořena a jsou jí přiřazeny podřízené formule do položek L a R pomocí procedury AssignSub.

```
procedure AssignSub(Ob : TObject);   { Přiřazuje formuli její podstromy }
begin
  Put(Ob);pom := Cut;
  TSub(pom^.node).R := TSub(VPSStack^.node);
  TSub(VPSStack^.node).Ac := pom^.node;
  garbage := Cut;dispose(garbage);
  TSub(pom^.node).L := TSub(VPSStack^.node);
  TSub(VPSStack^.node).Ac := pom^.node;
  garbage := Cut;dispose(garbage);SInsert(pom);
end;
```

Procedura Put vytváří objekt zásobníku, jenž zapouzdřuje tuto položku během jejího setrvání v zásobníku, než je přiřazen jeho nadřízenému objektu. Potom je vybrán ze zásobníku pomocí procedury Cut a do zásobníku je pak uložen nadřízený objekt. Celá procedura se realizuje tak dlouho, dokud není vytvořen uzel na nejvyšší úrovni. Procedura ICE, přiřazená neterminálu <SubFormula>, plně demonstruje rekurzivní charakter tohoto algoritmu, neboť sama volá opět Eqv.

```
procedure ICE;
begin
if error = OK then
 begin
  case ch of
   cneg : begin
         Getchar; ICE;NegateF;
        end;
   cforall, cexists :
        begin
          Quant; ICE;
          if Error = OK then AssignQtoSub;
        end;
   'a'..'z', 'A'..'Z', '_', '0'..'9', '"', '(', '+', '-' :
        begin
          Atom;
        end;
    '[' : begin
          GetChar; Eqv;
          if (ch <> ']')and(error = OK) then Error := missbra;
          Getchar;
        end
   else if error = OK then Error := missexp;
   end;
 end;
end;
```

Případ pro cneg reprezentuje negovanou podformuli, volajíce sama sebe. Další případy řeší ostatní možné syntaktické struktury. Použili jsme pouze některé ze syntaktických struktur v tomto textu, neboť gramatika je mnohem složitější. Každý student si však může celou gramatiku a její programátorské vyjádření najít ve zdrojovým textech aplikace.

Shrňme nejpodstatnější témata:

- predikátová logika a vyjadřování znalostí pomocí ní,
- rezoluční princip, jeho podstata a zobecněná verze,
- rezoluční zamítnutí a jeho strategie (prohledávání stav. prostoru, podpůrné strategie),
- důkazy aplikovaných problémů.

Důležité však je zejména to, že GERDS je také programátorskou pomůckou. To znamená, že slouží nejen k procvičování teoretických témat, ale s pomocí řešení algoritmických problémů a jejich implementace poskytuje obousměrný transfer mezi teoretickou informatikou a algoritmizací. Studenti si, jinak řečeno, osvojí způsoby implementace vybraných problémů a zároveň jsou lépe motivováni k pochopení vlastní teorie. Shrňme nyní základní moduly.

- **objektová hierarchie** (objektový model formule), například specifikace dědičnosti u obecné formule TSub a potomka TImp apod.

- **překladač** (kompilátor formulí do formy stromu), implementace metody rekurzivní sestupu a generování vnitřní reprezentace formule apod.

- **inferenční proces** (řízení procesu rezolučního dokazování), strategie, omezování rezolvent, nepřímé dokazování;

- **rezoluční důkazy** (rezoluční pravidlo, výběr premis)

5.2 Program Prover9

Prover9 je program pro automatické dokazování vět, a zároveň nadstavbou programu Otter theorem prover. Oba jsou založeny na rezoluci, kdy hledáme spor v množině klauzulí zapsané v jazyce predikátové logiky prvního řádu, ale i v obecných klauzulích obsahující disjunkce literálů. Na rozdíl od Otteru, Prover9 nerozlišuje pojem klauzule a formule, kde formule může nyní obsahovat volné proměnné a klauzule je podmnožinou formulí.

Další výhodou tohoto programu je, že můžeme přímo zadávat cíl jako formuli, kterou program sám zneguje a jako výsledek vrátí vždy kontradikci, je-li důkaz proveden. Prover9 automaticky generuje důkaz ve výchozím nastavení na rozdíl od svého předchůdce, kde se musely všechny předvolby nastavovat.

Jak již bylo řečeno, program vždy vygeneruje kontradikci, i když nebyl žádný cíl zadán, hledá spor v zadaných předpokladech. Naopak nemusíme zadat žádné předpoklady, pokud jsou stanoveny v cíli. Pokud z uvedených předpokladů nebo cílů nevyplívá žádný závěr (kontradikce)
Syntaxí programu máme na mysli strukturu vstupního souboru, která se musí dodržovat pro správnou funkci programu. Jak je uvedeno už výše, klauzule jsou podmnožinou formulí a formule mohou obsahovat volné proměnné. Proměnné obecně budeme zapisovat pomocí malých písmen, nejčastěji v rozmezí u – z. Budeme-li mít formuli tvaru R(a,u), pak term „a“ označuje konstantu a „u“ je proměnnou. Volné proměnné jsou považovány za proměnné s univerzálním kvantifikátorem. Odvozovací pravidla pracují s formulemi v klauzulárním tvaru, pokud však vložíme neklauzulární formuli, program ji transformuje pomocí Skolemizace, negativní či konjunktivní normální formy, na klauzulární tvar.

Vstupní soubor obsahující veškeré informace nutné ke zpracování úlohy, je ve formě textového dokumentu. Při zápisu formulí nebo klauzulí musíme dodržovat následující konvence:

Symbol	Význam	Příklad
-	negace	-p.
&	konjunkce	p & q.
|	disjunkce	p | q.
→	implikace	p → q.
↔	ekvivalence	p ↔ q.

Tabulka 4. Pravidla pro zápis logických spojek

Zápis	Význam	Příklad
All	univerzální kvantifikátor	all x (p(x)).
Exists	existenční kvantifikátor	exists x (p(x)).

Tabulka 5. Pravidla pro zápis kvantifikátorů

- Každá formule nebo klauzule je ukončena tečkou.
- Komentář se označuje symbolem %.
- Predikát rovnosti je znázorněn =, zatímco predikát nerovnosti !=.

Klauzule a formule se zapisují do seznamů, tímto rozdělením se urychlí nalezení důkazu, ale také to pomůže k lepší orientaci v programu.

Předdefinovanými seznamy jsou:

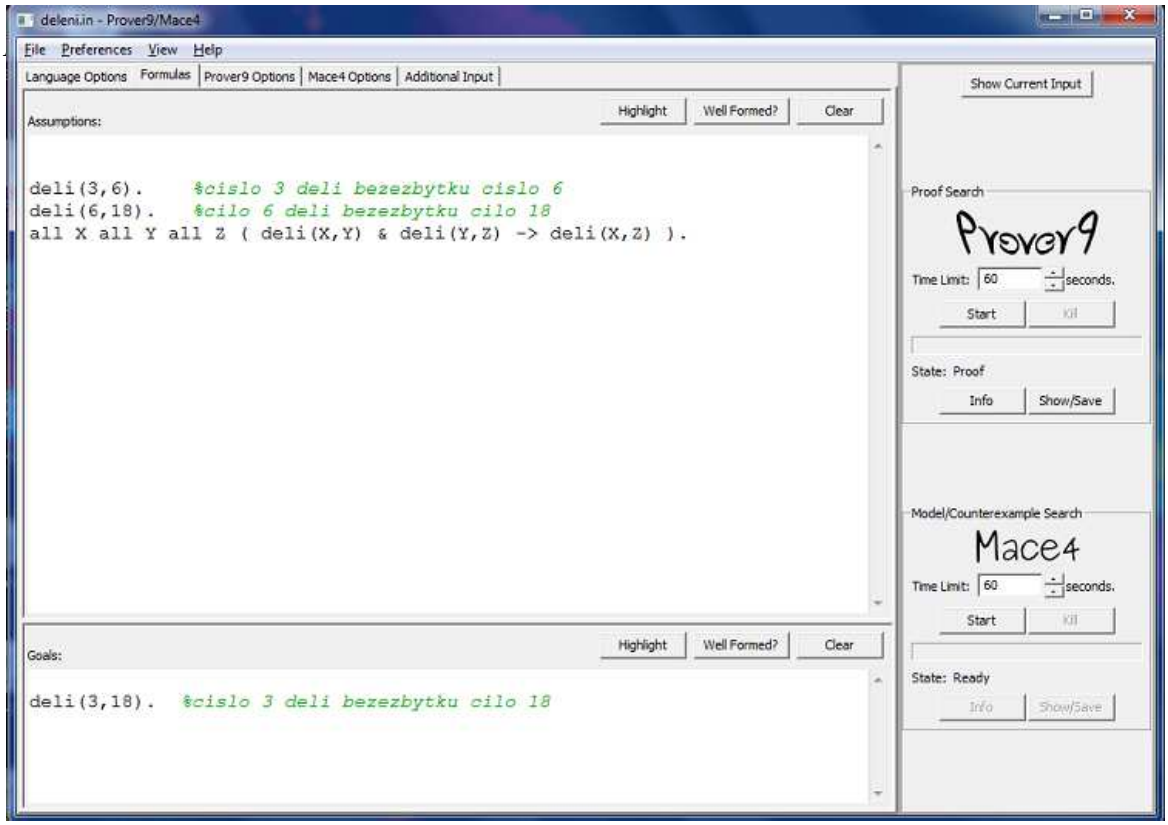
- `SOS`(set-of-support) seznam - obsahující klauzule čekající na inferenci s jinými klauzulemi,
- `usable` seznam - obsahující klauzule, které sem byly před inferencí přesunuty ze seznamu SOS a nově odvozené klauzule vzniklé inferencí.
- `assumptions` seznam – obsahující seznam přepokladů

Vstupní soubor může vypadat následovně:

```
formulas(assumptions).      % začátek seznamu předpokladů
  -muz(x) | smrtelnik(x).
  muz(jan).
end_of_list.                % konec seznamu předpokladů

formulas(goals).            % začátek seznamu cílů
smrtelnik(jan).
end_of_list.                % konec seznamu cílů
```

Pokud zapisujeme příklad rovnou do programu, nemusíme uvádět začáteční ani koncové popisy (`formulas ( ).` a `end_of_list.`), zadáme pouze předpoklady a cíle do příslušného okna Assumptions a Goals, viz. obrázek 5.1

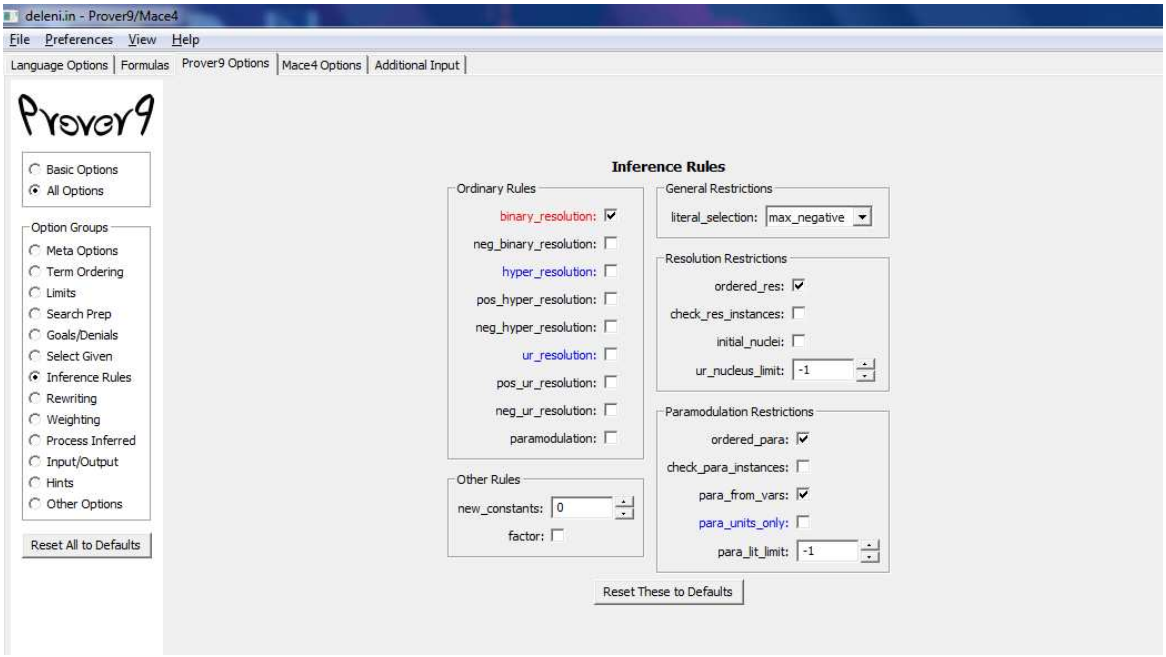


Obrázek 24: Prover9

Veškerá nastavení, pomocí nichž program dojde ke zpracování úlohy, lze zadat přímo ve vstupním souboru, a to příkazy `set( ), clear ( )` a `assingn( ).` Celé nastavení je uzavřeno v příkazu `if.`

```
if(Prover9). % nastavení pro Prover9
  clear(auto).      %vypnutí    funkce    automatické
nastavení
  set(binary_resolution).     % zapnutí řešení pomocí
bin_res
  assign(max_seconds, 60).    % nastavení proměnné
                              max_seconds hodnotu 60
end_if.
```

Stejného nastavení dosáhneme i ručně, pomocí přehledného uživatelského prostředí, které přímo nabízí záložku Prover9 Options, kde si zatržením označíme, možnosti jakými se bude daná úloha řešit.



Obrázek 25: Nastavení aplikace.

Cílem programu je najít důkaz provedený sporem, jinak řečeno, znegováním zadaného cíle, získat spor daných předpokladů, který se ve výstupním souboru vyskytne jako symbol `$F`.

Výstupní soubor

```
========================                    prooftrans
============================
Obsahující verzi programu, datum, příkaz spouštějící
aplikaci
======================       end       of       head
============================

========================                        input
==============================
Obsahuje popisky, které nejsou označeny komentářem %.
======================       end       of       input
============================
==================     PROCESS     INITIAL     CLAUSES
==================
% Clauses before input processing:

formulas(usable).
end_of_list.

formulas(sos).
end_of_list.

formulas(demodulators).
end_of_list.
```

```
==================    end    of    initial    Clauses
==================

==========================                        PROOF
==========================
% -------- Comments from original proof --------
% Proof 1 at 0.01 (+ 0.00) seconds.
% Length of proof is 9.
% Level of proof is 4.
% Maximum clause weight is 9.
% Given clauses 6.

1 (all X all Y all Z (deli(X,Y) & deli(Y,Z) ->
deli(X,Z))) # label(non_clause).  [assumption].
2  deli(5,20)  #  label(non_clause)  #  label(goal).
[goal].
3 deli(5,10).  [assumption].
4 deli(10,20).  [assumption].
5    -deli(x,y)    |    -deli(y,z)    |    deli(x,z).
[clausify(1)].
6 -deli(5,20).  [deny(2)].
8 -deli(10,x) | deli(5,x).  [resolve(5,a,3,a)].
11 deli(5,20).  [resolve(8,a,4,a)].
12 $F.  [resolve(11,a,6,a)].
======================       end       of       proof
======================
```

Sekce mezi `PROOF a end of proof` obsahuje samotné odvození důkazu a statistiky, jako počet a čas nalezení důkazu, délka a úroveň důkazu, maximální váha a počet zadaných klauzulí.

Každá klauzule je v souboru označena integer ID a nastavenim, které může odkazovat na klauzuli s jiným ID, jde vlastně o seznam obsahující jeden primární krok a několik sekundárních kroků. Většina primárních kroků jsou odvozovací pravidla, jako hyper_resolution, atd. Sekundární kroky jsou pak zjednodušení, přepisování, atd.

Jak lze vidět na ukázce z výstupního souboru, tak některé kroky (např. 8) obsahují odkazy na pozice literálů nebo termů rodičovských klauzulí. Literály jsou označeny písmeny a, b, c… a znamenají pozici a (první literál), b(druhý literál), atd. Termy definujeme pomocí literálu a posloupností celých čísel udávající pozici argumentu, např. (`a, 1,3`), kde vybíráme první literál a v něm první a třetí argument.

Primární kroky:

- `assumption` – vstupní podmínka,
- `clausify(1)` – převod neklauzulární podmínky (první), do klauzulárního tvaru,
- `goal` – cíl, stanoven ve vstupním souboru,

- `deny` – znegování zadaného cíle (popřípadě převod do klauzulární formy),
- `resolve(11,a,6,a)` – rezoluce (binární) použitá na první literál 11. klauzule a první literál 6. klauzule,
- `hyper(5, a,3,a, b,4,a)` – hyper-rezoluční odvození, které se skládá z ID začínající klauzule a seznamu obsahující trojici `a,3,a,` (literál,ID,literál). Jde o posloupnost kroků binární rezoluce,
- `ur(5, a,3,a, b,4,a, c,2,a, d,1,a)` – unit-resulting rezoluce, kde seznam má stejný význam jako u hyper-rezoluce.

Příklad 5.1 – postup provedení důkazu

```
1 (all X all Y all Z (deli(X,Y) & deli(Y,Z) -> deli(X,Z))) # label(non_clause). [assumption].
2 deli(5,20) # label(non_clause) # label(goal). [goal].
3 deli(5,10). [assumption].
4 deli(10,20). [assumption].
5 -deli(x,y) | -deli(y,z) | deli(x,z). [clausify(1)].
6 -deli(5,20). [deny(2)].
8 -deli(10,x) | deli(5,x). [resolve(5,a,3,a)].
11 deli(5,20). [resolve(8,a,4,a)].
12 $F. [resolve(11,a,6,a)].
```

Popis provádění důkazu:

1. program nejprve vypíše formule (zadané ve vstupním souboru), které nejsou v klauzulárním tvaru a až pak zbylé předpoklady, které už v klauzulárním tvaru jsou (kroky 1 - 4),
2. převod do klauzulárního tvaru a znegování závěru (kroky 5,6),
3. samotný proces odvozování, v našem případě pomocí tří kroků binární rezoluce (kroky 8-11),
4. dosažení sporu `$F.`,důkaz byl nalezen (krok 12).

Základní cyklus pro odvození, zpracování klauzulí a získání důkazu se nazývá given clause algoritmus (určených klauzulí). Tento cyklus pracuje pouze se seznamy SOS a usable, kde v každé iteraci cyklu je given klauzule vybrána z SOS seznamu a přesunuta do usable seznamu. Odvození dosáhneme použitím given klauzule a další klauzule v usable seznamu. Pak pro každé odvození platí, že jedna z rodičovských klauzulí je given klauzule a ostatní jsou z usable seznamu.

Binární rezoluce – je standardní odvozovací pravidlo, které ze dvou klauzulí obsahující unifikované literály opačných znamének odvodí rezolventu, skládající se ze zbylých literálu,

$$\frac{p \vee l \quad q \vee \neg l}{(p \vee q)\sigma}$$

kde σ je nejobecnější unifikací (mgu) literálů l a $\neg l$.

Negativní binární rezoluce – je stejná jako pozitivní jen jedna z premis musí být negativní.

Uspořádané odvození – je omezením rezoluce, založené na uspořádání literálů, v mnohých případech je omezením výběr maximálního literálu. Selekce literálů je prováděna z hlavní premisy a jde o výběr negativních literálů.

Faktoring – odvozovací pravidlo, kde v jedné klauzuli jsou dva nebo více literálů se stejným označením (již unifikované). Mgu těchto literálů jsou shodné a sloučí se do jednoho.

Hyper-rezoluce – (pozitivní hyper-rezoluce) jde o více binárních rezolucí v jednom kroku a rezolventou je pozitivní klauzule. Obsahuje zvláštní (pozitivní) premisy nazývané *elektrony*, s nimiž se rezolvuje každý negativní literál z poslední (nepozitivní) premisy, nazývané *jádro* odvození.

Negativní hyper-rezoluce – je hyper-rezoluce, ale s opačným označením pozitivních a negativních premis.

UR-rezoluce – (unit resulting) je rezoluce, kde jádro musí obsahovat alespoň dva literály, ostatní elektrony musí mít po jednom literálu (jednotková klauzule). Výsledkem je jednotková klauzule.

Paramodulace - je odvozovací pravidlo zobecňující substituce na základě rovnosti. Paramodulace je aplikována vždy na dvojici klauzulí. Alespoň jedna klauzule z dvojice musí obsahovat pozitivní literál rovnosti, který lze unifikovat s částí druhé klauzule. Taková klauzule se nazývá ***from clause***. Klauzule, ve které je prováděna substituce, se nazývá ***into clause***.
Např. z klauzulí `equal(sum(0, X),X).`a `rozdil(sum(X, 1),sum(0, 1)).`, lze pomocí paramodulace odvodit klauzuli `rozdil(sum(X, 1), 1).`
V tomto případě je `equal ()` from klauzulí a `rozdil()` into klauzulí.

Subsumption – neboli vyřazování klauzulí, pokud je klauzule v seznamu klauzulí už obsažena nebo je obecnější, např. žena(x) je obecnější než žena(jana)
Forward subsumption - pokud nově odvozenou klauzuli vyřadíme na základě toho, že se v seznamu klauzulí již nachází obecnější klauzule.

Back subsumption – opačný případ, pokud je díky nově odvozené klauzuli ze seznamu vypuštěna dříve získaná klauzule.

Uspořádání je v Prover9 následující, nejvyšší prioritu mají logické symboly F/T, pak unární funkce a následně binární funkce.

Uspořádání termů může být třemi způsoby:

1. Lexikografické (LPO) – abecední uspořádání
2. Rekurzivní (RPO)
3. Knuth-bendix uspořádání (KBO)

- Fuzzy Predicate Logic Genealized Deductive Systém
- Prover9

Korespondenční úkol:

Vyberte si jednu z následujících úloh:

1. Implementujte vlastní deduktivní systém pro výrokovou logiku (GUI není povinné).
2. Najděte na webu jiný (nejen pro výrokovou logiku) deduktivní systém než jsou dva zmiňované v opoře a pokuste se ho zprovoznit (nejlépe na platformě Windows). Vytvořte několik příkladů a proveďte automatizované důkazy.

3. Vytvořte příklady pro GERDS, resp. FPLGERDS a to tak, aby byla vidět časová složitost vzhledem k velikosti znalostní báze a použité strategii. Tedy vytvořte např. 10 srovnatelných znalostních bází, které obsahují 20, 40, 60,… formulí a nejlépe do grafu vyneste čas nutný k nalezení důkazu.

Literatura

1. [Bachmair, 1997] Bachmair, L., Ganzinger, H. A theory of resolution. Tech. Rep.: Max-Planck-Institut, 1997.
2. [Bachmair, 2001] Bachmair, L., Ganzinger, H. Resolution theorem proving. In Handbook of Automated Reasoning, MIT Press, 2001.
3. [Baader, 2002] Baader, F., Nutt, W. Basic Description Logics. In the Description Logic Handbook, edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, Cambridge University Press, 2002, pages 47-100.
4. [Baader, 2003] Baader et col. (eds.). The Description Logic Handbook - Theory, Interpretation and Applications. Cambridge Univ. Press, 2003.
5. [Dukic, 2005] Dukic, N., Avdagic, Z. Fuzzy Functional Dependency and the Resolution Principle. In Informatica, Vilnius: Lith. Acad. Sci. (IOSPRESS), 2005, Vol.16, No. 1, pp. 45 - 60, 2005.
6. [Dvořák et. al, 2003] The concept of LFLC 2000 - its specificity, realization and power of applications. Computers in Industry. 03/2003(51), Elsevier, Amsterdam, 2003, pp.269-280.
7. [Habiballa, 2000] Habiballa, H. Non-clausal resolution - theory and practice. Research report: University of Ostrava, 2000, http://www.volny.cz/habiballa/files/gerds.pdf
8. [Habiballa, 2002] Habiballa, H., Novák, V. Fuzzy General Resolution. In Proc. of Intl. Conf. Aplimat 2002. Bratislava, Slovak Technical University, 2002. pp. 199-206, also available as research rep. at http://ac030.osu.cz/irafm/ps/rep47.ps
9. [Habiballa, 2005] Habiballa, H. Non-clausal Resolution in Fuzzy Predicate Logic with Evaluated Syntax (background and implementation). In Proc. of Intl. Conf. The Logic of Soft Computing IV, Ostrava, pp. 51 - 54, 2005, also available as research rep. at http://ac030.osu.cz/irafm/ps/rep70.ps.gz
10. [Habiballa, 2006] Habiballa, H. Resolution Based Reasoning in Description Logic. In Proc. of Intl. Conf. ZNALOSTI 2006, Univ. of Hradec Kralove, 2006, also available as research rep. at http://ac030.osu.cz/irafm/ps/rep66.ps.gz.
11. [Habiballa, 2006a] Habiballa, H. Fuzzy Predicate Logic Generalized Resolution Deductive System. Technical Report, Institute for Research and Application of Fuzzy Modeling, Univ. of Ostrava, 2006.
12. [Habiballa, 2009a] Habiballa, H. Bachelor – propositional logic rewrite systém. Dostupné z: http://www1.osu.cz/home/habibal/page9.html
13. [Habiballa, 2009b] Habiballa, H. GERDS. Dostupné z: http://www1.osu.cz/home/habibal/page8.html
14. [Habiballa, 2009c] Habiballa, H. FPLGERDS. Dostupné z: http://www1.osu.cz/home/habibal/page6.html
15. [Hájek, 2000] Hájek, P. Metamathematics of fuzzy logic. Kluwer - Dordrecht, 2000.
16. [Hájek, 2005] Hájek, P. Making fuzzy description logic more general. Fuzzy Sets and Systems 154(2005),pp. 1-15.
17. [Haarslev, 2006] Deng, X., Haarslev, V., Shiri, N. Resolution Based Explanations for Reasoning in the Description Logic ALC. Proceedings of the Canadian Semantic Web Working Symposium, June 6, 2006, Québec City, Québec, Canada, Series: Semantic Web and Beyond: Computing for Human Experience, Vol. 4, Springer Verlag, 2006, pp. 189-204.
18. [Hustadt, 1999] Hustadt, U., Schmidt., R.A. On the relation of resolution and tableaux proof systems for description logics. In T. Dean, editor, Proceedings

of International Joint Conference Articial Intelligence, pp. 110-115. Morgan Kaufmann, 1999.
19. [Kleene, 1967] Kleene S.C. Mathematical logic. John Willey & Sons, 1967.
20. [Lehmke, 1995] Lehmke, S. On resolution-based theorem proving in propositional fuzzy logic with ‘bold’ connectives. Diploma thesis:University of Dortmund, 1995
21. [Lu03] Lukasová, Alena. Formální logika v umělé inteligenci, Computer Press, Brno 2003, str. 177
22. [Lukasová, 2005] Lukasová, A. Reasoning in Description Logic with semantic tableua binary trees. Resarch report: Institute for Research and Applications of Fuzzy Modeling,Univ. of Ostrava, 2005.
23. [Murray, 1982] Murray, N. Completely non-clausal theorem proving. Artificial Intelligence, 18, 1982, 67-85.
24. [Novák, 1999] Novák, V., Perfilieva, I., Močkoř, J. Mathematical principles of fuzzy logic. Kluwer, 1999.
25. [Novák, 2005] Novák, V. Mathematical Fuzzy Logic In Narrow And Broader Sense – A Unified Concept. In Proc. BISCE’05, University of California, Berkeley, 2005.
26. [Ramsay, 1991] Ramsay, A. Formal Methods in Artificial Intelligence. (Cambridge Tracts in Theoretical Computer Science, Vol. 6), Cambridge press, 1991,279 pp.
27. [Straccia, 2001] Straccia, U. Reasoning within Fuzzy Description Logics. J. of AI Research 14 (2001), pp. 137-166.
28. [Stachniak, 1996] Stachniak, Z. Resolution Proof Systems: An Algebraic Theory, Kluwer Academic Publishers, 1996.
29. [Sutcliffe, 2002] Sutcliffe, G. Automated Theorem Proving: A Review. AI Magazine 23(1): (2002).
30. [Sutcliffe, 1998] Sutcliffe, G., Suttner C.B. The TPTP Problem Library: CNF Release v1.2.1, Journal of Automated Reasoning 21(2), pp.177-203., 1998.
31. [Wiedenbach, 1999] Wiedenbach, Ch. SPASS for Windows. http://spass.mpi-sb.mpg.de.
32. [Wiedenbach, 2001] Weidenbach, Ch. SPASS: Combining Superposition, Sorts and Splitting. In A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, Elsevier, 2001.