Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

Implementace důkazového systému pro výrokovou logiku

Jan Švajcr

Vedoucí práce: Mgr. Jan Starý, Ph.D.

26. června 2014

Poděkování

Děkuji svému vedoucímu práce Mgr. Janu Starému, Ph.D. za přívětivost a pomoc, své rodině za podporu a zázemí a své milované za lásku a věrnost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 26. června 2014 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Švajcr, Jan. *Implementace důkazového systému pro výrokovou logiku.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

Abstrakt

Cílem této práce je vypracovat terminálovou aplikaci implementující prostředí důkazového systému výrokové logiky. Její hlavní funkcionalitou je syntaktická analýza textového vstupu v podobě posloupnosti výrokových formulí a ověření, zdali je tato posloupnost korektním výrokovým důkazem. Software je implementován v jazyce C++ a je podporován prostředím UNIX. Součástí práce je dokumentace zdrojového kódu a uživatelská příručka v podobě standardní manuálové stránky.

Klíčová slova Výroková logika, Důkazový systém, Parsing, C++.

Abstract

The goal of this thesis is to create a command line application implementing the proof system environment of the propositional logic. It's main functionality is parsing a textual input representing a sequence of propositional formulas and verifying this sequence as a proof. The software is implemented in the C++ language and is supported by UNIX environment. Code documentation and a standard manual page as a user manual are also included.

Keywords Propositional logic, Proof system, Parsing, C++.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Logika je formální věda zkoumající část lidského myšlení. Jejím předmětem je správné vyvozování důsledků z předpokladů, jejichž volbu, pravdivost nebo snad smysl blíže nezkoumáme. Nečiníme tak nejen proto, že naše vyvození je správné i v případě, kdy předpoklady správné nejsou, ale i proto, že to této disciplíně ani nepřísluší. Matematická logika toto usuzování formalizuje, čímž nás oprošťuje od psychologického aspektu. Dává tak vzniknout postupům, které lze kdykoliv opakovaně aplikovat. Příkladem takového postupu je ověřování správnosti našeho usuzování, tzv. důkazu. To je dokonce natolik mechanické, že jej můžeme svěřit strojovému zpracování[1]. Právě tento aspekt výrokové logiky byl podnětem pro vznik této práce, která formalismus výrokové logiky implementuje počítačovým programem. Konkrétně implementuje principy Hilbertova systému. Ústřední kapitolou výrokového počtu, o kterou se budeme v rámci teoretické přípravy zajímat zejména, je *dokazatelnost*.

Ještě předtím, než se ponoříme do problematiky implementace, je třeba analyzovat vlastnosti příslušné oblasti výrokového počtu. V první kapitole proto vysvětlíme klíčové pojmy výrokové logiky, které nás budou provázet životním cyklem projektu a budou tak pro nás nutnou znalostí. Pokračovat budeme kapitolou, ve které upřesníme zadání projektu a vymezíme požadavky na implementaci. Následuje kapitola, ve které položíme základy pro vlastní implementaci. Budeme se zde věnovat rozkladu systému na menší celky a návrhu jejich řešení. K softwarové realizaci přejdeme v navazující kapitole zabývající se technickými detaily samotné implementace. Nastíníme zde význam datových struktur a popíšeme klíčové algoritmy. Předposlední kapitola se věnuje testování. Budeme se zde snažit maximálně pokrýt rizikové oblasti, které mohou ohrozit stabilitu aplikace. Práci zakončíme krátkým zamyšlením nad jejím možným pokračováním a prodiskutujeme její rozšiřitelné stránky.

Kapitola 1

Formální kontext

V úvodní kapitole se seznámíme s několika základními pojmy výrokového počtu a zavedeme terminologii užitou v tomto textu, abychom na ni mohli čtenáře později odkázat. Tato část slouží jako teoretický základ celé práce. Čtenář znalý výrokové logiky ji může vynechat. Standardní terminologii, značení a hlavní myšlenky přebíráme z [1] a [2].

1.1 Výrok

Nejprve zavedeme elementární pojem *výrok*.

Výrok je takové tvrzení, o kterém má smysl uvažovat, zdali je pravdivé, či nikoliv. Výrokům tedy přiřazujeme *pravdivostní hodnoty* pravda (true) nebo nepravda (false).

Příklad 1. Věta „Dnes je hezký den.“ je výrok. Je na čtenáři, aby rozhodl o pravdivosti tohoto výroku. Jiný čtenář by mohl případně rozhodnout opačně.

V této práci se však pravdivostním ohodnocením výroků ani jejich významem nebudeme zabývat.

Výroková logika od vnitřní struktury výroků abstrahuje v pojmu *elementárního výroku*, ze kterých buduje *výrokové formule* pomocí *logických operací*. Jednotlivé elementární výroky značíme velkými písmeny latinky: $A, B, C, \ldots$

1.2 Logické operace

Logické operace dělíme na unární a binární podle jejich arity[1]. Symbolicky je reprezentují příslušné logické spojky následovně:

- Unární

[1] Arita – počet operandů operace potřebných k jejímu provedení.

Negace $\neg$

- Binární

  Konjunkce $\wedge$

  Disjunkce $\vee$

  Implikace $\Rightarrow$

  Ekvivalence $\Leftrightarrow$

Každá operace specificky určuje pravdivostní hodnotu složeného výroku v závislosti na ohodnocení výroků, které pojí. Analogii s logickými operátory lze vidět v operátorech aritmetických. Význam uvedených operací nebudeme zkoumat, protože není pro účely této práce podstatný. Zkoumáme pouze výrokové formule a jejich posloupnosti jako čistě syntaktické objekty.

1.3 Formule

Ústředním pojmem pro tuto práci je *formule*.

Definice 1. Ve výrokové logice definujeme formuli takto:

1. Každá elementární formule je formulí.
2. Vznikne-li α unární logickou operací z formule β nebo binární logickou operací z formulí β a γ , je α také formulí.
3. Každá formule vznikne konečnou aplikací předchozích pravidel.

Jednotlivé formule značíme malými písmeny řecké abecedy: $\alpha, \beta, \gamma, \ldots$

Příklad 2. Pro ilustraci uvažujme dvě následující formule α, β:

- $\alpha = A$
- $\beta = \neg(A \vee B)$

Formule α je elementární formulí, kdežto formule β je složená z několika formulí. Pro názornost popíšeme výstavbu formule β podle předchozí definice.

1. A a B jsou elementární formule (výroky).
2. Formule A, B pojí operace disjunkce do podoby složeného výroku $A \vee B$.
3. Na dosavadní formuli aplikujeme unární operaci negace, čímž vznikne formule $\beta = \neg(A \vee B)$.

1.3.1 Notace

Nyní popíšeme několik způsobů zápisu výrokových formulí.

Výrokové formule lze zapisovat ve třech, sémanticky ekvivalentních, notacích: *prefixní*, *infixní* a *postfixní*. Tyto notace se liší pouze pořadím výpisu logické spojky ve složených výrocích.

Příklad 3. Následující jsou různé zápisy téže formule:

prefixní $\neg \vee \alpha\, \beta$

infixní $\neg(\alpha \vee \beta)$

postfixní $\alpha\, \beta \vee \neg$

Všimněme si, že v případě prefixní a postfixní notace je přednost operací určena jednoznačně na rozdíl od notace infixní, která vyžaduje užití závorek. Je třeba brát na vědomí, že s rostoucí složitostí formule je pro nás čím dál obtížnější udržet pozornost nad její strukturou. Protože infixní zápis je našemu vnímání nejpřirozenější, budeme jej v této práci používat i nadále. Víme však, že kdykoliv lze jednu notaci převést na druhou.

1.4 Důkazový systém

Důkazový systém výrokové logiky rozhoduje o *dokazatelnosti* výrokových formulí. Každý takový systém je tvořen dvěma součástmi:

- Množinou axiomů
- Množinou odvozovacích pravidel

Axiomy jsou schémata výrokových formulí jistého tvaru. Každou formuli ve tvaru popsaném axiomem nazýváme *instancí* tohoto axiomu. Instancí axiomů je tedy nekonečně mnoho, stejně jako výrokových formulí.

Odvozovací pravidla popisují způsoby, jakými lze z daných výrokových formulí odvozovat formule další.

1.4.1 Hilbertův systém

Hilbertův systém je důkazový systém klasické výrokové logiky. Za účelem úspornosti jazyka se omezuje pouze na dvě logické spojky: negaci a implikaci. Tyto dvě spojky tvoří tzv. *minimální univerzální množinu*, proto všechny ostatní spojky můžeme vyjádřit těmito a touto redukcí neomezujeme vyjadřovací možnosti jazyka. Způsob, jakým lze vyjadřovat jedny spojky pomocí druhých, zkoumat nebudeme.

Axiomem Hilbertova systému je každá formule některého z následujících syntaktických tvarů:

A1 $(\varphi \Rightarrow (\psi \Rightarrow \varphi))$

A2 $((\varphi \Rightarrow (\psi \Rightarrow \chi)) \Rightarrow ((\varphi \Rightarrow \psi) \Rightarrow (\varphi \Rightarrow \chi)))$

A3 $((\neg\varphi \Rightarrow \neg\psi) \Rightarrow ((\neg\varphi \Rightarrow \psi) \Rightarrow \varphi))$

Množina odvozovacích pravidel obsahuje jediný prvek, pravidlo *modus ponens*, které je zavedeno následovně:

Definice 2. Z formulí φ a $\varphi \Rightarrow \psi$ odvoď formuli ψ.

V této práci se nadále budeme zabývat výhradně tímto axiomatickým systémem. Existují však i jiné důkazové systémy, které jsou navzájem ekvivalentní (viz část 6.2).

1.4.2 Důkaz

Nyní zavedeme klíčový pojem *důkaz*.

Definice 3. Buď φ výroková formule. Řekneme, že konečná posloupnost výrokových formulí $\varphi_1, \ldots, \varphi_n$ je *důkazem* formule φ, pokud φ_n je formule φ, a každá formule φ_i z této posloupnosti je buďto instancí některého axiomu, nebo je z některých předchozích $\varphi_j, \ldots, \varphi_k$, kde $j, \ldots, k < i$, odvozena odvozovacím pravidlem. Pokud existuje důkaz formule φ, řekneme, že je tato formule je *dokazatelná* ve výrokové logice, a píšeme $\vdash \varphi$.

Každý důkaz tedy nutně začíná axiomem a triviálně každá instance axiomu je dokazatelnou formulí.

Pro nás klíčová je skutečnost, že důkaz vychází z konečně mnoha daných předpokladů, postupuje dle konečně mnoha daných pravidel a je v každém kroku ověřitelný, což lze provést i mechanicky. Na druhou stranu to nedává žádný návod, jak případně důkaz dané formule nalézt. Ostatně tato problematika již přesahuje rozsah této práce.

Příklad 4. Pro ilustraci předvedeme důkaz formule $\varphi \Rightarrow \varphi$ v Hilbertově systému.

axiom 1 $(\varphi \Rightarrow ((\varphi \Rightarrow \varphi) \Rightarrow \varphi))$

axiom 2 $((\varphi \Rightarrow ((\varphi \Rightarrow \varphi) \Rightarrow \varphi)) \Rightarrow ((\varphi \Rightarrow (\varphi \Rightarrow \varphi)) \Rightarrow (\varphi \Rightarrow \varphi)))$

modus ponens, formule 1 a 2 $((\varphi \Rightarrow (\varphi \Rightarrow \varphi)) \Rightarrow (\varphi \Rightarrow \varphi))$

axiom 1 $(\varphi \Rightarrow (\varphi \Rightarrow \varphi))$

modus ponens, formule 4 a 3 $(\varphi \Rightarrow \varphi)$

1.4.2.1 Důkaz z předpokladů

Důkaz lze zobecnit na tzv. *důkaz z předpokladů* rozšířením stávající definice důkazu. Navíc zavedeme množinu *předpokladů* T, tj. formulí, ze kterých v rámci důkazu vycházíme. Zobecnění spočívá v tom, že kromě axiomů jako členy důkazu obdobně připouštíme i formule z množiny předpokladů. Předpoklad je tedy formule, kterou, ačkoliv není axiomem, elementárně považujeme za dokazatelnou. Existuje-li důkaz formule φ z předpokladů T, píšeme $T \vdash \varphi$.

1.4.2.2 Minimální důkaz

Přirozený požadavek na jednoduchost vede k otázce, zdali daný důkaz není v nějakém smyslu zbytečně složitý. Například stočlenná posloupnost libovolných instancí axiomů je formálně korektním důkazem (svého posledního členu), který lze ovšem zkrátit na setinu původní délky. Podobně, pokud v nějakém důkazu na náhodně zvoleném místě rozvineme nějaký jiný důkaz, vnikne posloupnost, která je opět formálně korektním důkazem, ovšem zbytečně složitým.

Ptáme se tedy, zdali předložený důkaz neobsahuje podobné „zbytečnosti“, což nás vede k pojmu *minimální důkaz*.

Definice 4. Buď $\varphi_1, \ldots, \varphi_n$ posloupnost formulí tvořící důkaz formule φ_n. Tento důkaz nazveme *minimálním*, pokud vynecháním libovolné formule posloupnosti $\varphi_1, \ldots, \varphi_n$ přestane důkaz být korektním.

Takový důkaz tedy neobsahuje zbytečné poddůkazy ani duplicitní formule, protože, pokud nějaká formule φ_i vyplývá z nějakých předchozích formulí pomocí odvozovacího pravidla, pak stejným způsobem vyplývá i z každých jejich předchozích, speciálně z prvních, výskytů. Z minimálního důkazu už nelze nic odstranit. To však neznamená, že neexistuje nějaký úplně jiný důkaz téže formule, který by byl ještě úspornější.

Kapitola 2

Vymezení požadavků

Na základě oficiálního zadání této práce, kterým začíná tento text, vymezíme v této kapitole požadavky na implementovaný software. Tyto požadavky kategorizujeme na *funkční* a *nefunkční*.

Funkční požadavky definují cíle, kterých má projekt dosáhnout. V našem případě se jedná o funkcionalitu námi implementovaného software. Na základě funkčních požadavků lze navrhnout metody testování a ověřit úspěšné splnění zadání na konci projektu.

Mezi nefunkční požadavky řadíme náležitosti, které popisují způsob, jakým máme provést implementaci. Některé z nich jsou součástí zadání, některé si pro úplnost zadání stanovíme sami. Nefunkční požadavky jsou omezení, ze kterých vycházíme, a nejsou předmětem testování.

2.1 Funkční požadavky

Úkolem je vypracovat aplikaci *pl* (propositional logic), která primárně dokáže rozhodnout, zdali je daná posloupnost výrokových formulí korektním formálním důkazem v Hilbertově systému. Požadavkem nad rámec zadání je minimalizace těchto důkazů. Elementární funkcionalitou programu je syntaktická analýza vstupních formulí a jejich případný výpis ve zvolené alternativní notaci a jazyce. Nekorektní vstup program správně detekuje a případně jeho nekorektnost podrobně hlásí.

Aplikaci je třeba náležitě otestovat a metody užité při testování zdokumentovat. Dále je třeba vytvořit dokumentaci zdrojového kódu a uživatelskou příručku k aplikaci.

2.2 Nefunkční požadavky

2.2.1 Aplikace

Aplikace bude podporována na platformách systému UNIX a implementována v jazyce C++. Její obsluha bude možná přes systémový terminál, tedy nebude disponovat grafickým uživatelským rozhraním. To nepředstavuje žádné podstatné omezení funkcionality, neboť pracujeme pouze s textovým vstupem. Veškerá funkcionalita aplikace bude dostupná standardně pomocí přepínačů příkazové řádky. Vstupní data budou aplikaci předávána buďto na standardním vstupu, anebo v textovém souboru. Návratová hodnota programu bude signalizovat úspěšnost výsledku.

2.2.2 Vstup

Aplikace čte buďto ze standardního vstupu nebo ze souboru textový ASCII vstup v podobě posloupnosti řádek obsahujících výrokové formule zapsané ve stanoveném jazyce. Podporován je prefixní, infixní i postfixní zápis formulí. Vstup nevyhovující stanovené formě je považován za nekorektní a je odmítnut.

2.2.3 Výstup

Forma výstupu závisí na zvolené funkcionalitě, ke které je program nakonfigurován při spuštění. Podporován je výpis formulí v prefixní, infixní i postfixní notaci. Výstupní jazyk formulí může mít podobu ASCII znaků, přirozeného jazyka nebo jazyka LaTeX. Výstupem programu může být v určitých případech chybové hlášení o nekorektním vstupu.

2.2.4 Dokumentace

Programátorská dokumentace bude dostupná v podobě HTML stránek za použití nástroje *Doxygen*, který takovou dokumentaci generuje. Zdrojový kód programu proto opatříme komentáři speciálního stylu, který tento nástroj vyžaduje.

2.2.5 Příručka

Uživatelskou příručku realizujeme v podobě standardní manuálové stránky operačních systémů UNIX. Tato stránka bude po instalaci přístupná standardně pomocí příkazu `man pl`.

Kapitola 3

Analýza a návrh

3.1 Technický kontext

V této části popíšeme technologie, které jsme pro implementaci stanovili nefunkčními požadavky, a odůvodníme jejich užití.

3.1.1 C++

Programovací jazyk C++ je rozšířením jazyka C. Vybrali jsme jej zejména proto, že podporuje objektově-orientované paradigma, kterého se budeme držet. Přednosti tohoto programovacího stylu nám usnadní nejen implementaci, ale i návrh.

Pro jazyk C++ existuje velké množství standardních knihoven, které nabízejí běžné funkce či běžné datové typy. Jejich užití vede k úspoře zdrojového kódu a menší míře zanesených chyb. Při implementaci budeme maximálně využívat těchto standardních knihoven. Zejména využijeme knihovny STL obsahující šablony základních kontejnerů. Naopak, nebudeme využívat prostředky třetích stran.

Při implementaci také dbáme na přenositelnost a rozšiřitelnost aplikace, jak požaduje zadání.

3.1.1.1 C++11

C++11 je jedním z posledních standardů jazyka C++. Vybrali jsme jej především z důvodu úspory kódu, což umožňují některé konstrukty, které tento standard zavedl, zejména např. alternativní zápis cyklu `for`, který abstrahuje od iterátorů při procházení kontejnerů[3].

3.1.2 Make

Nástroj `make` slouží k zjednodušení sestavování programů. Jeho předností je zejména schopnost na základě časové známky určit, které součásti programu

je třeba zkompilovat v případě změny zdrojového kódu. Sestavení tento nástroj řídí konfiguračním souborem `Makefile`. Ten má předepsanou strukturu a obsahuje definice pravidel, tzv. *cílů*, které mají svůj název a konkrétní účel. Cíl je definován posloupností příkazů pro `shell`, které jsou provedeny při jeho volání. Každý cíl může navíc obsahovat *závislosti*. Závislost je další cíl, který je volán přednostně. Speciálními (konečnými) cíli bývá kompilace konkrétního zdrojového souboru.

My použijeme nástroj GNU `make`, který je jednou z implementací klasického `make` rozšířenou o pokročilé funkce. Z nich konkrétně využijeme např. *pattern rules*, *wildcard characters* či *string functions*[4].

3.1.3 Doxygen

Doxygen je nástroj pro automatickou tvorbu dokumentace zdrojového kódu. Umožňuje dokumentovat kód mnohých populárních programovacích jazyků, zejména C++. Podporuje různé formy výstupu, přičemž my zvolíme dokumentaci v podobě HTML stránek. Text dokumentace tento nástroj čerpá přímo ze souborů zdrojového kódu prostřednictvím speciálních komentářů. To zajišťuje neustálou konzistenci mezi dokumentací a zdrojovým kódem. Tento nástroj funguje i v případě nezdokumentovaného kódu, kdy alespoň podá základní přehled o prvcích zdrojového kódu. V neposlední řadě je schopen vizualizovat relace mezi elementy v podobě diagramů, jako je např. diagram tříd[5].

My budeme dokumentaci zapisovat především do hlavičkových souborů `.hpp`, aby nepřekážela v implementaci.

3.1.4 Mdoc

Manuálové stránky systému UNIX lze psát ve speciálním značkovacím jazyce. Takový jazyk je ve skutečnosti balíček maker pro jazyk *troff*. Troff pochází ze 70. let a podobně jako např. TEX slouží k sazbě textu. Jazyk zpracovává stejnojmenný textový procesor, který podporuje právě zmíněné balíčky maker pro tento jazyk. Jednotlivá makra, podobně jako např. značky jazyka HTML, formátují daný obsah do požadované podoby[6]. Běžné manuálové stránky užívají balíček maker `man`. My však zvolíme sofistikovanější balíček `mdoc`, který narozdíl od `man` disponuje sémantickou koncepcí maker. Záznam o tomto balíčku nalezneme v sekci *mdoc(7)* manuálových stránek[7].

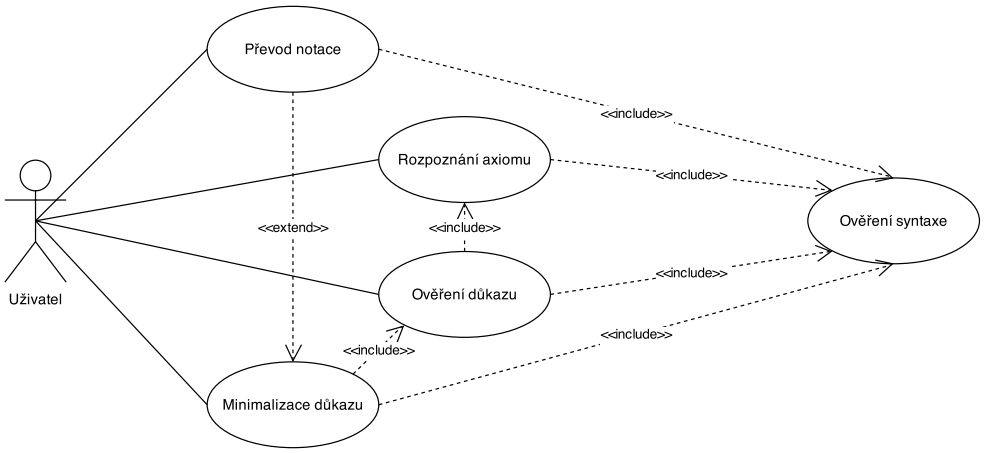
Aby manuálová stránka byla dostupná pomocí příkazu `man`, musí splňovat určité náležitosti. Název souboru manuálové stránky má následující formát:

```
[název aplikace].[manuálová sekce]
```

Stejnojmenný archiv souboru s manuálovou stránkou je umístěn v adresáři, ve kterém příkaz `man` implicitně hledá manuálové stránky[2] pod adresářem

[2]Tato umístění zobrazíme, popř. nastavíme pomocí příkazu `manpath`.

Obrázek 3.1: Diagram případů užití



příslušícím dané manuálové sekci. V našem případě se jedná o adresář `man1`. Absolutní cesta k naší manuálové stránce může vypadat například následovně: `/usr/local/man/man1/pl.1.gz`. Soubor archivu s manuálovou stránkou má vlastníka uživatele `root` a přístupová práva 0644.

Manuálové stránky mají standardní strukturu, kterou dodržíme. Text příručky napíšeme v angličtině a její obsah rozdělíme do příslušných standardních sekcí.

3.2 Případy užití

Na základě funkčních požadavků nastíníme možné případy užití aplikace (viz obrázek 3.1).

3.2.1 Převod notace

Elementární funkcionalitou aplikace je převod formulí dané notace do notace vybrané. Zároveň je tato funkcionalita vhodná i pro kontrolu syntaktického zápisu formulí. Forma logických spojek výstupu je dostupná v podobě znaků ASCII (stejně jako vstup), v přirozeném jazyce nebo v jazyce LaTeX.

3.2.2 Rozpoznání axiomu

Protože součástí ověření důkazu je rozpoznání axiomů, nabídneme tuto funkci také samostatně. Každá formule vstupní posloupnosti je potom ověřována jako axiom a na výstupu je případně uváděn jeho typ.

Tabulka 3.1: Výstupní formy logických spojek

Operátor	ASCII	Slovní	LaTeX
Negace	`-`	`not`	`\neg`
Konjunkce	`.`	`and`	`\wedge`
Disjunkce	`+`	`or`	`\vee`
Implikace	`>`	`implies`	`\Rightarrow`
Ekvivalence	`=`	`iff`	`\Leftrightarrow`

3.2.3 Ověření důkazu

Tato funkcionalita je hlavním cílem této práce. Na vstupní posloupnost formulí je nahlíženo jako na posloupnost členů důkazu, přičemž aplikace ověří, zdali je důkaz korektním. Je možné ověřovat i důkaz z předpokladů. Tehdy je prvních n vstupních formulí považováno za prvky teorie, pak následují formule vlastního důkazu. Výstupem aplikace jsou podrobnosti o dokazatelnosti, popř. nedokazatelnosti každého členu důkazu.

3.2.4 Minimalizace důkazu

Důkaz je nejprve ověřen stejně jako v předchozím případě. Následně, pokud je to možné, je minimalizován do nejúspornější možné podoby (viz definici 4). Výstupem je důkaz téže formule právě v této podobě.

3.3 Návrh

3.3.1 Forma vstupu

Textový vstup aplikace `pl` se skládá z posloupnosti formulí, přičemž každá z nich je zakončena řádkovým zalomením `\n`. Zápis formulí užívá znaků `A-Z` pro elementární výroky. Tím je vstup omezen na 26 různých elementárních výroků, což ovšem pro naše účely plně postačuje. Symboly `-`, `.`, `+`, `>`, `=` značí po řadě logické operátory negace, konjunkce, disjunkce, implikace a ekvivalence. Pro infixní notaci formulí navíc rozpoznáváme závorky `()` pro určování přednosti operací.

3.3.2 Forma výstupu

Kromě výstupní notace lze zvolit i jednu ze tří výstupních forem (jazyků) logických spojek. Jak jsou jednotlivé operátory zastoupeny v těchto jazycích, znázorňuje tabulka 3.1.

3.3.3 Uživatelské rozhraní

Nefunkčním požadavkem na uživatelské rozhraní je obsluha pomocí přepínačů. V této části tedy navrhneme přepínače aplikace `pl` pokrývající veškerou funkcionalitu stanovenou v požadavcích.

-A (axiom checker) Ověří vstupní formule jako axiomy.

-e (echo) Povolí hlášení na standardním a standardním chybovém výstupu. Implicitně není produkován žádný výstup.

-f file (input file) Použije soubor *file* jako vstup namísto (implicitního) standardního vstupu.

-i syntax (input syntax) Nastaví danou vstupní notaci formulí. Možné hodnoty jsou: *prefix*, *infix* a *postfix*. Implicitní hodnota je *infix*.

-l language (output language of connectives) Nastaví danou výstupní formu jazyka logických spojek. Možné hodnoty jsou: *ascii*, *words* a *latex*. Implicitní hodnota je *ascii*.

-o syntax (output syntax) Nastaví danou výstupní notaci formulí. Možné hodnoty jsou: *prefix*, *infix* a *postfix*. Implicitní hodnota je *infix*.

-M n (proof minimizer) Minimalizuje důkaz ze vstupní posloupnosti formulí. Parametr *n* vyjadřuje počet formulí množiny předpokladů, které na vstupu předcházejí samotnému důkazu.

-P n (proof checker) Ověří důkaz ze vstupní posloupnosti formulí. Parametr *n* vyjadřuje počet formulí množiny předpokladů, které na vstupu předcházejí samotnému důkazu.

-s (strict) Způsobí zastavení vykonávání programu při prvním neúspěchu, např. nekorektním syntaktickém zápisu formule. Přepínače `-M` a `-P` takové chování povolují automaticky, protože nemá smysl dále minimalizovat či ověřovat evidentně nekorektní důkaz.

Axiom checker, *proof checker* a *proof minimizer* budeme nazývat *cíli* aplikace. Volba cíle je nepovinná, je však povoleno spustit aplikaci s jediným zvoleným cílem.

Kapitola 4

Implementace

V této kapitole popíšeme, jak ty které části zdrojového kódu implementují jednotlivé součásti systému. Zmíníme však pouze ty nejdůležitější z nich.

4.1 Hlavní program

Hlavní program aplikace představuje funkce `main`.

main Inicializací instance třídy `Configuration` dojde ke konfiguraci aplikace dle parametrů příkazové řádky. V případě úspěchu je proveden cíl aplikace. V případě, že se nepodaří aplikaci nakonfigurovat, je vypsáno chybové hlášení a běh aplikace končí neúspěchem.

4.2 Popis tříd

V této části vysvětlíme význam jednotlivých funkcí, tříd a jejich metod. Následující text koresponduje s dokumentací zdrojového kódu.

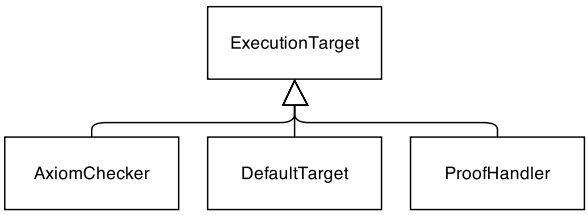
4.2.1 Třída Configuration

Tato třída představuje konfiguraci aplikace. Obsahuje proměnné, které řídí její běh.

Konstruktor Pomocí funkce `getopt` z knihovny `unistd.h` jsou zpracovány přepínače z příkazové řádky, podle kterých jsou nastaveny třídní proměnné.

parseFormula Tato metoda zavolá příslušný parser formulí, kterému předá příslušný vstupní proud.

Obrázek 4.1: Diagram tříd rodiny ExecutionTarget



printFormula Tato metoda vrátí textovou podobu dané formule v příslušné notaci a jazyce logických spojek.

4.2.2 Třída ExecutionTarget

Tato abstraktní třída představuje cíl aplikace. Hierarchii tříd rodiny `ExecutionTarget` znázorňuje diagram 4.1.

executeTarget Účelem této abstraktní metody je provést cíl v závislosti na konfiguraci aplikace a vrátit příslušnou návratovou hodnotu. Základem všech implementací této metody je cyklické zpracování formulí vstupní posloupnosti, které vždy začíná přijetím formule metodou `parseFormula`. V případě nekorektního vstupu je mimo cyklus zachycena výjimka typu `ParseException` a vypsáno chybové hlášení získané metodou `getMessage`. Některé cíle v rámci cyklu zohledňují konfigurační příznak `strict` a v případě neúspěšné iterace cyklus předčasně zastavují.

4.2.2.1 AxiomChecker

Tato třída představuje cíl axiom checker.

executeTarget Přijatá vstupní formule je v rámci daného důkazového systému metodou `isAxiom` ověřena jako axiom. Následuje uvolnění formule z paměti a výpis hlášení o shodě či neshodě s konkrétním typem axiomu.

4.2.2.2 DefaultTarget

Tato třída představuje stav spuštění bez explicitního cíle.

executeTarget Přijatá vstupní formule je vypsána na výstupu pomocí metody `printFormula` a následně je uvolněna z paměti.

4.2.2.3 ProofHandler

Tato třída představuje cíle proof checker a proof minimizer.

executeTarget Určitý počet přijatých vstupních formulí je nejprve zařazen mezi prvky teorie. Tyto formule totiž ještě nejsou členy vlastního důkazu. Zbylé formule jsou zpracovány následovně.

Přijatá formule je postupně ověřena jako axiom metodou `isAxiom`, jako prvek teorie a jako formule odvoditelná v rámci daného důkazového systému metodou `isDeducible`. Každá formule splňující jedno z těchto kritérií je přidána do seznamu prvků třídy `ProofMember` představujícího důkaz. Nesplňuje-li formule ani jedno z uvedených kritérií, důkaz není korektním a aplikace končí neúspěchem. Je-li instance `ProofHandler` nastavena jako cíl proof minimizer, následuje iterativní algoritmus minimalizace důkazu.

Inicializujeme zásobník prvků třídy `ProofMember` a uložíme na něj poslední člen důkazu. Dokud není zásobník prázdný, prohlašujeme jeho vrchol členem minimálního důkazu metodou `setPreserve` a jeho přímé svědky ukládáme na zásobník. Zároveň sledujeme počet takto zpracovaných prvků. Je-li na konci tento počet shodný s počtem členů důkazu, důkaz již byl minimálním, tedy nelze jej minimalizovat.

Data: Z je zásobník členů důkazu
ulož poslední člen důkazu na Z;
while Z *není prázdný* **do**
  vrchol Z je členem minimálního důkazu;
  foreach *svědek* s *vrcholu* Z **do**
    ulož s na Z;
  end
end

Členy důkazu, které jsme označili jako členy minimálního důkazu, vypíšeme na výstup.

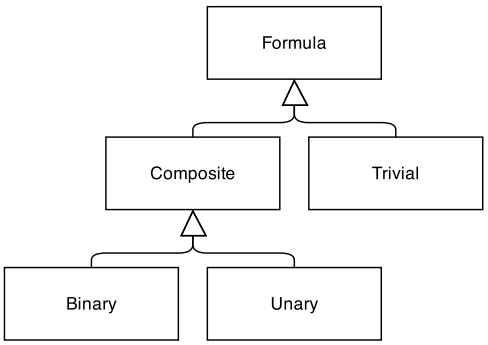
4.2.3 Třída Formula

Struktura výrokové formule má stromový charakter, a proto jsme ji implementovali jako *syntaktický strom*. Tato abstraktní třída představuje uzel tohoto stromu, tedy dílčí formuli celého výrazu. Objekt této třídy obsahuje znak reprezentující uzel jako logickou spojku nebo výrok. Hierarchii tříd rodiny `Formula` znázorňuje diagram 4.2.

equals Účelem této abstraktní metody je zjistit, zdali se daná formule shoduje s touto.

matches Účelem této abstraktní metody je zjistit, zdali daná formule vyhovuje tvaru určenému touto formulí.

Obrázek 4.2: Diagram tříd rodiny Formula



4.2.3.1 Třída Composite

Tato abstraktní třída představuje formuli složenou z logické spojky a příslušných operandů. Znak reprezentující tento uzel stromu je jednou z logických spojek.

4.2.3.2 Třída Trivial

Tato třída představuje triviální formuli, tedy elementární výrok. Znak reprezentující tento uzel stromu je jedním ze symbolů pro výroky.

equals Implementace spočívá v porovnání reprezentujícího znaku této a dané formule.

matches Implementace spočívá v porovnání metodou `equals` dané formule s formulí, která je substitucí za reprezentující znak této formule. Tyto substituce se vyhledávají v mapě substitucí, jejíž reference je předána parametrem metody `matches`. Pokud se v této mapě doposud nenachází substituce za reprezentující symbol této formule, do mapy je tento symbol vložen jako klíč a daná formule jako jeho hodnota a metoda vrátí `true`.

Data: s je reprezentující symbol této formule, φ je daná formule, T je tabulka substitucí
if *v* T *se nachází klíč* s **then**
  return *formule* $T(s)$ *se shoduje s formulí* φ;
end
else
  do T dosaď dvojici $\{s, \varphi\}$;
  return *true*;
end

4.2.3.3 Třída Binary

Tato třída představuje binární operátor. Objekt této třídy obsahuje odkaz na svůj levý a pravý operand, tedy levého a pravého potomka tohoto uzlu v syntaktickém stromu formule.

equals Implementace spočívá v porovnání reprezentujícího symbolu této a dané formule a následném porovnání shody levého a pravého operandu této formule metodou `equals` s levým a pravým operandem formule dané.

Data: φ je daná formule
if *reprezentující symboly této formule a* φ *se shodují* **then**
  přetypuj φ na objekt typu `Binary`;
  return *levý operand této formule se shoduje s levým operandem formule* φ *a zároveň pravý operand této formule se shoduje s pravým operandem formule* φ;
end

matches Implementace spočívá v porovnání reprezentujícího symbolu této a dané formule a následném porovnání tvaru levého a pravého operandu této formule metodou `matches` s levým a pravým operandem formule dané.

Data: φ je daná formule
if *reprezentující symboly této formule a* φ *se shodují* **then**
  přetypuj φ na objekt typu `Binary`;
  return *levý operand této formule vyhovuje tvaru levého operandu* φ *a zároveň pravý operand této formule vyhovuje tvaru pravého operandu* φ;
end

4.2.3.4 Třída Unary

Tato třída představuje formuli složenou unárním operátorem. Objekt této třídy obsahuje odkaz na svůj operand, tedy potomka tohoto uzlu v syntaktickém stromu formule.

equals Implementace spočívá v porovnání reprezentujícího symbolu této a dané formule a následném porovnání shody operandu této formule metodou `equals` s operandem formule dané.

matches Implementace spočívá v porovnání reprezentujícího symbolu této a dané formule a následném porovnání tvaru operandu této formule metodou `matches` s operandem formule dané.

4.2.4 Parser formulí

Parserem abstraktně nazveme tu část programu, která převádí vstup z textové do vnitřní formy. V našem případě jej reprezentují samostatné funkce *parsePrefix*, *parseInfix* a *parsePostfix*. Každá z těchto funkcí slouží ke zpracování vstupních formulí v příslušné notaci. V případě úspěchu funkce vrací kořen syntaktického stromu přijaté formule.

Syntaktická analýza postupuje v cyklu po jednotlivých znacích vstupního proudu, aby mohla být přerušena již v místě případné syntaktické chyby, tedy ještě před přijetím celého vstupu. Nyní popíšeme algoritmy syntaktické analýzy korektního vstupu těchto funkcí.

parsePrefix Prefixní parser ukládá přijaté logické operátory na zvláštní zásobník s operátory. Po přijetí elementární formule je tato nastavena jako operand zleva vrcholu zásobníku. Ve chvíli, kdy jsou operandy vrcholu zásobníku již plně obsazeny, dojde k sejmutí operátoru ze zásobníku. Tento operátor je pak nastaven jako operand zleva novému vrcholu zásobníku. To se opakuje, dokud aktuální vrchol zásobníku opět nedisponuje alespoň jedním volným operandem.

parseInfix Infixní parser ukládá přijaté logické operátory na zvláštní zásobník s operátory a elementární formule na zvláštní zásobník s formulemi. K orientaci ve struktuře formule využívá další zásobník se stavy zpracování větví syntaktického stromu. Každá taková úroveň zpracování může mít následující stav:

UNARY Byl nastaven unární operátor.

BLANK Byla otevřena nová větev stromu.

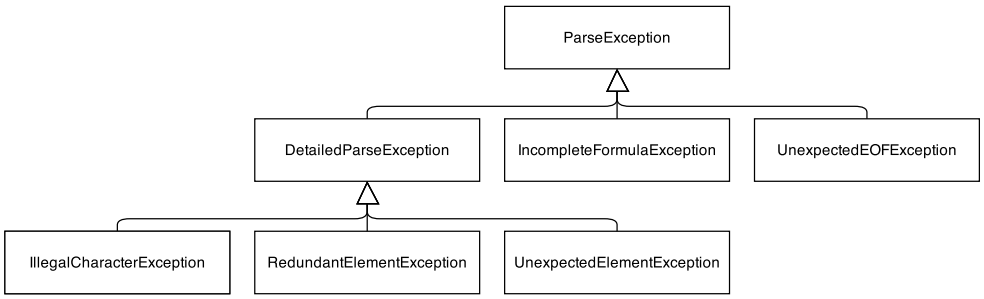
FIRST_OPERAND Byl nastaven první operand.

BINARY Byl nastaven binární operátor.

LAST_OPERAND Byl nastaven poslední operand.

Podle těchto stavů lze na základě jistých pravidel vždy rozhodně určit, zdali aktuálně zpracovávaný prvek neporušuje infixní syntaxi formulí. Taková pravidla popisují chování infixního parseru v závislosti na přijatém prvku a stavu zpracování aktuální větve stromu.

Obrázek 4.3: Diagram tříd rodiny ParseException



parsePostfix Postfixní parser ukládá přijaté elementární formule na zvláštní zásobník s formulemi. Ve chvíli, kdy je přijat logický operátor, dojde k sejmutí příslušného počtu formulí ze zásobníku, přičemž tyto jsou zprava nastaveny jako operandy přijatého operátoru. Ten je následně uložen na zásobník.

4.2.5 Třída ParseException

Tato obecná třída představuje výjimku při syntaktické analýze v případě nekorektního vstupu. Objekt této třídy obsahuje řetězec s popisem syntaktické chyby. Hierarchii výjimek rodiny `ParseException` znázorňuje diagram 4.3.

getMessage Tato metoda vrátí podrobnosti o chybě v podobě řetězce s chybovým hlášením.

4.2.5.1 Výjimka DetailedParseException

Tento obecný typ výjimky představuje syntaktickou chybu specifikovanou znakem vstupu, který chybu způsobil, a jeho pozici na řádce (nepočítaje bílé znaky).

4.2.5.2 Výjimka IncompleteFormulaException

Tento typ výjimky představuje případ nekompletní formule.

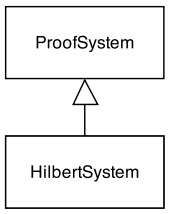
4.2.5.3 Výjimka UnexpectedEOFException

Tento typ výjimky představuje případ výskytu konce vstupu v místě, které není stanoveno korektní formou vstupu.

4.2.5.4 Výjimka IllegalCharacterException

Tento typ výjimky představuje případ užití nepovoleného znaku.

Obrázek 4.4: Diagram tříd rodiny ProofSystem



4.2.5.5 Výjimka RedundantElementException

Tento typ výjimky představuje případ výskytu nadbytečných prvků nacházejících se za korektní formulí.

4.2.5.6 Výjimka UnexpectedElementException

Tento typ výjimky představuje případ nekorektního zápisu formule (týká se pouze infixní notace).

4.2.6 Třída ProofMember

Tato třída představuje člen důkazu jako komplexní strukturu. Objekt této třídy obsahuje výrokovou formuli jakožto člen důkazu, seznam přímých svědků tohoto členu a logickou proměnnou `preserve` vyhrazenou pro algoritmus minimalizace důkazu.

4.2.7 Třída ProofSystem

Tato abstraktní třída představuje důkazový systém výrokové logiky. Objekt této třídy obsahuje axiomy důkazového systému. Hierarchii tříd rodiny `ProofSystem` znázorňuje diagram 4.4.

isAxiom Účelem této metody je ověřit, zdali je daná formule instancí některého z axiomů důkazového systému. Implementace spočívá v porovnání dané formule s jednotlivými typy axiomů pomocí metody `matches`.

Data: ψ je daná formule
foreach *axiom* α *důkazového systému* **do**
  if ψ *vyhovuje tvaru* α **then**
    return *typ axiomu* α;
  end
end
return *false*;

isDeducible Účelem této abstraktní metody je ověřit, zdali je daná formule dokazatelná z předchozích kroků daného důkazu. Metoda je abstraktní, protože každý konkrétní důkazový systém odvozuje formule jiným způsobem, resp. odvozovacím pravidlem. Návratovou hodnotou je posloupnost indexů členů důkazu, ze kterých je daná formule odvoditelná. Tato posloupnost je prázdná v případě, že formuli v rámci daného důkazu odvodit nelze.

4.2.7.1 Třída HilbertSystem

Tato třída představuje Hilbertův systém. Pro účely odvozování formulí obsahuje objekt této třídy formuli představující implikaci pravidla modus ponens ($\varphi \Rightarrow \psi$).

isDeducible Implementace spočívá v procházení jednotlivých formulí daného důkazu, přičemž na jednu z formulí je nahlíženo jako na předpoklad pravidla modus ponens (φ) a na druhou jako na jeho implikaci ($\varphi \Rightarrow \psi$). Daná formule (ψ) je odvoditelná, je-li nalezena dvojice formulí vyhovující právě těmto tvarům.

Data: ψ je daná formule
foreach *formule* α *důkazu* **do**
  foreach *formule* β *důkazu* **do**
    if β *vyhovuje tvaru* $\alpha \Rightarrow \psi$ **then**
      return *indexy formulí* α, β;
    end
  end
end

4.2.8 Třída UsageException

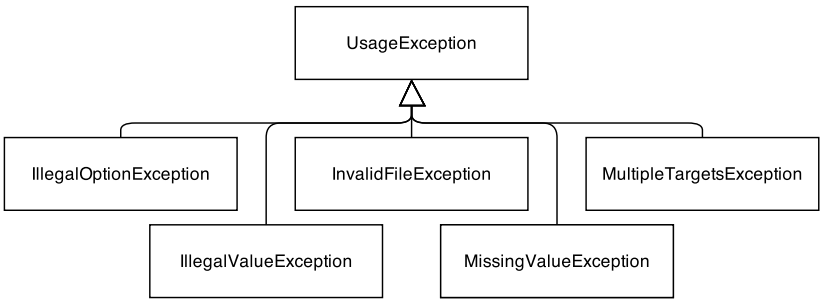
Tato obecná třída představuje výjimku při konfiguraci aplikace v případě nesprávné syntaxe příkazové řádky. Objekt této třídy obsahuje záznam o tom, který přepínač syntaktickou chybu způsobil, a řetězec s popisem této chyby. Hierarchii výjimek rodiny `UsageException` znázorňuje diagram 4.5.

getMessage Tato metoda vrátí podrobnosti o syntaktické chybě v podobě řetězce s chybovým hlášením.

4.2.8.1 Výjimka IllegalOptionException

Tento typ výjimky představuje případ použití nepovoleného přepínače.

Obrázek 4.5: Diagram tříd rodiny UsageException



4.2.8.2 Výjimka IllegalValueException

Tento typ výjimky představuje případ použití nepovolené hodnoty přepínače.

4.2.8.3 Výjimka InvalidFileException

Tento typ výjimky představuje případ zadání nepoužitelného vstupního souboru.

4.2.8.4 Výjimka MissingValueException

Tento typ výjimky představuje případ vynechání povinné hodnoty přepínače.

4.2.8.5 Výjimka MultipleTargetsException

Tento typ výjimky představuje případ zadání více cílů k provedení najednou.

Kapitola 5

Testování

V této kapitole popíšeme užité metody testování aplikace, jak požaduje zadání. Testovali jsme oblasti pokryté případy užití:

- Zpracování vstupu.
  - Ověření syntaxe.
  - Převod notace.
- Rozpoznání axiomu.
- Ověření důkazu.
- Minimalizaci důkazu.

Základní myšlenkou pro maximalizaci pokrytí případů testy je v každé této oblasti provést následující typy testů:

Pozitivní test Test korektního přijetí korektního vstupu.

Negativní test Test korektního odmítnutí nekorektního vstupu.

Princip všech testů spočívá ve zpracování připravených vstupních dat a následné konfrontaci výstupu aplikace s připravenými výstupními testovacími daty, která považujeme za korektní. Touto konfrontací myslíme porovnání těchto dat pomocí nástroje `diff`. Vstupní a výstupní testovací data jsme uložili do textových souborů do adresáře `test`.

Vlastní testy jsou technicky realizovány skriptem pro `shell`. Spuštění tohoto testovacího skriptu volá cíl `make test`. Soubory s příponou `_test` vyprodukované testy jsou směrovány do adresáře `out`.

5.1 Zpracování vstupu

Všechny případy užití aplikace zahrnují užití parseru výrokových formulí. Proto jsme nejdříve ze všeho ověřili, zdali parser korektně transformuje korektní textový vstup do stanovené vnitřní reprezentace a nekorektní vstup korektně odmítá. To jsme zjistili porovnáním vstupních dat s výstupními daty téže notace. Protože parser v naší implementaci reprezentují tři nezávislé funkce (`parsePrefix`, `parseInfix` a `parsePostfix`), bylo nezbytné testovat vstupní data všech tří notací.

5.1.1 Pozitivní test

Pozitivní test ověřuje korektní převod formulí každé ze tří notací vstupu do každé ze tří notací výstupu.

5.1.1.1 Vstupní data

Připravili jsme dostatečný počet různých výrokových formulí tak, abychom jimi pokryli následující případy:

- Kořenem stromu je:
  - Elementární formule
  - Složená formule
    - Unární operátor
    - Binární operátor
- Potomek uzlu je:
  - Elementární formule
  - Složená formule
    - Unární operátor
    - Binární operátor

Vstupními testovacími daty (pro infixní notaci) je následující posloupnost formulí:

```
1 A
2 -A
3 (A.B)
4 -(-A.-B)
5 (A.((B+(C>D))=E))
6 -(-A.-(-(-B+-(-C>-D))=-E))
```

5.1.1.2 Výstupní data

Výstupními testovacími daty (pro infixní notaci) jsou tatáž data.

5.1.2 Negativní test

Negativní test ověřuje korektní odmítnutí nekorektního vstupu. Nekorektní vstup však může mít mnoho podob, jež bylo třeba analyzovat. K tomu nám posloužil výčet druhů výjimek rodiny `ParseException` vrhaných parserem.

5.1.2.1 Vstupní data

Připravili jsme dostatečný počet různých nekorektních výrokových formulí tak, abychom jimi pokryli následující případy:

1. Formule je nekompletní.
2. Formule má chybnou syntaxi (pouze infixní notace).
3. Formule obsahuje nadbytečné prvky.
4. Formule obsahuje nepovolený znak.
5. Formule není korektně ukončena.

Vstupními testovacími daty (pro infixní notaci) je následující posloupnost formulí:

```
(A.((B+C>D))=E))
A.((B+(C>D))=E)
(A*((B+(C>D))=E))
(A.((B+(C>D))=E)
(A.((B+(C>D))=E))
```

5.1.2.2 Výstupní data

Výstupními testovacími daty (pro infixní notaci) je následující posloupnost hlášení:

```
Unexpected element '>' at position 9.
Unnecessary element '.' at position 2.
Illegal character '*' at position 3.
Incomplete formula.
Unexpected end of stream.
```

5.2 Rozpoznání axiomu

Pro testování rozpoznání axiomů jsme využili axiomy Hilbertova systému.

5.2.1 Pozitivní test

Pozitivní test ověřuje korektní rozpoznání axiomů všech typů.

5.2.1.1 Vstupní data

Vstupními testovacími daty je následující posloupnost formulí:

```
(A>(B>A))
((A>(B>C))>((A>B)>(A>C)))
((-A>-B)>(B>A))
```

5.2.1.2 Výstupní data

Výstupními testovacími daty je následující posloupnost hlášení:

```
Axiom of type 1.
Axiom of type 2.
Axiom of type 3.
```

5.2.2 Negativní test

Negativní test ověřuje korektní odmítnutí formulí zdánlivě podobných axiomům.

5.2.2.1 Vstupní data

Vstupními testovacími daty je následující posloupnost formulí:

```
(A>(A>B))
((A>(B>C))>((A>C)>(A>B)))
((-A>-B)>(A>B))
```

5.2.2.2 Výstupní data

Výstupními testovacími daty je následující posloupnost chybových hlášení:

```
Not an axiom.
Not an axiom.
Not an axiom.
```

5.3 Ověření důkazu

Pro testování ověření důkazu jsme využili pozměněný důkaz formule $A \Rightarrow A$ z příkladu 4.

5.3.1 Pozitivní test

Pozitivní test ověřuje korektní ověření značně neminimálního avšak korektního důkazu.

5.3.1.1 Vstupní data

Vstupními testovacími daty je následující posloupnost formulí:

```
1  (A>((A>A)>A))
2  (A>((A>A)>A))
3  (A>((A>A)>A))
4  ((A>((A>A)>A))>((A>(A>A))>(A>A)))
5  ((A>(A>A))>(A>A))
6  ((A>((A>A)>A))>((A>(A>A))>(A>A)))
7  (A>(A>A))
8  ((A>((A>A)>A))>((A>(A>A))>(A>A)))
9  ((A>(A>A))>(A>A))
10 ((A>((A>A)>A))>((A>(A>A))>(A>A)))
11 (A>A)
```

5.3.1.2 Výstupní data

Výstupními testovacími daty je následující posloupnost hlášení:

```
1  Axiom of type 1.
2  Axiom of type 1.
3  Axiom of type 1.
4  Axiom of type 2.
5  Deducible using formulas 1 4 as witnesses.
6  Axiom of type 2.
7  Axiom of type 1.
8  Axiom of type 2.
9  Deducible using formulas 1 4 as witnesses.
10 Axiom of type 2.
11 Deducible using formulas 7 5 as witnesses.
```

5.3.2 Negativní test

Negativní test ověřuje korektní odmítnutí nekorektního důkazu.

5.3.2.1 Vstupní data

Vstupními testovacími daty je následující posloupnost formulí:

```
(A>((A>A)>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
(A>(A>A))
(A>A)
```

5.3.2.2 Výstupní data

Výstupními testovacími daty je následující posloupnost hlášení:

```
Axiom of type 1.
Axiom of type 2.
Axiom of type 1.
Formula not deducible.
```

5.4 Minimalizace důkazu

Pro testování ověření důkazu jsme využili pozměněný důkaz formule $A \Rightarrow A$ z příkladu 4.

5.4.1 Pozitivní test

Pozitivní test ověřuje korektní minimalizaci značně neminimálního důkazu.

5.4.1.1 Vstupní data

Vstupními testovacími daty je následující posloupnost formulí:

```
(A>((A>A)>A))
(A>((A>A)>A))
(A>((A>A)>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
((A>(A>A))>(A>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
(A>(A>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
((A>(A>A))>(A>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
(A>A)
```

5.4.1.2 Výstupní data

Výstupními testovacími daty je následující posloupnost formulí:

```
(A>((A>A)>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
((A>(A>A))>(A>A))
(A>(A>A))
(A>A)
```

5.4.2 Negativní test

Negativní test ověřuje korektní odmítnutí již minimálního důkazu.

5.4.2.1 Vstupní data

Vstupními testovacími je následující posloupnost formulí:

```
(A>((A>A)>A))
((A>((A>A)>A))>((A>(A>A))>(A>A)))
((A>(A>A))>(A>A))
(A>(A>A))
(A>A)
```

5.4.2.2 Výstupní data

Výstupními testovacími daty je následující hlášení:

```
Proof already minimal.
```

5.5 Další testy

Kromě testů korektní funkčnosti aplikace jsme ověřili následující náležitosti.

5.5.1 Test uvolnění paměti

Programy psané v jazyce C++ vyžadují explicitní správu paměti již ve zdrojovém kódu. Pro detekci paměťových úniků za běhu aplikace a ověření uvolnění veškeré alokované paměti na konci jejího běhu jsme použili nástroj `valgrind`. Aplikaci jsme ručně spouštěli s testovacími daty a zjištěné nedostatky odstranili.

5.5.2 Test manuálové stránky

Jazyk manuálových stránek má stanovenou formu. Korektní manuálová stránka proto musí dodržovat správnou syntaxi maker a např. také nesmí obsahovat prázdné řádky. K ověření korektnosti manuálové stránky jsme použili procesor GNU `troff`:

```
groff -z -mdoc
```

5.5.3 Test přenositelnosti

Aplikaci jsme úspěšně otestovali na následujících operačních systémech:

- Debian 7.5 (i386)
- FreeBSD 10.0 (amd64)
- OpenBSD 5.5 (i386, amd64, macppc)
- OpenBSD 5.4 (amd64)
- MacOSX 10.5.8 (intel)
- Ubuntu 14.04 (amd64)

Jelikož při implementaci používáme konstrukty jazyka C++ ve standardu C++11, potřebujeme k sestavení aplikace překladač, který je zdrojový kód v této podobě schopen přeložit. Takový překladač však neexistuje na některých dalších systémech, což do jisté míry omezuje další (pohodlnou) přenositelnost.

Kapitola 6

Rozšiřitelnost

Po implementaci dosavadní funkcionality se nabízí některá přirozená rozšíření, která však přesahují zadání práce i časové možnosti. V této kapitole některá z těchto rozšíření uvedeme a nastíníme jejich možné začlenění do obsluhy aplikace.

6.1 Nalezení důkazu

Tato práce nás naučila *ověřovat* důkazy výrokových formulí v Hilbertově systému. Nezabývali jsme se však otázkou, jakým způsobem důkazy *hledat*. Naši implementaci by velice obohatila právě funkcionalita nalezení důkazu dané (dokazatelné) formule.

Pro nalezení důkazu bychom využili rozhodnutelnosti logiky, tedy skutečnosti, že dokazatelné jsou právě tautologie[3], a ty se dají efektivně poznat. Je totiž zřejmé, že hledat důkaz formule má smysl až tehdy, když zjistíme, že je dokazatelná.

Vztah mezi výpočetní složitostí procedury, která výrokové důkazy *ověřuje* (takovou proceduru jsme implementovali), a složitostí procedury, která výrokové důkazy *nachází*, je hlubokou otevřenou otázkou teoretické informatiky. Je totiž jednou z populárních reformulací tzv. *PNP problému*. Tyto souvislosti jdou ovšem značně nad rámec našeho zadání, a proto jsme se jimi nezabývali. Namísto toho odkazujeme čtenáře na [8] a [9].

6.1.1 Návrh obsluhy

Zavedeme nový přepínač `-F` (find a proof).

Přepínač `-F` je novým cílem aplikace `pl`. Při jeho použití jsou vstupní formule vnímány jako formule, ke kterým chceme nalézt důkaz. Aplikace se jej pokusí nalézt a případně jej vypíše na výstup.

[3]Tautologie – výroková formule, která je vždy pravdivá.

6.2 Gentzenův systém

V části 1.4.1 jsme zmínili, že Hilbertův systém není jediným formálním důkazovým systémem výrokové logiky. Pravidlo modus ponens však zřejmě nejlépe odpovídá postupům našeho uvažování[1]. Existují ovšem i jiná formální odvozovací pravidla, např. *pravidlo rezoluce*, které zavádí *Gentzenův systém*.

Stávající implementaci bychom mohli rozšířit právě o tento důkazový systém. Aplikace by tak mohla důkazy v tomto systému ověřovat, minimalizovat nebo je mezi těmito systémy převádět.

6.2.1 Pravidlo rezoluce

Pravidlo rezoluce používá jazyk logických spojek odlišný od jazyka Hilbertova systému. Je zavedeno následovně[2]:

Definice 5. Z formulí $\varphi \vee \alpha$ a $\neg\psi \vee \vartheta$ odvoď formuli $\psi \vee \vartheta$.

Každou takto odvozenou formuli nazýváme *rezolventou*.

6.2.2 Návrh obsluhy

6.2.2.1 Režim důkazového systému

Zavedeme nový přepínač `-r` (deduction rule).

Přepínač `-r` s hodnotami `hilbert` a `gentzen` je rozšířením stávající funkcionality práce s důkazy. Určuje, kterému důkazovému systému náleží vstupní důkaz, který chceme ověřit nebo minimalizovat.

6.2.2.2 Překlad důkazů

Zavedeme nový přepínač `-C` (convert).

Přepínač `-C` s hodnotami `hilbert` a `gentzen` je novým cílem aplikace. Při jeho použití aplikace přeloží vstupní důkaz do podoby daného výstupního důkazového systému. Protože systémy Hilbertův a Gentzenův jsou ekvivalentní[2] a překlad důkazů v těchto systémech je rutinní[1], úspěch překladu závisí pouze na korektnosti vstupního důkazu.

6.3 Paralelní práce

Zadání této bakalářské práce vychází z velkého množství požadavků na tutéž aplikaci `pl`. Z těchto požadavků byly sestaveny tři zadání bakalářských prací, které byly vypracovávány paralelně. Funkcionalita ostatních částí se měla týkat také např. sémantiky výrokové logiky, kterou jsme se v této práci nezabývali. Jaké možné pokračování této práce, stejně jako těch dvou dalších, se proto nabízí sjednocení funkcionality těchto tří aplikaci do jediné, což je původní motivace.

Závěr

Tato práce mi dala možnost uplatnit mnohé znalosti, které mi předala Fakulta informačních technologií ČVUT v Praze během mého bakalářského studia. Zejména jsem uplatnil programovací dovednost v jazyce C++, dokumentaci kódu ve stylu Doxygen, sestavování programu pomocí nástroje `make`, sazbu textu v LaTeX a znalost v oblasti výrokové logiky. Své dosavadní znalosti jsem navíc rozšířil. Nových znalostí jsem nabyl konkrétně v oblasti tvorby manuálových stránek pro systém UNIX pomocí balíčku maker `mdoc`. V neposlední řadě jsem si poprvé zkusil literární činnost v tomto rozsahu.

Literatura

[1] Sochor, A.: *Klasická matematická logika*. Praha: Univerzita Karlova v Praze, 2001, ISBN 80-246-0218-0.

[2] Starý, J.: Text a sbírka příkladů k přednášce předmětu Matematická logika. [online], [Citováno 2014-06-12]. Dostupné z: `http://users.fit.cvut.cz/~staryja2/BIMLO/matematicka-logika.pdf`

[3] *The C++ Resources Network*. [Citováno 2014-05-10]. Dostupné z: `http://www.cplusplus.com/`

[4] *GNU Make Manual*. [Citováno 2014-05-10]. Dostupné z: `https://www.gnu.org/software/make/manual/`

[5] *Doxygen*. [Citováno 2014-05-10]. Dostupné z: `http://www.stack.nl/~dimitri/doxygen/`

[6] Corderoy, R.: The Text Processor for Typesetters. [online], [Citováno 2014-05-10]. Dostupné z: `http://www.troff.org/`

[7] Dzonsons, K.: *Semantic markup language for formatting manual pages*. [Citováno 2014-05-10]. Dostupné z: `http://mdocml.bsd.lv/mdoc.7.html`

[8] Krajíček, J.: *Bounded arithmetic, propositional logic, and complexity theory*. Cambridge University Press, 1995, ISBN 0-521-45205-8.

[9] Cook, S.: The P versus NP Problem. [online], [Citováno 2014-06-25]. Dostupné z: `http://www.claymath.org/sites/default/files/pvsnp.pdf`

Příloha **A**

Seznam použitých zkratek

ASCII American Standard Code for Information Interchange

GNU GNU's Not Unix

HTML HyperText Markup Language

STL Standard Template Libraries

Příloha B

Obsah přiloženého CD

- `src` ..... adresář se zdrojovou formou práce
  - `impl` ..... adresář se zdrojovou formou implementace
    - `doc` ..... adresář se soubory HTML dokumentace
    - `src` ..... adresář se soubory zdrojového kódu
    - `test` ..... adresář se soubory pro testování
    - `Doxyfile` ..... soubor konfigurace nástroje Doxygen
    - `Makefile` ..... soubor konfigurace nástroje make
    - `pl.1` ..... soubor zdrojové formy manuálové stránky
    - `test.sh` ..... soubor testovacího skriptu pro shell
  - `thesis` ..... adresář se zdrojovou formou textu práce
- `text` ..... adresář s texty práce

Příloha C

Instalační příručka

Adresář `src/impl` uložte na svůj disk a přejděte do pracovního adresáře `impl`. Následuje popis cílů nástroje GNU `make`, které nyní lze spouštět.

build Sestaví aplikaci. Binární formu aplikace `pl` naleznete po spuštění tohoto cíle v adresáři `out`. K úspěšnému sestavení je nutné, aby váš `gcc` kompilátor podporoval standard C++11.

clean Odstraní veškeré výstupy vyprodukovaných při práci s `make`. Spuštění tohoto cíle vrátí adresář `impl` do původního stavu.

doc Vygeneruje programátorskou dokumentaci ve formátu HTML. Dokumentaci naleznete po spuštění tohoto cíle v adresáři `doc`. Tento cíl vyžaduje přítomnost nástroje `doxygen` v systému.

install Sestaví a nainstaluje do systému aplikaci a manuálovou stránku. K úpravě instalačního umístění změňte proměnnou `PREFIX` v souboru `Makefile` na požadovanou hodnotu. Pokud instalujeme do systémového umístění, je k instalaci nutné mít oprávnění uživatele `root`. Po instalaci je aplikace dostupná odkudkoliv pomocí příkazu `pl` a manuálová stránka pomocí příkazu `man pl`.

test Sestaví a otestuje aplikaci za využití testovacích souborů v adresáři `test`. Testování končí hlášením o úspěšnosti testů. Soubory vyprodukované testy naleznete po spuštění tohoto cíle v adresáři `out`.

uninstall Odinstaluje aplikaci a manuálovou stránku ze systému, pokud do něj předtím byla nainstalována spuštěním cíle `install`.