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.

22. 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 22. 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. Jeho 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.

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)[1].

Příklad 1. např. 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

  Negace $\neg$

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

- 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[1].

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.

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 schemata 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[2]. 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.

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 $\vdash \varphi$[2].

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[2]. Na druhou stranu nám 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 Optimalizace důkazů

Protože se v této práci také zabýváme optimalizací, resp. minimalizací, důkazů, definujeme nyní důkaz, který nazveme *optimálním*.

Definice 4. Buď $\varphi_1, \ldots, \varphi_n$ posloupnost formulí tvořící důkaz formule φ_n. Důkaz nazveme optimálním, pokud každá formule φ_i, kde $i < n$, je nezbytná k důkazu formule φ_n, tj. je jejím přímým či nepřímým svědkem za použití odvozovacích pravidel.

Takový důkaz tedy neobsahuje zbytečné vnořené důkazy a neobsahuje 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 optimálního důkazu se tedy již nedá odstranit žádná formule, důkaz by jinak přestal být korektním. Neznamená to však, že neexistuje jiný důkaz téže formule, který by byl např. úspornější. Hledání takového důkazu však opět přesahuje rozsah této práce.

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 Hilbertova systému. Požadavkem nad rámec zadání je optimalizace 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 buď ze standardního vstupu anebo 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. Naopak, nebudeme využívat prostředky třetích stran. Zejména využijeme knihovny STL obsahující šablony základních kontejnerů.

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

jsou třeba zkompilovat v případě změny zdrojového kódu. K řízení sestavení tento nástroj využívá konfiguračního souboru `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) cíly bývá kompilace konkrétního zdrojového souboru[4].

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. LaTeXslouží 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 v sekci *mdoc(7)* manuálových stránek[7].

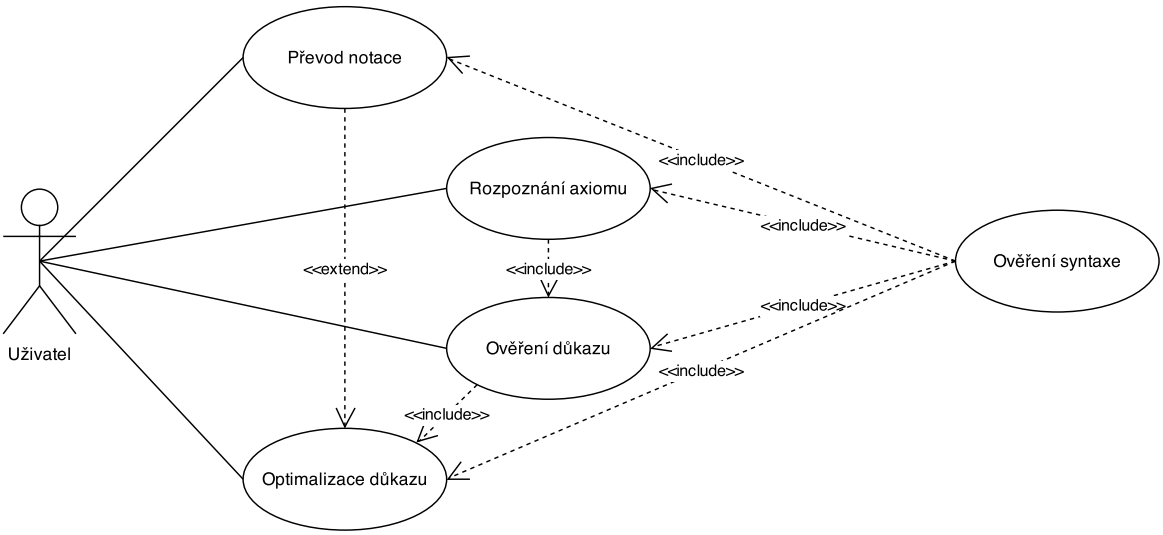
Aby manuálová stránka byla dostupná pomocí příkazu `man`, musí splňovat určité náležitosti. Název souboru manuálové stránky má následující formát:

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

V našem případě tedy `pl.1`. Stejnojmenný archiv souboru s manuálovou stránkou je umístěn v adresáři, ve kterém příkaz `man` implicitně hledá manuálové stránky[2] pod adresářem příslušícím dané manuálové sekci. V našem

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

Obrázek 3.1: Diagram případů užití



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.

3.2 Případy užití

Na základě funkčních požadavků nastíníme možné případy užití aplikace 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 TEX.

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.

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,

Tabulka 3.1: Výstupní jazyky spojek

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

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 Optimalizace důkazu

Důkaz (z předpokladů) je nejprve ověřen stejně jako v předchozím případě. Následně, pokud je to možné, je optimalizován do nejúspornější možné podoby `??`. 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 logické operátory negace, konjunkce, disjunkce, implikace a ekvivalence. Pro infixní notaci formulí navíc zavádíme symboly závorek`()` 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 formule vstupní posloupnosti 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 *syntax* formulí s hodnotami *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 *syntax* formulí s hodnotami *prefix*, *infix* a *postfix*. Implicitní hodnota je *infix*.

-O n (proof optimizer) Optimalizuje důkaz ze vstupní posloupnosti formulí. Nepovinný parametr *n* vyjadřuje počet formulí množiny předpokladů, které na vstupu předcházejí skutečnému důkazu. Implicitní hodnota je 0, tedy důkaz není implicitně optimalizován jako důkaz z předpokladů.

-P n (proof checker) Ověří vstupní posloupnost formulí jako důkaz. Nepovinný parametr *n* vyjadřuje počet formulí množiny předpokladů, které na vstupu předcházejí skutečnému důkazu. Implicitní hodnota je 0, tedy důkaz není implicitně ověřován jako důkaz z předpokladů.

-s (strict) Povolí zastavení vykonávání programu při prvním výskytu nekorektního syntaktického zápisu formule. Přepínače *-O* a *-P* takové chování povolují automaticky, protože nemá smysl dále optimalizovat či ověřovat evidentně nekorektní důkaz.

Axiom checker, *proof checker* a *proof optimizer* budeme dále nazývat cíly aplikace. Volba těchto cílů je nepovinná, avšak je vždy omezena na jediný.

3.3.4 Manuálová stránka

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

NAME Název a krátký popis aplikace.

SYNOPSIS Výčet podporovaných přepínačů.

DESCRIPTION Podrobný popis aplikace.

Language Popis vstupního jazyka.

Input Popis formy vstupu.

Output Popis formy výstupu.

Options Popis podporovaných přepínačů.

EXIT STATUS Popis návratových hodnot.

EXAMPLES Vzorové příklady užití.

HISTORY Historický kontext aplikace.

AUTHOR Informace o autorovi.

Kapitola 4

Implementace

V této kapitole popíšeme, jak ty které části zdrojového kódu implementují jednotlivé součásti systému.

4.1 Hlavní program

Hlavní program aplikace představuje funkce `main`.

main Inicializací instance třídy `Configuration` nejprve dojde ke konfiguraci aplikace dle parametrů příkazové řádky. V případě úspěchu je proveden cíl aplikace, jehož návratová hodnota je zároveň návratovou hodnotou 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 tříd, funkcí a metod. Následující text koresponduje s dokumentací zdrojového kódu.

4.2.1 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 výrazového 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.

4.2.1.1 Funkce 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.

4.2.1.2 Funkce 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 úrovní (větví) výrazového stromu. Každá taková úroveň zpracování může mít následující stav:

UNARY Naposledy byl nastaven unární operátor.

BLANK Naposledy byla otevřena nová větev stromu.

FIRST_OPERAND Naposledy byl nastaven první operand.

BINARY Naposledy byl nastaven binární operátor.

LAST_OPERAND Naposledy 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.

4.2.1.3 Funkce parsePostfix

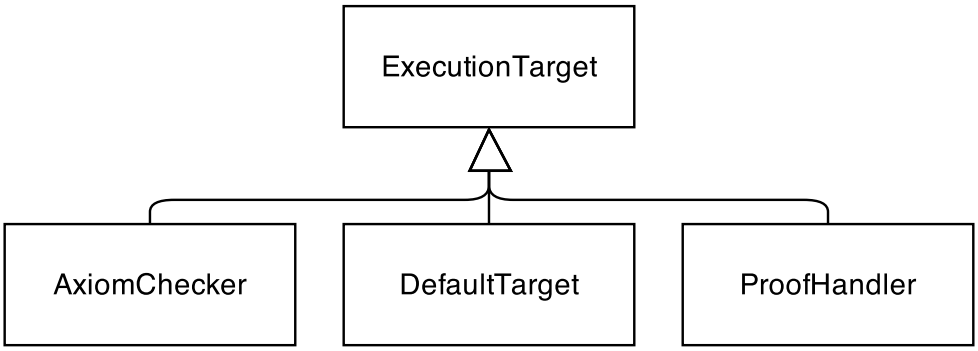
Postfixní parser ukládá přijaté elementární formule na zvláštní zásobník s formulemi. Ve chvíli, kdy je přijat logický operátor, dojde k sejmutí příslušného počtu formulí ze zásobníku, přičemž tyto jsou zprava nastaveny jako operandy přijatého operátoru. Ten je následně uložen na zásobník.

4.2.2 Třída Configuration

Tato třída představuje konfiguraci aplikace. Obsahuje proměnné, které řídí průběh programu.

Konstruktor Konstruktor pomocí funkce `getopt` z knihovny `unistd.h` zpracovává přepínače z příkazové řádky, podle kterých nastavuje třídní proměnné.

Obrázek 4.1: Diagram tříd rodiny ExecutionTarget



4.2.3 Třída ExecutionTarget

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

executeTarget Tato abstraktní metoda provede cíl v závislosti na konfiguraci aplikace předané instancí třídy `Configuration` a vrátí návratovou hodnotu aplikace. Základem všech implementací této metody je cyklické zpracování formulí vstupní posloupnosti, které 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 chybné syntaxe přijaté formule cyklus předčasně zastavují.

4.2.3.1 AxiomChecker

Tato třída představuje cíl axiom checker.

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

4.2.3.2 DefaultTarget

Tato třída představuje stav spuštění aplikace bez cíle.

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

4.2.3.3 ProofHandler

Tato třída představuje cíle proof checker a proof optimizer.

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

Přijatá formule je postupně ověřena jako axiom, jako prvek teorie a jako formule odvoditelná v rámci nastaveného 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, aplikace skončí neúspěchem.

Po úspěšném zpracování všech vstupních formulí následuje iterativní algoritmus optimalizace důkazu. Inicializujeme zásobník prvků třídy `ProofMember` a uložíme na něj poslední člen zásobníku. Dokud není zásobník prázdný prohlašujeme jeho vrchol členem optimálního 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 optimálním a aplikace končí neúspěchem.

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

4.2.4 Třída Formula

Struktura výrokové formule se dá vyjádřit *stromem* a proto jsme ji implementovali jako *výrazový 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.

Metody print Abstraktní metody `printPrefix`, `printInfix` a `printPostfix` jsou metody vracející řetězec s formulí v prefixní, infixní a postfixní notaci a v daném jazyce logických spojek.

equals Tato abstraktní metoda zjistí, zdali se daná formule shoduje s touto.

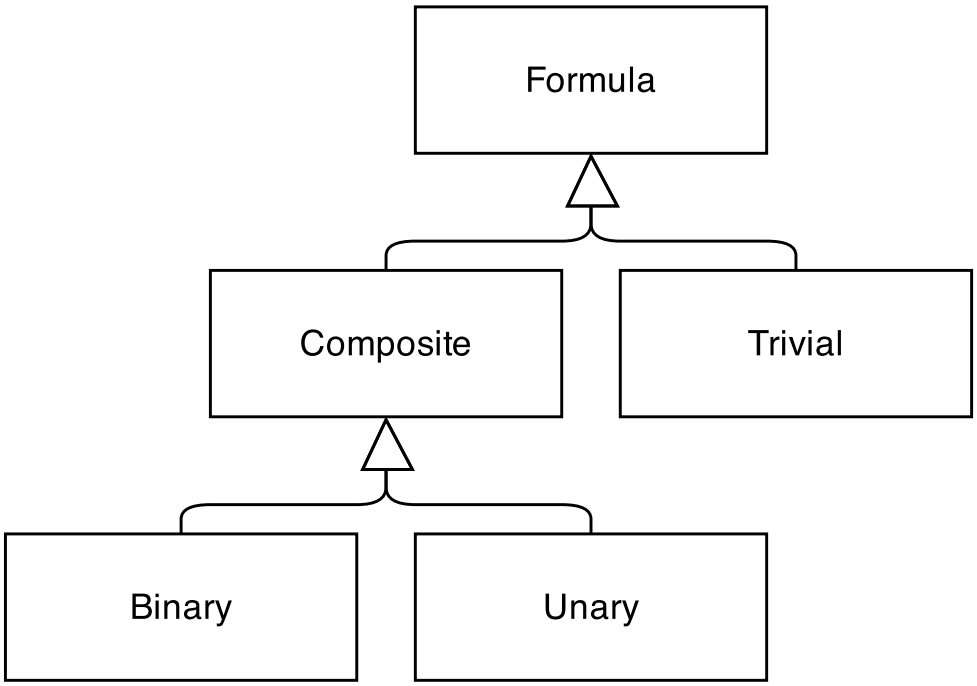
matches Tato abstraktní metoda zjistí, zdali daná formule vyhovuje tvaru určenému touto formulí.

4.2.4.1 Třída Composite

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

Metody set Tyto abstraktní metody obsadí první (`setFirst` nebo poslední (`setLast` operand zleva této formule danou formulí. Návratová hodnota značí, zdali po provedení této operace jsou již všechny operandy obsazené.

Obrázek 4.2: Diagram tříd rodiny Formula



4.2.4.2 Třída Trivial

Tato třída představuje triviální formuli, tedy 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 prostřednictvím dané formule s formulí, která je substitucí za reprezentující znak této formule, metodou `equals`. Tyto substituce se vyhledávají v dané mapě substitucí. 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.

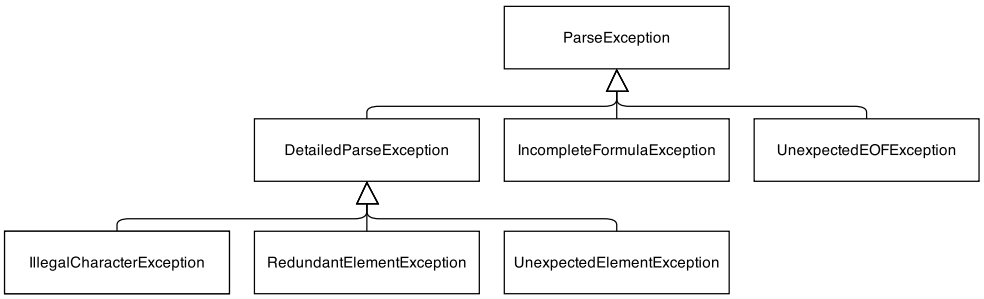
4.2.4.3 Třída Binary

4.2.4.4 Třída Unary

4.2.5 Třída ParseException

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

Obrázek 4.3: Diagram tříd rodiny ParseException



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 (kromě bílých znaků).

4.2.5.2 Výjimka IncompleteFormulaException

Tento typ výjimky představuje případ nekompletní formule.

4.2.5.3 Výjimka UnexpectedEOFException

Tento typ výjimky představuje případ konce vstupu v místě, které nestanovuje korektní forma vstupu.

4.2.5.4 Výjimka IllegalCharacterException

Tento typ výjimky představuje případ užití nepovoleného znaku.

4.2.5.5 Výjimka RedundantElementException

Tento typ výjimky představuje případ výskytu nadbytečných prvků formule.

4.2.5.6 Výjimka UnexpectedElementException

Tento typ výjimky představuje případ nekorektního infixního zápisu formule.

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 vyhrazenou pro algoritmus optimalizace důkazu.

4.2.7 Třída ProofSystem

Tato abstraktní třída představuje důkazový systém výrokové logiky. Ačkoliv tato práce implementuje Hilbertův systém jako jediný konkrétní důkazový systém, třídu `ProofSystem` jsme zavedli z důvodu rozšiřitelnosti aplikace. Objekt této třídy obsahuje axiomy důkazového systému. Hierarchii tříd rodiny `ProofSystem` znázorňuje diagram 4.4.

isAxiom Tato metoda prostřednictvím metody `matches` ověří, zdali je daná formule instancí některého z axiomů důkazového systému.

isDeducible Tato abstraktní metoda ověří, zdali je daná formule v rámci daného důkazu odvoditelná pravidly důkazového systému. Metoda je abstraktní, protože každý konkrétní důkazový systém odvozuje formule jiným způsobem.

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 Tato metoda ověří, zdali je daná formule v rámci daného důkazu odvoditelná pravidlem modus ponens. Implementace spočívá v procházení jednotlivých formulí daného důkazu, přičemž na jednu z formulí je vždy nahlíženo jako na předpoklad pravidla modus ponens a na druhou jako jeho implikaci. Daná formule je odvoditelná, je-li nalezena dvojice formulí vyhovující právě tomuto tvaru.

4.2.8 Třída SyntaxException

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 tříd rodiny `SyntaxException` znázorňuje diagram 4.5.

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

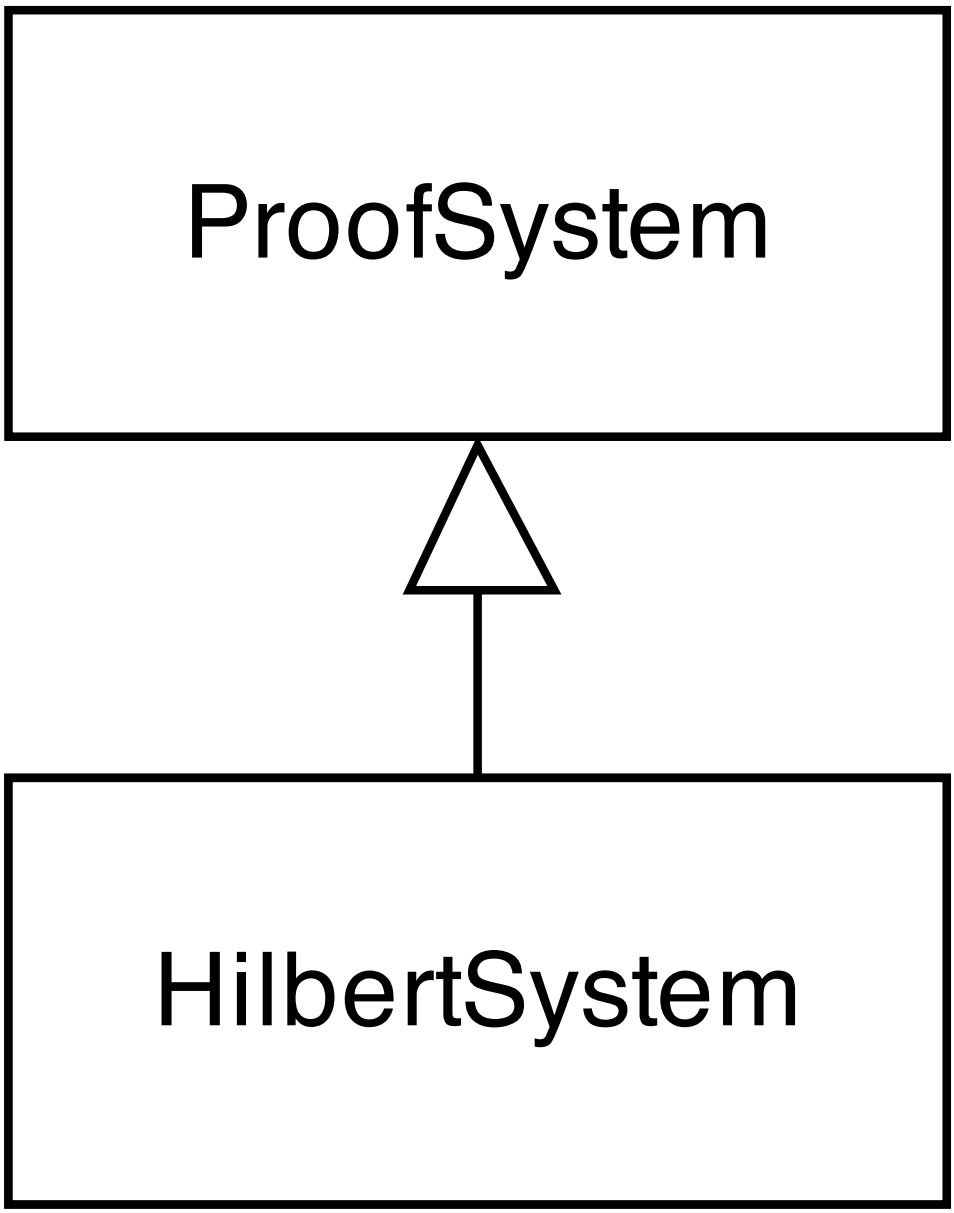
4.2.8.1 Výjimka IllegalOptionException

Tento typ výjimky představuje případ použití nepovoleného přepínače.

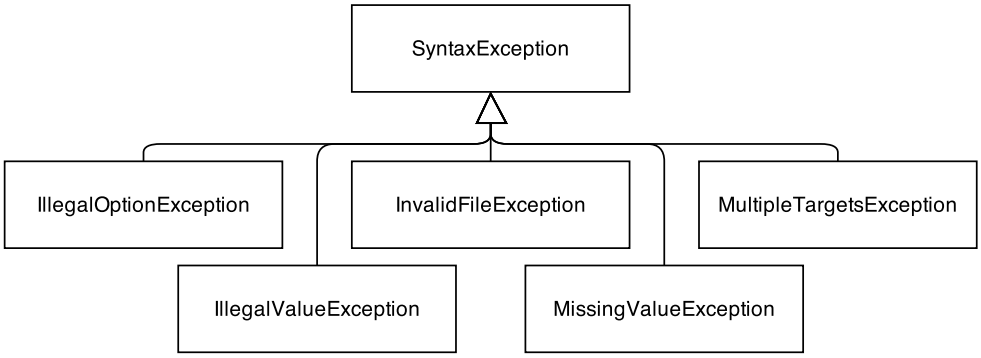
4.2.8.2 Výjimka IllegalValueException

Tento typ výjimky představuje případ zadání nepovolené hodnoty přepínače.

Obrázek 4.4: Diagram tříd rodiny ProofSystem



Obrázek 4.5: Diagram tříd rodiny SyntaxException



4.2.8.3 Výjimka InvalidFileException

Tento typ výjimky představuje případ uvedení nepoužitelného vstupního souboru.

4.2.8.4 Výjimka MissingValueException

Tento typ výjimky představuje případ neuvedení povinné hodnoty přepínače.

4.2.8.5 Výjimka MultipleTargetsException

Tento typ výjimky představuje případ nastavení 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 3.2:

- Zpracování vstupu.
  - Ověření syntaxe.
  - Převod notace.
- Rozpoznání axiomu.
- Ověření důkazu.
- Optimalizaci 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 je následující posloupnost formulí:

```
A
-A
(A.A)
-(-A.-A)
(A.((A.A).A))
-(-A.-(-(-A.-A).-A))
```

5.1.1.2 Výstupní data

Výstupními testovacími daty 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 všechny 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.((A.A).A)
(A.(A.A.A))
A.((A.A).A)
(A&((A&A)&A))
(A.((A.A).A))
```

5.1.2.2 Výstupní data

Výstupními testovacími daty je následující posloupnost hlášení:

```
Incomplete formula.
Unexpected element '.' at position 8.
Unnecessary element '.' at position 2.
Illegal character '&' at position 3.
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 podobným 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 $A \Rightarrow A$ 4.

5.3.1 Pozitivní test

Pozitivní test ověřuje korektní ověření značně neoptimálního avšak korektního důkazu.

5.3.1.1 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)
```

5.3.1.2 Výstupní data

Výstupními testovacími daty je následující posloupnost hlášení:

```
Axiom of type 1.
Axiom of type 1.
Axiom of type 1.
Axiom of type 2.
Deducible using formulas 1 4 as witnesses.
Axiom of type 2.
Axiom of type 1.
Axiom of type 2.
Deducible using formulas 1 4 as witnesses.
Axiom of type 2.
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 Optimalizace důkazu

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

5.4.1 Pozitivní test

Pozitivní test ověřuje korektní optimalizaci značně neoptimálního 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 optimálního 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 optimal.
```

5.5 Další testy

Kromě testů korektní funkčnosti aplikace jsme dále provedli následující ověření.

5.5.1 Test uvolnění paměti

Programy psané v jazyce C++ vyžadují explicitní správu paměti programátorem. Pro detekci paměťových úniků za běhu aplikace a ověření uvolnění veškeré alokované paměti na konci jejího běhu jsme použili nástroj `valgrind`. Zajistili jsme tak poctivou správu paměti aplikace v její konečné podobě.

5.5.2 Test manuálové stránky

Jazyk manuálových stránek má stanovenou formu. Korektní manuálová stránka proto musí dodržovat 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 systémech:

- Linux

- OpenBSD

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 ovšem otázkou, jakým způsobem důkazy hledat. Naší 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.

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.

6.2 Gentzenův systém

Hilbertův systém není jediným formálním důkazovým systémem výrokové logiky. Pravidlo modus ponens však zřejmě nejlépe odpovídá postupům našeho

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

uvažování[1]. Existují ovšem i jiná formální odvozovací pravidla, např. *pravidlo rezoluce*, které zavádí *Gentzenův systém*. Ten je formálně stejně silný jako systém Hilbertův[2].

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

6.2.1 Pravidlo rezoluce

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

Definice 5. Z formulí $\varphi \vee \alpha$ a $\neg\psi \vee \vartheta$ odvoď formuli $\psi \vee \vartheta$[1].

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 možnými hodnotami `hilbert` a `gentzen` je rozšířením stávající funkcionality práce s důkazy. Určuje, kterému důkazovému systému náleží vstupní důkaz.

6.2.2.2 Převod důkazů

Zavedeme nový přepínač `-C` (convert).

Přepínač `-C` s možnými hodnotami `hilbert` a `gentzen` je novým cílem aplikce. Při jeho použití aplikace převede vstupní důkaz do podoby daného výstupního důkazového systému. Protože systémy Hilbertův a Gentzenův jsou ekvivalentní, úspěch převodu závisí pouze na korektnosti vstupního důkazu.

6.3 Paralelní práce

Zadání této bakalářské práce vychází z velkého množství požadavků na tutéž aplikaci `pl`. Její funkcionalita se měla týkat také sémantiky výrokové logiky, kterou jsme se v této práci nezabývali. Z těchto požadavků byly sestaveny tři zadání bakalářských prací, které byly vypracovávány paralelně. Jaké možné pokračování této práce, stejně jako těch dvou dalších, se proto nabízí sjednocení funkcionality těchto tří aplikaci do jediné tak, jak bylo zamýšleno zcela původně.

Závěr

Tato implementační práce mi dala možnost uplatnit mnohé znalosti, které mi dala 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 nástrojem `make`, sazbu textu v jazyce LaTeXa znalost v oblasti výrokové logiky. Nicméně, své dosavadní znalosti jsem také rozšířil. Nových znalostí jsem nabyl konkrétně v oblasti tvorby manuálové stránky pro systém UNIX v pomocí maker MDOC. V poslední řadě jsem si poprvé zkusil literární činnost ve velkém rozsahu. Za veškeré nabyté zkušenosti jsem vděčný. Práce na tomto projektu mě velice těšila.

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`

Příloha A

Seznam použitých zkratek

ASCII American Standard Code for Information Interchange

ČVUT České vysoké učení technické

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