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.

18. 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 18. 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 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. Například 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 [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 $((\varphi \Rightarrow (\varphi \Rightarrow \varphi)) \Rightarrow (\varphi \Rightarrow \varphi))$

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

modus ponens $(\varphi \Rightarrow \varphi)$

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

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

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 například alternativní zápis cyklu `for`, který abstrahuje od iterátorů při procházení standardních kontejnerů.

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 [3].

3.1.2.1 GNU make

My použijeme nástroj `gmake` (GNU make), který je jednou z implementací klasického `make` rozšířenou o pokročilé funkce. Z nich konkrétně využijeme např. *pattern rules*, *wildcard characters* či funkce pro práci z řetězci [3]

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. My budeme dokumentaci zapisovat především do hlavičkových souborů `.hpp`. 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 [4].

3.1.4 Mdoc

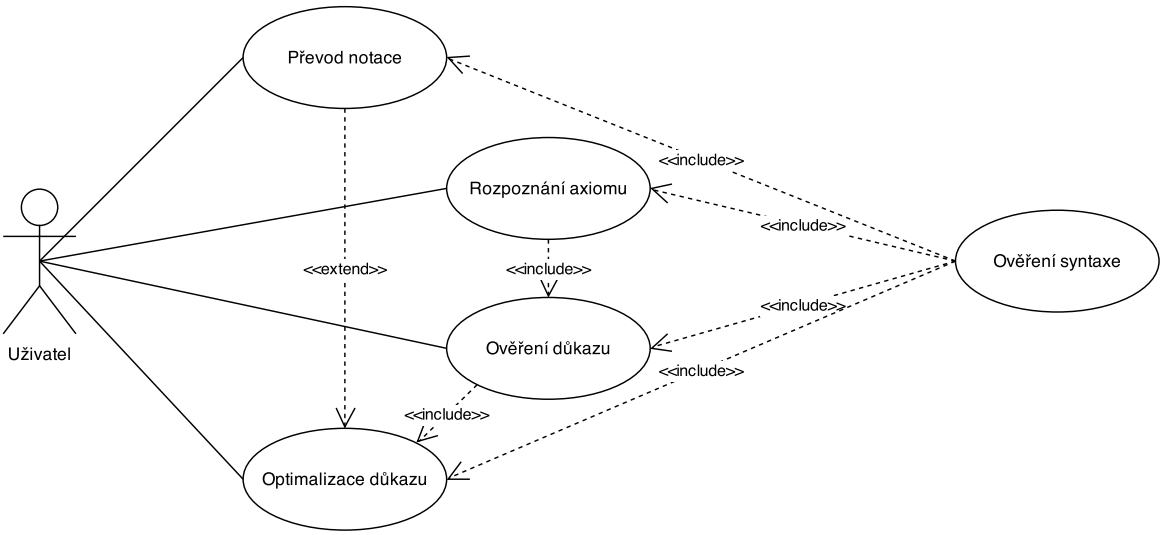
Manuálové stránky systému UNIX lze psát ve speciálním značkovacím jazyce. Takový jazyk je ve skutečnosti balíček maker pro jazyk *troff*. Troff pochází ze 70. let a podobně jako např. LaTeXslouží k sazbě textu. Jazyk zpracovává stejnojmenný textový procesor, který podporuje právě zmíněné balíčky maker pro tento jazyk. Jednotlivá makra, podobně jako například značky jazyka HTML, formátují daný obsah do požadované podoby [5]. 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 mimo jiné v sekci *mdoc(7)* manuálových stránek [6].

3.2 Analýza

3.2.1 Případy užití

Na základě funkčních požadavků nastíníme možné případy užití aplikace 3.1.

Obrázek 3.1: Diagram případů užití



3.2.1.1 Převod notace

Elementární funkcionalitou aplikace je převod posloupnosti 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.

3.2.1.2 Rozpoznání axiomu

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

3.2.1.3 Ověření důkazu

Tato funkce je hlavním cílem této práce. Na vstupní posloupnost formulí je nahlíženo jako na posloupnost formulí důkazu, přičemž aplikace ověří, zdali je důkaz korektním. Je možné ověřovat důkaz jako důkaz z předpokladů. Tehdy je prvních n formulí vstupu považováno za prvky teorie T, poté následují formule vlastního důkazu. Výstupem aplikace jsou podrobnosti odůvodňující platnost, popř. neplatnost formule v důkazu.

3.2.1.4 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.

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`

3.3 Návrh

3.3.1 Forma vstupu

Textový vstup aplikace `pl` se skládá z posloupnosti formulí, které jsou odděleny řádkovým zalomením, znakem
`n`, přičemž formule poslední je jím zakončena. Zápis formulí užívá znaků `A-Z` pro elementární výroky. Tím je vstup omezen na 26 různých elementárních výroků, což pro naše účely plně postačuje. Symboly `-.+>=` symbolizují logické operátory negace, konjunkce, disjunkce, implikace, ekvivalence. Speciálně pro infixní notaci formulí zavádíme navíc symboly závorek `()` pro určování přednosti operací.

3.3.2 Forma výstupu

Kromě výstupní notace lze zvolit i jeden ze tří výstupních forem 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 ověrován 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.

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 Vysvětlení významu přepínačů.

EXIT STATUS Vysvětlení významu návratových hodnot.

EXAMPLES Praktické příklady použití.

HISTORY Historický kontext aplikace.

AUTHOR Informace o autorovi.

Kapitola 4

Implementace

V této kapitole popíšeme, jak ty které části zdrojového kódu implementují doposud zmíněné prvky systému.

4.1 Sestavení

Cíle v našem `Makefile` jsme rozdělili do dvou kategorií: hlavní a podpůrné. Hlavní cíle jsou určeny uživateli aplikace, podpůrné cíle jsou využívány hlavními a slouží k rozkladu složitějšího úkonu na jednodušší celky. Abychom si proces sestavení aplikace s pomocí nástroje `make` maximálně usnadnili, nakonfigurovali jsme jej tak, abychom nemuseli do souboru `Makefile` zasahovat po každé změně zdrojových souborů (např. přejmenování souboru nebo změna závislostí na hlavičkových souborech). Nástroj `make` tedy aplikaci sestaví dynamicky, podle aktuální podoby zdrojové formy.

Námi implementované cíle mají následující účel:

build Sestaví aplikací.

clean Odstraní veškeré výstupy.

doc Vygeneruje dokumentaci zdrojového kódu.

test Otestuje aplikaci.

install Nainstaluje aplikaci do systému.

uninstall Odinstaluje aplikaci ze systému.

4.1.1 Cíl build

Speciálním podpůrným cílem je navíc pravidlo definující způsob kompilace jednotlivých zdrojových souborů. Toto pravidlo je v procesu kompilace klíčovým, protože dynamicky vyvolává kompilaci souborů s příponou `.cpp` a my

tak nemusíme pro každý zdrojový soubor `.cpp` zvlášť definovat pravidlo pro jeho kompilaci.

4.1.2 Cíl clean

4.1.3 Cíl doc

4.1.4 Cíl test

4.1.5 Cíl install

4.1.6 Cíl uninstall

4.2 Hlavní program

Hlavní program aplikace představuje funkce `main`.

Nejdříve se program pokusí inicializovat instanci třídy `Configuration`. V případě úspěchu je proveden cíl aplikace uložený v této instanci, jehož návratová hodnota je vrácena jako návratová hodnota aplikace. V případě, že se nepodaří aplikaci nakonfigurovat, je zachycena výjimka s podrobnostmi o chybě a běh aplikace končí neúspěchem.

4.3 Komponenty

4.3.1 Třída Configuration

Tato třída představuje konfiguraci aplikace. Její konstruktor pomocí funkce `getopt` z knihovny `unistd.h` zpracovává přepínače z příkazové řádky a nastavuje třídní proměnné, podle kterých je řízen průběh aplikace. Tyto proměnné jsou pak dostupné pomocí getterů.

4.3.2 Třída SyntaxException

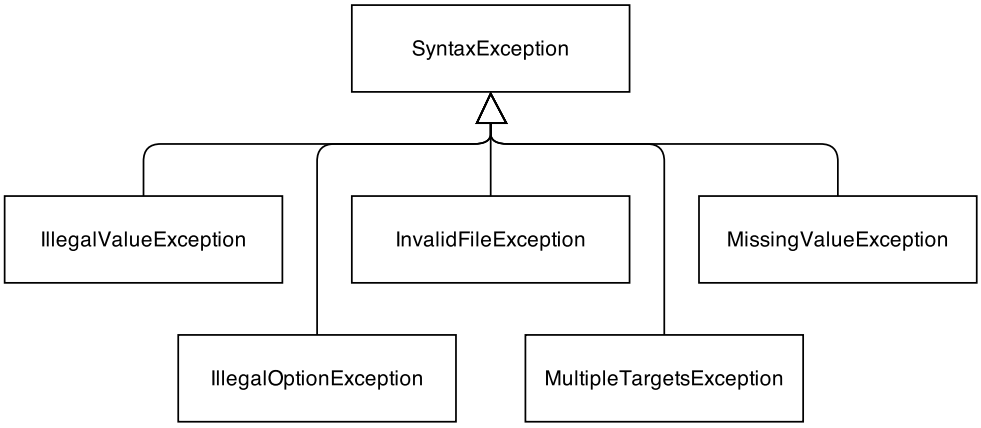
Tato obecná třída představuje výjimku vrženou při konfiguraci aplikace v případě nesprávné syntaxe příkazové řádky. Objekt této třídy si uchovává záznam o tom, který přepínač syntaktickou chybu způsobil, a řetězec s detaily o této chybě. Metoda `getMessage` tyto informace vrací v podobě řetězce s přehledným chybovým hlášením.

Hierarchii tříd rodiny `SyntaxException` ilustruje diagram 4.1.

4.3.2.1 Výjimka IllegalOptionException

Výjimka tohoto typu je vržena v případě, že byl použit nepovolený přepínač.

Obrázek 4.1: Diagram tříd rodiny SyntaxException



4.3.2.2 Výjimka IllegalValueException

Výjimka tohoto typu je vržena v případě, že byla zadána nepovolená hodnota přepínače.

4.3.2.3 Výjimka InvalidFileException

Výjimka tohoto typu je vržena v případě, že byl uveden nepoužitelný vstupní soubor.

4.3.2.4 Výjimka MissingValueException

Výjimka tohoto typu je vržena v případě, že nebyla uvedena povinná hodnota přepínače.

4.3.2.5 Výjimka MultipleTargetsException

Výjimka tohoto typu je vržena v případě, že bylo nastaveno více cílů k provedení.

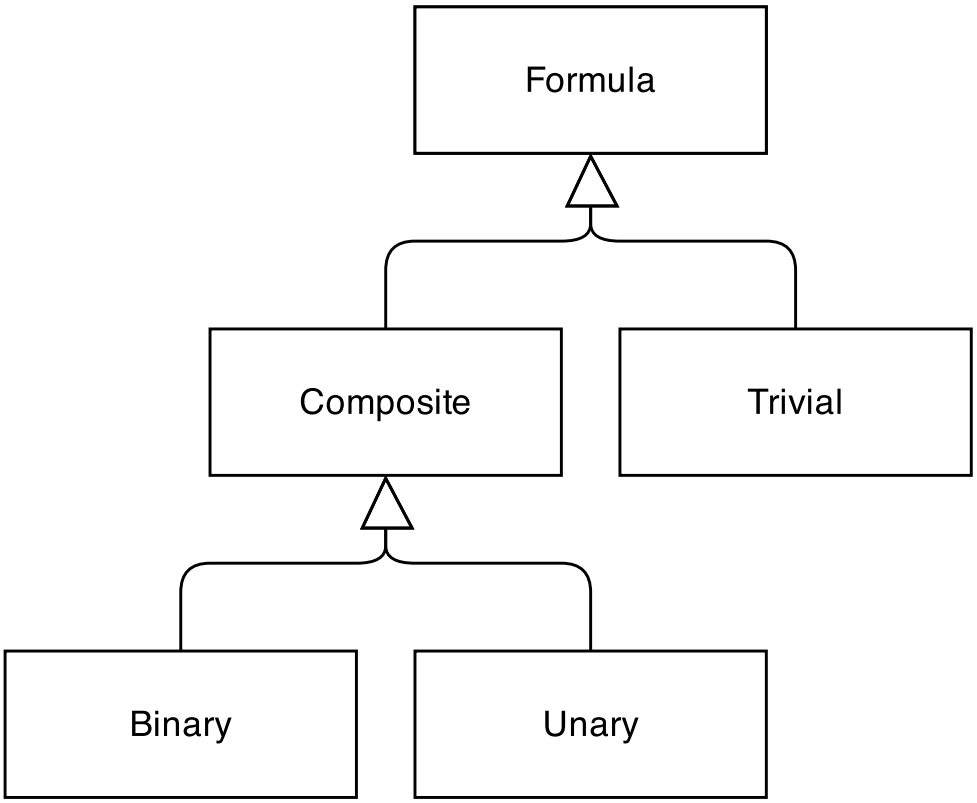
4.3.3 Třída ExecutionTarget

Tato třída představuje cíl aplikace. Disponuje pouze metodou `executeTarget`, která v závislosti na konfiguraci aplikace cíl provede a vrátí návratovou hodnotu aplikace.

4.3.4 Třída Formula

Výroková formule je výraz se stromovou strukturou, proto je také jako strom implementována. Tato abstraktní třída představuje uzel tohoto stromu a její návrh odpovídá definici výrokové formule 1. Objekt této třídy obsahuje znak,

Obrázek 4.2: Diagram tříd rodiny Formula



který reprezentuje dílčí složenou, resp. elementární, formuli jakožto operátor, resp. výrok. Zdůrazníme, že výsledkem našeho řešení není *sémantický*, ale *výrazový* strom.

Hierarchii tříd rodiny `Formula` ilustruje diagram 4.2.

4.3.5 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. Syntaktická analýza postupuje po jednotlivých znacích vstupního proudu, aby mohla být přerušena již v místě případné syntaktické chyby. Funkce v případě úspěchu vrací kořen výrazového stromu formule.

Základem syntaktické analýzy je cyklus, ve kterém se okamžitě zpracovává jediný znak vstupu. Nyní částečně popíšeme algoritmy syntaktické analýzy korektního vstupu parsovacích funkcí.

4.3.5.1 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.

4.3.5.2 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.

4.3.5.3 Funkce parsePostfix

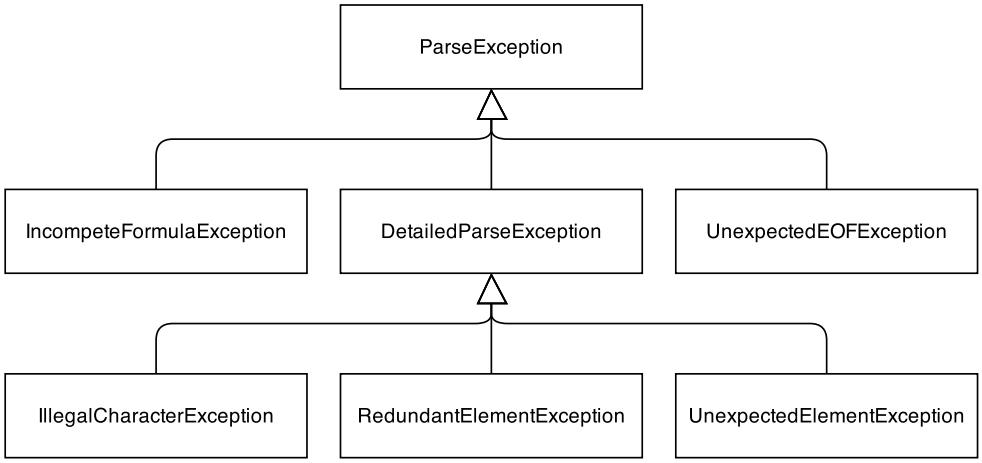
Postfixní parser ukládá přijaté elementární formule na zvláštní zásobník s formulemi. Ve chvíli, kdy je přijat logický operátor, dojde k sejmutí příslušného počtu formulí ze zásobníku, přičemž tyto jsou zprava nastaveny jako operandy přijatého operátoru. Ten je následně uložen na zásobník.

4.3.6 Třída ParseException

Tato obecná třída představuje výjimku vrženou při syntaktické analýze nekorektního vstupu. Objekt této třídy obsahuje řetězec s popisem příčiny syntaktické chyby. Metoda `getMessage` tuto informaci vrací v podobě řetězce s chybovým hlášením.

Hierarchii výjimek rodiny `ParseException` ilustruje diagram 4.3.

Obrázek 4.3: Diagram tříd rodiny ParseException



4.3.6.1 Výjimka IncompleteFormulaException

Výjimka tohoto typu je vržena v případě, že vstupní formule je nekompletní.

4.3.6.2 Výjimka DetailedParseException

Tato obecná výjimka představuje syntaktickou chybu speficikovanou znakem vstupu, který chybu způsobil, a pozici tohoto znaku na řádce (kromě bílých znaků). Metoda `getMessage` je touto třídou překryta a je implementována tak, aby chybové hlášení obsahovalo tyto podrobnosti.

4.3.6.3 Výjimka UnexpectedEOFException

Výjimka tohoto typu je vržena v případě, že parser narazil na konec vstupu v místě, které nestanovuje korektní forma vstupu.

4.3.6.4 Výjimka IllegalCharacterException

Výjimka tohoto typu je vržena v případě, že vstup obsahuje nepovolený znak.

4.3.6.5 Výjimka RedundantElementException

Výjimka tohoto typu je vržena v případě, že řádek vstupu obsahuje korektní zápis formule a alespoň jeden nadbytečný prvek.

4.3.6.6 Výjimka UnexpectedElementException

Výjimka tohoto typu je vržena v případě, že infixní syntaktický zápis formule je nekorektní.

4.3.7 Třída ProofSystem

Ačkoliv tato práce implementuje Hilbertův systém jako jediný konkrétní důkazový systém, z důvodu rozšiřitelnosti jsme zavedli tuto abstraktní třídu, jejíž návrh odpovídá definici důkazového systému 1.4. Objekt této třídy obsahuje axiomy důkazového systému a implementuje metodu `isAxiom`, prostřednictvím které lze ověřit, zdali je daná formule instancí axiomu tohoto konkrétního důkazového systému. Metoda `isDeducible` slouží k odvození dané formule v rámci daného důkazu a vrací posloupnost indexů vstupních formulí, které jsou svědky odvozované formule za použití odvozovacího pravidla důkazového systému. Protože každý konkrétní důkazový systém odvozuje výrokové formule jiným způsobem, je tato metoda abstraktní.

Hierarchii tříd rodiny `ProofSystem` ilustruje diagram 4.4.

4.3.7.1 Třída HilbertSystem

Tato třída představuje Hilbertův systém. Je zřejmé, že je jedinou, která implementuje konkrétní důkazový systém. Pro účely odvozování formulí objekt této třídy obsahuje formuli představující implikaci pravidla modus ponens $(\varphi \Rightarrow \psi)$.

Implementovaná metoda `isDeducible` odpovídá odvozování formulí pravidlem modus ponens. Její implementace spočívá v procházení jednotlivých formulí daného důkazu, přičemž na jednu z formulí je vždy nahlíženo jako na předpoklad pravidla modus ponens (φ) a na druhou jako jeho implikaci $(\varphi \Rightarrow \psi)$. Daná formule ψ je odvoditelná, je-li nalezena dvojice vyhovující právě tomuto tvaru.

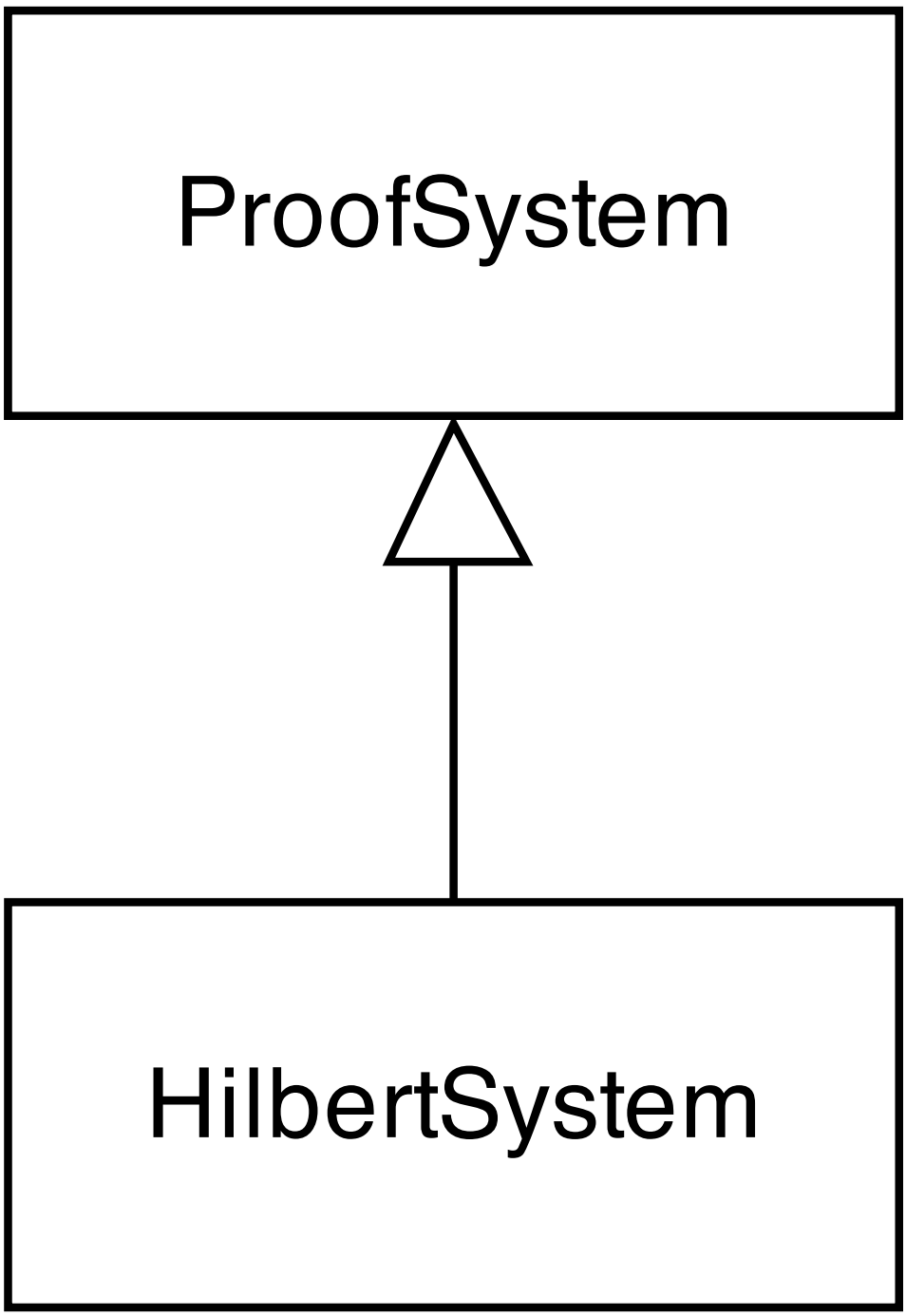
4.3.8 Třída ProofElement

Tato třída představuje člen důkazu jako komplexní strukturu. Objekt této třídy obsahuje výrokovou formuli, která je členem důkazu, a posloupnost členů téhož důkazu, které jsou svědky tohoto členu za užití odvozovacího pravidla. V poslední řadě je zde uložena logická proměnná vyjadřující, zdali je tento člen součástí optimální podoby důkazu, jehož je členem. Tato proměnná slouží pouze algoritmu optimalizace důkazu.

4.4 Manuálová stránka

Aby manuálová stránka byla dostupná příkazem `man pl`, musí splňovat jisté náležitosti. Název souboru manuálové stránky se skládá z názvu dokumentované aplikace, tečky a čísla příslušné manuálové sekce, v našem případě tedy `pl.1`. Tento soubor je zabalen v archivu `.gz` a nachází se v umístění, ve kterém příkaz `man` implicitně hledá manuálové stránky. Tato umístění jsou nastavena v konfiguračním souboru `/etc/manpath.config` a lze je případně

Obrázek 4.4: Diagram tříd rodiny ProofSystem



vypsat příkazem `manpath`. Archiv manuálové stránky je dále nutné umístit do podadresáře příslušícího dané manuálové sekci, v našem případě do adresáře `man1`. Absolutní cesta k naší manuálové stránce tedy může vypadat následovně: `/usr/local/man/man1/pl.1.gz`. Nakonec ještě souboru archivu nastavíme příslušné atributy dle vzoru systémových manuálových stránek, vlastník a skupina: root, práva: 0644.

Manuálovou stránku tvoří dvoupísmenná makra. Začíná-li makro na začátku řádku, předchází mu znak tečka. V textu manuálové stránky se nesmí vyskytovat prázdný řádek.

Korektní manuálová stránka začíná následujícími makry v dodrženém pořadí:

Dd Datum dokumentu ve formátu [měsíc den, rok].

Dt Titulek dokumentu (velká písmena).

Os Operační systém.

Naše implementace potom vypadá následovně:

```
\centering
.Dd May 10, 2014
.Dt PL 1
.Os Ubuntu Linux 14.04
```

Při tvorbě manuálové stránky dále využijeme makra `Sh` pro úvod nových sekcí, případně `Ss` pro úvod sekcí druhé úrovně.

4.4.1 Sekce NAME

V sekci `NAME` definujeme makrem `Nm` název aplikace a makrem `Nd` definujeme krátký popis účelu aplikace.

```
\centering
.Sh NAME
.Nm pl
.Nd handle formulas of propositional logic
```

4.4.2 Sekce SYNOPSIS

V sekci `SYNOPSIS` stanovíme makry `Op`, `Fl` a případně `Ar` výčet podporovaných přepínačů.

```
\centering
.Sh SYNOPSIS
.Nm
```

```
.Op Fl [přepínač] Ar [argument]
.
.
.
```

4.4.3 Sekce DESCRIPTION

Sekci uvedeme podrobným popisem účelu aplikace. Osvětlíme zde vstupní jazyk aplikace a uvedeme tabulku reprezentace spojek pomocí makra pro tabulkové prostředí `Bl -column`. Makrem `Ta` pak oddělíme jednotlivé buňky řádku tabulky.

```
\centering
.Bl -column "Connective" "ASCII" "words" "TeX"
.It Em "Connective ASCII Words TeX"
.It Li negation Ta - Ta not Ta \eneg
.It Li conjunction Ta . Ta and Ta \ewedge
.It Li disjunction Ta + Ta or Ta \evee
.It Li implication Ta > Ta implies Ta \eRightarrow
.It Li biconditional Ta = Ta iff Ta \eLeftrightarrow
.El
```

Dále popíšeme formu vstupu a výstupu a nakonec vysvětlíme funkcionalitu jednotlivých přepínačů pomocí odrážkového seznamu `Bl -tag`.

```
\centering
.Ss Options
The options are as follows.
.Bl -tag -width Fl
.It Fl
A~Check whether each formula is a Hilbert axiom.
.
.
.
.El
```

4.4.4 Sekce EXIT STATUS

Tato sekce vysvětluje význam návratových hodnot aplikace v závislosti na použitém přepínači. Užijeme dvouúrovňového odrážkového seznamu.

```
\centering
.Sh EXIT STATUS
.Bl -tag -width Fl
.It 0
```

```
.Bl -item
.It
Valid formula syntax.
.
.
.
.El
.It 1
.Bl -tag
.It
Invalid formula syntax.
.
.
.
.El
.El
```

4.4.5 Sekce EXAMPLES

V této sekci uvedeme krátký výčet příkladů použití. K tomu použijeme odrážkový seznam `Bl -tag`.

```
\centering
.Sh EXAMPLES
.Bl -tag -width Fl
.It Outputs \(dq-+AB\(dq
echo \(dq-(A+B)\(dq | pl -e -o prefix
.It Outputs \(dqType 2 axiom.\(dq
echo \(dq((A>(B>C))>((A>B)>(A>C)))\(dq | pl -e -A
.El
```

Sekvence
`(dq` zastupuje znak `"`, který jinak má speciální význam.

4.4.6 Sekce HISTORY

Zde krátce uvedeme historický kontext této práce.

```
\centering
.Sh HISTORY
Written for academic purposes in 2014.
```

4.4.7 Sekce AUTHORS

Zde uvedeme jméno autora. Makro `Mt` má význam adresy elektronické pošty.

```
\centering
.Sh AUTHORS
Written by
.An Jan Švajcr Mt svajcjan@fit.cvut.cz
```

Kapitola 5

Testování

V této kapitole popíšeme metody testování, jak požaduje naše zadání. Testovat budeme zmíněné případy užití:

- Ověření syntaxe a převod notace).
- Rozpoznání axiomů.
- Ověření důkazů.
- Optimalizace důkazů.

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

Pozitivní test Test korektního přijetí korektního vstupu.

Negativní test Test korektního odmítnutí nekorektního vstupu.

V případech, kdy na různé vstupy očekáváme reakci aplikace různými výstupy, zakládáme testování na principu konfrontace výstupu aplikace s předem připravenými testovacími daty, která považujeme za korektní. V případech, kdy na různé vstupy očekáváme stejnou reakci aplikace, nám stačí znát úspěšnost provedení v podobě návratové hodnoty.

Testování implementujeme pomocí skriptu pro `shell` za využití textových souborů s testovacími daty, které jsou umístěny v adresáři `test`. Zde jsou také uloženy soubory s očekávanými Spuštění tohoto testovacího skriptu volá cíl `make test`. Soubory vyprodukované testy končí v názvu `_test` a jsou směrovány do adresáře `out`.

5.1 Převod notace

Protože parsery výrokových formulí v naší implementaci reprezentují tři nezávislé funkce (`parsePrefix`, `parseInfix` a `parsePostfix`), je nezbytné provést testování vstupu ve všech třech možných notacích.

5.1.1 Pozitivní test

Všechny případy užití zahrnují užití parseru výrokových formulí. Proto nejdříve ze všeho ověříme, zdali parser správně transformuje korektní textový vstup do stanovené vnitřní reprezentace. To nezjistíme jinak než porovnáním vstupu s výstupem téže notace. Navíc tímto testem zároveň otestujeme korektnost vzájemného převodu prefixní, infixní a postfixní notace.

Připravíme dostatečný počet různých výrokových formulí tak, abychom v nich pokryli výskyty následujících případů:

- Kořenem stromu je:
  - Elementární výrok
  - Unární operátor
  - Binární operátor
- Potomek uzlu je:
  - Elementární výrok
  - Unární operátor
  - Binární operátor

My jsme vybrali následující formule:
_valid.txt

5.1.1.1 Implementace

Tyto formule jsou korektně zapsány v každé ze tří možných notací v testovacích souborech `prefix_valid.txt`, `infix_valid.txt` a `postfix_valid.txt`.

Testovací soubory postupně předáme na vstupu aplikaci `pl` a provedeme konverzi obsažených formulí do každé notace. Tím dosáhneme nejen porovnání stejných notací před a po zpracování aplikací, ale zároveň i porovnání notací různých. Výstup aplikace přesměrujeme do příslušných souborů a nástrojem `diff` nakonec ověříme, zdali se tyto soubory shodují s testovacími.

Tabulka 5.1: Nekorektní testovací formule

Případ	Prefix	Infix	Postfix
1	AB	$\neg$	$\neg$
2		$\wedge$	
3	$A\wedge$	$A\neg$	$A\wedge$
4	?	?	?
5	A	A	A

5.1.2 Negativní test

Základem negativního testování je analyzovat všechny možné podoby nekorektního vstupu. K této analýze nám pomůže výčet námi implementovaných druhů výjimek vyvolávaných parserem. Připravíme dostatečný počet různých nekorektních výrokových formulí tak, abychom pokryli následující případy:

1. Formule je nekompletní.
2. Formule má chybnou syntaxi (platí pouze pro infixní notaci).
3. Formule je kompletní přičemž následují nadbytečné prvky.
4. Formule obsahuje neplatný znak.
5. Formule není zakončená řádkovým zalomením.

5.1.2.1 Implementace

Každou z těchto nekorektních formulí zapíšeme v každé notaci do příslušných vzorových souborů `prefix_error.txt`, `infix_error.txt` a `postfix_error.txt`. Vzorové formule znázorňuje tabulka 5.1. Dále pro každou z těchto formuli zapíšeme odpovídající chybové hlášení do vzorových souborů `prefix_messages.txt`, `infix_messages.txt` a `postfix_messages.txt`.

Soubory `prefix_error.txt`, `infix_error.txt` a `postfix_error.txt` předáme jako vstup aplikaci `pl -e`. Výstup přesměrujeme do souborů `prefix_error_test.txt`, `infix_error_test.txt` a `postfix_error_test.txt`. Nástrojem `diff` nakonec porovnáme, zdali se vytvořené soubory shodují se vzorovými soubory s hlášeními.

Implementace testů vypadá následovně:

```
done

# Negative test
for IN in prefix infix postfix
do
```

```
$PL_CMD -i $IN -f $IN"_invalid.txt" >
    $OUT_PATH"parser_"$IN"_neg_test.txt" 2>&1
if ! diff $IN"_invalid_msg.txt"
    $OUT_PATH"parser_"$IN"_neg_test.txt" >
    "/dev/null" 2>&1;
then
        echo "Parser: Negative test failed!
            ($IN syntax)"
```

5.2 Rozpoznání axiomu

Protože naše práce implementuje Hilbertův systém, testovat budeme správné rozpoznání jeho axiomů.

5.2.1 Pozitivní test

Axiomy Hilbertova systému jsou korektně zapsány ve vzorovém souboru `axiom_valid.txt`.

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

Soubor `axiom.txt` předáme jako vstup aplikaci `pl -A` a zkontrolujeme, zdali její návratová hodnota značí úspěch.

Implementace testu vypadá následovně:

```
# Positive test
$PL_CMD -A -f "axiom_valid.txt" >
    $OUT_PATH"axiom_checker_pos_test.txt" 2>&1
if ! diff "axiom_valid_msg.txt"
    $OUT_PATH"axiom_checker_pos_test.txt" > "/dev/null"
    2>&1;
then
        echo "Axiom checker: Positive test failed!"
```

5.2.2 Negativní test

Formule představující Hilbertovské axiomy nepatrně pozměníme tak, aby tyto formule již axiomy nebyly, a korektně je zapíšeme do souboru `axiom_error.txt`.

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

Dále pro každou z těchto formulí zapíšeme do vzorového souboru `axiom_error_messages.txt` odpovídající chybové hlášení.

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

Soubor `axiom_error.txt` předáme jako vstup aplikaci `pl -A -e`. Výstup přesměrujeme do souboru `axiom_error_test.txt`. Nástrojem `diff` nakonec porovnáme, zdali se vytvořený soubor shoduje se vzorovým souborem s hlášeními.

Implementace testu vypadá následovně:

```
# Positive test
$PL_CMD -A -f "axiom_valid.txt" >
    $OUT_PATH"axiom_checker_pos_test.txt" 2>&1
if ! diff "axiom_valid_msg.txt"
    $OUT_PATH"axiom_checker_pos_test.txt" > "/dev/null"
    2>&1;
then
        echo "Axiom checker: Positive test failed!"
```

5.3 Ověření důkazu

Pro testování validátoru důkazů využijeme zmíněný důkaz formule $A \Rightarrow A$**??**.

5.3.1 Pozitivní test

Důkaz formule $A \Rightarrow A$ korektně je korektně zapsán ve vzorovém souboru `proof_valid.txt` následovně:

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

Soubor `proof.txt` předáme jako vstup aplikaci `pl -P` a zkontrolujeme, zdali její návratová hodnota značí úspěch.

Implementace testu vypadá následovně:

```
### Proof checker

# Positive test
$PL_CMD -P -f "proof_valid.txt" >
    $OUT_PATH"proof_checker_pos_test.txt" 2>&1
if ! diff "proof_valid_msg.txt"
    $OUT_PATH"proof_checker_pos_test.txt" > "/dev/null"
    2>&1;
then
```

5.3.2 Negativní test

Poslední formuli důkazu $A \Rightarrow A$ pozměníme tak, aby se důkaz stal nekorektním, a takto upravený důkaz zapíšeme v jazyce ASCII do souboru `proof_error.txt` následovně:

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

Soubor `proof_error.txt` předáme jako vstup aplikaci `pl -P` a zkontrolujeme, zdali její návratová hodnota značí neúspěch.

Implementace testu vypadá následovně:

```
# Negative test
$PL_CMD -P -f "proof_invalid.txt" >
    $OUT_PATH"proof_checker_neg_test.txt" 2>&1
if [ $? -eq 0 ];
then
        echo "Proof checker: Negative test failed!"
```

5.4 Optimalizace důkazu

5.4.1 Pozitivní test

5.4.2 Negativní test

5.5 Další testy

5.5.1 Test manuálové stránky

groff -z -mdoc...

5.5.2 Test přenositelnosti

Na kterých systémech jsme testovali...

5.5.3 Test uvolnění paměti

Ruční kontrola uvolněných paměťových bloků Valgrindem...

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 funkcionalita nalezení důkazu dané (dokazatelné) formule.

Pro nalezení důkazu bychom využili rozhodnutelnosti logiky, tedy skutečnosti, že dokazatelné jsou právě tautologie[2], a ty se dají efektivně poznat. Je 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).

Při použití přepínače `-F` jsou vstupní formule vnímány jako formule, ke kterým chceme nalézt důkaz. Aplikace se jej pokusí nalézt a případně jej vypíše na výstup. Úspěch, resp. neúspěch nalezení důkazu aplikace standardně indikuje návratovou hodnotou 0, resp. 1.

6.2 Gentzenův systém

Hilbertův systém není jediný možný důkazový systém pro výrokovou logiku. Pravidlo modus ponens však zřejmě nejlépe odpovídá postupům našeho uvažo-

[2]Tautologie – složený výrok, který je bez ohledu na dílčí pravdivostní ohodnocení vždy pravdivý.

vání [1]. Ovšem, existují i jiná formální odvozovací pravidla, například *pravidlo rezoluce* Gentzenova systému.

6.2.1 Rezoluční metoda

Rezoluční metoda je metodou hledání sporu v dané množině formulí. Vyžaduje formule v konjunktivně-disjunktním tvaru (disjunkce konjunkcí). Připomeneme, že takový jazyk je ekvivalentní jazyku, který využívá Hilbertův systém. Pravidlo rezoluce je zavedeno následovně:

Definice 5. Z formulí φ, ψ, α odvoď formuli $\varphi \vee \psi$, platí-li $(A \vee B) \wedge (\neg A \vee C)$.

Takto odvozenou formuli pak nazýváme *rezolventou*.

Stávající implementaci bychom tedy mohli rozšířit právě o tento důkazový systém. Aplikace by pak mohla ověřovat důkazy v tomto systému nebo dokonce důkazy mezi těmito systémy převádět. V logice jsou totiž tyto dva systémy stejně silné.

6.2.2 Návrh obsluhy

6.2.2.1 Režim důkazového systému

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

Přepínač `-s` s možnými hodnotami `hilbert` a `gentzen` je rozšířením funkcionality stávajícího přepínače `-P`. Určuje, v rámci kterého důkazového systému má aplikace operovat při práci s výrokovými důkazy.

6.2.2.2 Konvertor důkazů

Zavedeme nový přepínač `-C` (convert).

Přepínač `-C` s možnými hodnotami `hilbert` a `gentzen` je novým cílem aplikce `pl`. Při jeho použití aplikace převede důkaz na vstupu do daného důkazového systému. Protože tyto systémy jsou ekvivalentní, úspěch převodu závisí pouze na korektnosti vstupního důkazu. Aplikace jej standardně indikuje návratovou hodnotou.

6.3 Paralelní práce

Zadání této bakalářské práce vychází z velkého množství požadavků na tutéž aplikaci. Její funkcionalita se měla dotýkat oblasti sémantiky výrokové logiky, kterou jsme se v této práci nezabývali. Z těchto požadavků byly sestaveny tři zadání bakalářských prací, které byly vypracovávány paralelně. Možným navázáním na tuto práci, stejně jako na ty dvě další, se proto nabízí sjednocení funkcionality těchto tří aplikaci do jediné tak, jak bylo zamýšleno 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 dále 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] *GNU Make Manual*. [Citováno 2014-05-10]. Dostupné z: `https://www.gnu.org/software/make/manual/`

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

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

[6] 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

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