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.

13. května 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 13. května 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 jsou náležitosti jako 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 a creation of a terminal application implementing the proof system environment of the propositional logic. It's main functionality is parsing the textual input represented as a sequence of propositional formulas and validation of this sequence as a proof. The software is implemented in the C++ language and is supported by UNIX environment. This thesis also requires a code documentation and a standard manual page as a user manual.

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 vzniku 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. Již z názvu této práce vyplývá, že implementuje principy důkazového systému, konkrétně Hilbertova systému. Ústřední kapitolou výrokového počtu, o kterou se budeme v rámci teoretické přípravy zejména zajímat, je tedy jistě dokazatelnost.

Ještě předtím než se ponoříme do problematiky implementace, je třeba analyzovat vlastnosti příslušné oblasti výrokového počtu. V první kapitole si proto vysvětlíme klíčové pojmy výrokové logiky, které nás budou provázet životním cyklem projektu a budou tak pro nás nutnou znalostí. Pokračovat budeme kapitolou, kterou zahájíme projekt vymezením požadavků na implementaci. Zde si zcela upřesníme zadání projektu na základě zadání původního. Následuje kapitola, ve které položíme základy pro vlastní implementaci. Budeme se zde věnovat rozkladu problémů na menší celky a návrhem jejich řešení. K programátorské realizaci projektu přejdeme v navazující kapitole zabývající se detaily samotné implementace. Nastíníme zde zdrojový kód datových struktur a klíčových algoritmů. Už během implementace nás zabývá správná funkčnost produktu. Nelze proto vynechat kapitolu, která se věnuje testování. Budeme se zde snažit pokrýt všechny rizikové oblasti, které mohou ohrozit stabilitu aplikace. Práci zakončíme krátkým zamyšlením nad jejím možným pokračováním. Pro potenciální zájemce o navázání zde proto prodiskutujeme její rozšířitelné stránky.

Kapitola 1

Formální kontext

V úvodní kapitole se seznámíme s několika základními pojmy výrokového počtu, aby každý čtenář byl srozuměn s terminologií užitou v tomto textu a mohl na ni být později kdykoli zpětně odkázán. Tato část slouží jako teoretický základ celé práce a je zřejmé, že čtenář znalý výrokové logiky ji vynechá.

1.1 Výrok

Nejprve zavedeme elementární pojem *výrok*.

Výrok je takové tvrzení, o kterém má dostatečný smysl uvažovat, zdali je pravdivé či nikoliv. Výrokům tedy přiřazujeme pravdivostní hodnotu pravda (true) nebo nepravda, (false) což nazýváme jejich *pravdivostním ohodnocením*. Jednotlivé elementární výroky značíme velkými písmeny latinky: $A, B, C, \ldots$

Příklad 1. Pro ilustraci *elementárního výroku* uvažujme následující výrok A: „Dnes je hezký den.“ Je na čtenáři, aby rozhodl o pravdivosti tohoto výroku a jistě bude souhlasit, že jiný čtenář by mohl rozhodnout například opačně.

V této práci se ale pravdivostním ohodnocením výroků stejně jako jejich významem nebudeme zabývat. Zmíněná fakta o výroku slouží pouze jako součást definice a čtenář ji nemusí považovat za klíčovou.

1.1.1 Složený výrok

Ze základních výroků lze tvořit výroky další, tzv. *složené výroky*, pomocí *logických operací*. V závislosti na pravdivostním ohodnocení elementárních výroků a na typu použité logické operace přisuzujeme pravdivostní ohodnocení výrokům složeným.

1.1.1.1 Logické operace

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

- Unární

  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 definovat, protože není pro účely této práce podstatný. Jako důležité vnímejme pouze rozlišení jednotlivých operací a jejich aritu.

Příklad 2. Pro ilustraci *složeného výroku* uvažujme následující výrok: $B = \neg A$. V souvislosti s předchozím příkladem elementárního výroku prohlásíme, že se jedná o výrok s opačnou pravdivostní hodnotou než má výrok A. Do přirozeného jazyka bychom jej tedy přeložili jako „Dnes není hezký den“ nebo „Není pravda, že dnes je hezký den.“. Druhá varianta překladu více koresponduje se syntaktickým vnímáním tohoto složeného výroku.

1.2 Formule

Ústředním pojmem pro tuto práci je *formule*.

Definice 1. V matematické logice obecně definujeme formuli tak, že:

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. Každou formuli dostaneme postupnou aplikací předchozích pravidel[1].

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

Příklad 3. Pro ilustraci formule uvažujme dvě následující formule α, β:

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

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

Je zřejmé, že formule α je elementární formulí, kdežto formule β je složená z několika formulí. Pro názornost popíšeme skladbu formule β na základě předchozí definice formule.

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

1.2.1 Syntaxe

Nyní se chvíli věnujme způsobu zápisu výrokových formulí. Tuto znalost budeme později potřebovat.

Výrokové formule lze zapisovat ve třech různých notacích: *prefixní*, *infixní* a *postfixní*. Tyto notace jsou navzájem sémanticky ekvivalentní, liší se pouze pořadím výpisu logické spojky složených výroků.

Příklad 4. Všechny tyto zápisy názorně předvedeme na konkrétním tvaru formule následovně:

prefix $\neg \vee \alpha\, \beta$

infix $\neg(\alpha \vee \beta)$

postfix $\alpha\, \beta \vee \neg$

Všimněme si, že v případě prefixní a postfixní notace je přednost operací určena jednoznačně na rozdíl od notace infixní, která striktně 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.

1.3 Důkazový systém

Důkazový systém je aparát výrokové logiky, který rozhoduje o dokazatelnosti libovolné výrokové formule. Každý takový systém je tvořen dvěma důležitými množinami:

- Množinou axiomů
- Množinou odvozovacích pravidel

Axiom je schema definující určitý tvar výrokových formulí. Můžeme si ho představit jako výrokovou formuli, za jejíž elementární členy lze dosazovat již konkrétní výrokové formule. Každá formule vyhovující tvaru popsaného axiomem nazýváme instancí tohoto axiomu. Instancí axiomů je tedy nekonečně mnoho stejně jako výrokových formulí.

Odvozovací pravidlo je předpis, který definuje způsob, jakým lze z výrokových formulí dokázat formule další.

Tyto náležitosti nyní objasníme na konkrétním důkazovém systému.

1.3.1 Hilbertův systém

Hilbertův axiomatický systém je formální důkazový systém, který se za účelem úspory tvaru formulí omezuje pouze na dvě logické spojky: negaci a implikaci. Protože tyto dvě spojky tvoří minimální universální množinu, touto redukcí neomezujeme vyjadřovací možnosti jazyka, tedy všechny ostatní spojky můžeme těmito případně adekvátně vyjádřit[2]. Jakým způsobem lze takto vzájemně spojky převádět je však záležitost, kterou také zkoumat nebudeme.

Množina axiomů obsahuje tyto prvky:

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, tzv. pravidlo *modus ponens*, které zavedeme následovně: φ lze odvodit, platí-li ψ a zároveň $\psi \Rightarrow \varphi$.

V této práci se nadále budeme zabývat pouze tímto axiomatickým systémem.

1.3.2 Důkaz

Definice 2. Buď φ výroková formule. Řekneme, že konečná posloupnost výrokových formulí $\varphi, \ldots, \varphi$ je *důkazem* formule φ ve výrokové logice, pokud φ_n je formule φ, a každá formule φ_i z této posloupnosti je buďto instancí některého axiomu, nebo je z některých předchozích φ_j, φ_k, kde $j, k < i$, odvozena pravidlem modus ponens. Pokud existuje důkaz formule φ, řekneme, že φ je dokazatelná ve výrokové logice, a píšeme $\vdash \varphi$[2].

Je zřejmé, že každý důkaz musí začínat axiomem, tedy 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]. Tato fakta však nenapomáhají důkaz dané formule nalézt a není to ani předmětem našeho zájmu, protože tato problematika již přesahuje rozsah této práce.

Příklad 5. Pro ilustraci důkazu předvedeme důkaz formule $\varphi \Rightarrow \varphi$. V kontextu této formule si připomeňme, že formalismus logiky nás oprošťuje od psychologického aspektu, tedy banalitu tohoto důkazu se snažíme nevnímat.

1. $(\varphi \Rightarrow ((\varphi \Rightarrow \varphi) \Rightarrow \varphi))$
2. $((\varphi \Rightarrow ((\varphi \Rightarrow \varphi) \Rightarrow \varphi)) \Rightarrow ((\varphi \Rightarrow (\varphi \Rightarrow \varphi)) \Rightarrow (\varphi \Rightarrow \varphi)))$
3. $((\varphi \Rightarrow (\varphi \Rightarrow \varphi)) \Rightarrow (\varphi \Rightarrow \varphi))$
4. $(\varphi \Rightarrow (\varphi \Rightarrow \varphi))$
5. $(\varphi \Rightarrow \varphi)$

1.3.2.1 Důkaz z předpokladů

Důkaz lze zobecnit na tzv. *důkaz z předpokladů* doplněním stávající definice. Navíc zavedeme množinu formulí, ze kterých v rámci důkazu vycházíme, tzv. *předpoklady*. Toto zobecnění spočívá v tom, že kromě axiomů jako členy důkazu podobně připouštíme i prvky množiny předpokladů[1]. Předpoklad je tedy formule, kterou, ačkoliv není axiomem, v rámci důkazu považujeme za dokazatelnou.

Tento obecný typ důkazu tato práce neimplementuje. Zmínka o něm je zde pouze pro účely pozdější diskuze nad rozšířitelností této práce.

1.3.2.2 Optimalizace důkazů

V rámci pozdějšího návrhu software je nyní nutné zamyslet se, jakým způsobem lze důkazy v Hilbertově systému optimalizovat.

Lemma 1. Buď $\varphi_1, \ldots, \varphi_n$ posloupnost formulí tvořící Hilbertovský důkaz. Pak každá vybraná posloupnost, která vznikne z $\varphi_1, \ldots, \varphi_n$ vynecháním druhého a každého dalšího výskytu nějaké zvolené formule φ_i, je opět důkazem. Tedy i po vynechání všech duplicit zůstává daná posloupnost důkazem.

Důkaz tohoto tvrzení je netriviální pouze v případě použití modus ponens ve výrokovém důkazu. Je totiž zřejmé, že důkaz zůstane stále důkazem, vynecháme-li všechny duplicitní výskyty toho kterého axiomu.

Důkaz 1. Pokud nějaká formule φ_i vyplývá z nějakých předchozích formulí φ_j a $\varphi_j \to \varphi_i$ pomocí modus ponens, pak stejným způsobem vyplývá i z každých jejich předchozích (speciálně z prvních) výskytů.

Tento obrat můžeme považovat za zárodek optimalizace důkazů.

Kapitola 2

Vymezení požadavků

Na základě oficiálního zadání této práce, kterým začíná tento text, vymezíme v této kapitole požadavky na implementovaný software. Tyto požadavky nám později pomohou analyzovat všechny potřebné součásti systému. 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^2, která dokáže rozhodnout, zdali je daná posloupnost výrokových formulí korektním formálním důkazem Hilbertova axiomatického systému, popřípadě dokáže tento důkaz optimalizovat. Elementární funkcionalitou programu je syntaktická analýza vstupních formulí a jejich případný výpis ve zvoleném alternativním jazyce a syntaxi. Nekorektní vstup program správně detekuje a případně jeho nekorektnost podrobně hlásí. Zdrojový kód programu je třeba náležitě zdokumentovat. K aplikaci je třeba vytvořit uživatelskou příručku.

[2] pl – propositional logic

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. Veškerá funkcionalita aplikace bude dostupná standardně pomocí přepínačů. Vstupní data budou aplikaci předávána buď pomocí standardního vstupu, anebo textovým souborem. Návratová hodnota programu bude signalizovat úspěšnost výsledku.

2.2.2 Vstup

Program čte 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 jeho použití způsobí neúspěšné vykonání aplikace.

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, anglických slov nebo jazyka LaTeX. Výstupem programu může být v určitých případech chybové hlášení.

2.2.4 Dokumentace

Programátorská dokumentace bude dostupná v podobě HTML stránek za použití nástroje *Doxygen*, který takovou dokumentaci generuje. Zdrojový kód programu proto zdokumentujeme 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 přístupná standardně přes příkaz `man pl` po instalaci aplikace do systému.

Kapitola 3

Analýza a návrh

Před samotným návrhem je potřeba vybrat si platformu systému UNIX. My zvolíme platformu Linux, konkrétně distribuci *Ubuntu 14.04 Trusty Tahr*.

3.1 Aplikace

3.1.1 Sestavení

K sestavení aplikace využijeme nástroj *GNU make*[3]. Proces sestavení lze nyní řídit konfiguračním souborem *Makefile*. Tento soubor má předepsanou strukturu a obsahuje definice pravidel, tzv. *cílů*. Jednotlivé cíle mají svůj význam a jsou definovány jménem, závislostmi na ostatních cílech a posloupností příkazů, které mají být provedeny při volání cíle. Cíle rozdělíme 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 kroky. Protože nástroj `make` má programátorovi proces sestavení aplikace usnadnit, nakonfigurujeme proces kompilace tak, abychom nemuseli zasahovat do struktury souboru `Makefile` po každé provedené změně zdrojových souborů, například po přejmenování souboru nebo změně závislostí na hlavičkových souborech. Příkaz `make` tedy bude aplikaci sestavovat dynamicky podle aktuální podoby zdrojové formy.

Náš `Makefile` bude disponovat těmito hlavními cíly:

build Kompilace zdrojových souborů a sestavení aplikace.

clean Vyčištění adresáře odstraněním všech výstupů.

doc Generování Doxygen dokumentace.

test Vyvolání aplikačních testů.

install Instalace aplikace do systému.

uninstall Odinstalace aplikace ze systému.

Tabulka 3.1: Výstupní jazyky spojek

ASCII	Slovní	LaTeX
`-`	`not`	`\neg`
`.`	`and`	`\wedge`
`+`	`or`	`\vee`
`>`	`implies`	`\Rightarrow`
`=`	`iff`	`\Leftrightarrow`

3.1.2 Vstup

Textový vstup předaný aplikaci `pl` má jednoduchý formát. Skládá se z posloupnosti formulí z nichž každá (včetně poslední) je zakončená řádkovým zalomením, znakem
`n`. Zápis formulí užívá znaků `A-Z`[3] pro elementární výroky, `-.+>=` pro logické operátory (negace, konjunkce, disjunkce, implikace, ekvivalence), případně symbolů `()` pro infixní syntaxi.

3.1.3 Výstup

Kromě výstupní syntaxe je možné zvolit i jeden ze tří výstupních jazyků. Jak jsou jednotlivé operátory zastoupeny v těchto jazycích znázorňuje tabulka 3.1.

3.1.4 Přepínače

Přepínače, které aplikace `pl` podporuje mají následující význam:

A Ověř, zdali formule jsou Hilbertovské axiomy.

D Optimalizuj Hilbertovský důkaz.

e Povol hlášení na výstup.

f Čti z daného souboru.

i Očekávej danou vstupní syntaxi.

l Použij daný výstupní jazyk.

o Použij danou výstupní syntaxi.

P Ověř, zdali posloupnost formulí je korektní Hilbertovský důkaz.

s Ukonči aplikaci při první nalezené chybě.

[3] Vstup je tímto omezen na 26 různých výroků, což pro naše účely plně postačuje.

Tímto výčtem jsme stanovili veškerou funkcionalitu aplikace. Ještě dodáme, že aplikace implicitně neprodukuje žádný výstup, čte ze standardního vstupu, očekává infixní vstupní syntaxi, pro výstup používá ASCII jazyk a infixní syntaxi a neukončuje se při první nalezené chybě.

3.2 Manuálová stránka

Manuálové stránky systému UNIX lze psát ve speciálním značkovacím jazyce. Takový jazyk je ve skutečnosti balíček maker pro jazyk *troff*[4]. Troff pochází ze 70. let a podobně jako například LaTeXslouží k sazbě textu. Zpracovává jej textový procesor `troff`, který podporuje právě zmíněné balíčky maker pro tento jazyk. Jednotlivá makra podobně jako například HTML formátují daný obsah do požadované podoby. Běžné manuálové stránky užívají balíček maker `man`. My však zvolíme sofistikovanější balíček `mdoc`[5], který narozdíl od `man` disponuje sémanticky koncipovanými makry. Záznam o tomto balíčku nalezneme v sekci *MDOC(7)*.

Aby manuálová stránka byla dostupná příkazem `man pl`, musí splňovat určité náležitosti. Název souboru manuálové stránky se skládá z názvu dokumentované aplikace, tečky a čísla příslušné manuálové sekce, v našem případě tedy `pl.1`. Textový soubor manuálové stránky 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 vypsat příkazem `manpath`. Archiv manuálové stránky je ještě 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.

3.2.1 Návrh

Manuálové stránky mají standardní strukturu, kterou uživatel očekává a my se ji vynasnažíme dodržet. Text příručky napíšeme v angličtině a rozdělíme jej 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

Při implementaci se držíme několika stanovených konvencí, které nám pomáhají udržet zdrojový kód konzistentním. Jednou ze zásad vyplývající ze zadání je komentování zdrojového kódu komentáři stylu Doxygen. Tyto speciální komentáře zapisujeme výhradně do hlavičkových souborů.

4.1 Aplikace

4.1.1 Sestavení

Podpůrné cíle odlišíme od hlavních tečkou, kterou název podpůrného cíle bude začínat:

.folders Vytvoření výstupních adresářů pro proces sestavení.

.install Umístění binární formy aplikace a její manuálové stránky do příslušných systémových adresářů.

.test Spuštění aplikačních testů.

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 Implementace

4.2 Manuálová stránka

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.

4.2.1 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
```

4.2.2 Sekce SYNOPSIS

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

```
\centering
.Sh SYNOPSIS
.Nm
.Op Fl [přepínač] Ar [argument]
.
.
.
```

4.2.3 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
```

4.2.4 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
```

4.2.5 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.

4.2.6 Sekce HISTORY

Zde krátce uvedeme historický kontext této práce.

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

4.2.7 Sekce AUTHORS

Zde uvedeme jméno autora. Použité 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 objasníme metody testování, které použijeme k ověření správného fungování aplikace. Testovat budeme konkrétně tyto oblasti:

- Syntaktickou analýzu formulí.
- Rozpoznání Hilbertovských axiomů.
- Ověření Hilbertovských důkazů.

Základní myšlenkou pro maximalizaci pokrytí případů testy je v každé oblasti provést dvě varianty testů:

Pozitivní test Test korektního přijetí korektního vstupu.

Negativní test Test korektního nepřijetí nekorektního vstupu.

Testování implementujeme pomocí skriptu pro `shell` za využití souborů s testovacími daty, které umístíme do adresáře `test`. Volání tohoto testovacího skriptu volá cíl `make test`.

Inicializace testovacího skriptu zahrnuje inicializaci proměnné `TEST_SUCCESS`, která značí úspěch celého testování, a výpis hlášení o zahájení testů.

```
#!/bin/sh

TEST_SUCCESS=1

cd test
echo "Performing tests\n-------------------"
```

V závěru testovacího skriptu zkontrolujeme hodnotu proměnné `TEST_SUCCESS` a vypíšeme příslušné hlášení o úspěchu testování.

Tabulka 5.1: Korektní testovací formule

Prefix	Infix	Postfix
A	A	A
$\neg A$	$\neg A$	$A\neg$
$\wedge AA$	$(A \wedge A)$	$AA\wedge$
$\neg \wedge \neg A\neg A$	$\neg(\neg A \wedge \neg A)$	$A\neg A\neg \wedge \neg$
$\wedge A \wedge \wedge AAA$	$(A \wedge ((A \wedge A) \wedge A))$	$AAA \wedge A \wedge \wedge$
$\neg \wedge \neg A\neg \wedge \neg \wedge \neg A\neg A\neg A$	$\neg(\neg A \wedge \neg(\neg(\neg A \wedge \neg A) \wedge \neg A))$	$A\neg A\neg A\neg \wedge \neg A\neg \wedge \neg \wedge \neg$

```
if [ $TEST_SUCCESS -eq 1 ];
then
        echo "Success!"
else
        echo "--------------------\nFailure!"
fi
```

5.1 Parser

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

5.1.1 Pozitivní test

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

Každou z těchto formulí korektně zapíšeme v každé notaci do příslušných vzorových souborů `prefix.txt`, `infix.txt` a `postfix.txt`. Vzorové formule znázorňuje tabulka 5.1.

Soubory `prefix.txt`, `infix.txt` a `postfix.txt` předáme jako vstup aplikaci `pl -e` tak, abychom provedli jejich konverzi do každé ze tří notací. Výstup přesměrujeme do souborů `prefix_test.txt`, `infix_test.txt` a `postfix_test.txt`. Nástrojem `diff` nakonec porovnáme, zdali se vytvořené soubory shodují se vzorovými.

Implementace testů vypadá následovně:

```
for in in prefix infix postfix
do
        for out in prefix infix postfix
        do
                ../out/pl -e -i $in -o $out -f $in.txt
                   > ../out/$in\_$out\_test.txt 2>&1
                if ! diff -q $out.txt
                    ../out/$in\_$out\_test.txt >
                    /dev/null;
                then
                        echo "Test ' $in ----> $out ' 
                            failed!"
                        TEST_SUCCESS=0
                fi
        done
done
```

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 parsery. Připravíme dostatečný počet různých nekorektních výrokových formulí tak, abychom pokryli následující případy:

1. Formule je nekompletní.
2. Formule má nekorektní zápis (pouze infixní parser).
3. Formule je kompletní přičemž pokračuje nadbytečnými prvky.
4. Formule obsahuje neplatný znak.
5. Formule není zakončená řádkovým zalomením.

Každou z těchto nekorektních formulí zapíšeme v každé notaci do příslušných vzorových souborů `prefix_error.txt`, `infix_error.txt` a `postfix_error.txt`.

Tabulka 5.2: 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

Vzorové formule znázorňuje tabulka 5.2. 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:

```
for in in prefix infix postfix
do
        ../out/pl -e -i $in -f $in\_error.txt >
            ../out/$in\_error\_test.txt 2>&1
        if ! diff -q $in\_messages.txt
            ../out/$in\_error\_test.txt > /dev/null;
        then
                echo "Invalid $in syntax test failed!"
                TEST_SUCCESS=0
        fi
done
```

5.2 Ověření axiomů

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

5.2.1 Pozitivní test

Axiomy Hilbertova systému korektně zapíšeme do vzorového souboru `axiom.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:

```
../out/pl -A -f axiom.txt
if [ $? -ne 0 ];
then
        echo "Positive axioms testing failed!"
        TEST_SUCCESS=0
fi
```

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:

```
../out/pl -A -f axiom.txt
if [ $? -ne 0 ];
then
        echo "Positive axioms testing failed!"
        TEST_SUCCESS=0
fi
```

5.3 Validátor důkazů

Pro testování validátoru důkazů využijeme důkaz formule $A \Rightarrow A$ ze strany 7.

5.3.1 Pozitivní test

Důkaz formule $A \Rightarrow A$ korektně zapíšeme do vzorového souboru `proof.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:

```
../out/pl -P -f proof.txt
if [ $? -ne 0 ];
then
        echo "Test 'valid proof' failed!"
        TEST_SUCCESS=0
fi
```

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:

```
../out/pl -P -f proof_error.txt
if [ $? -eq 0 ];
then
        echo "Test 'invalid proof' failed!"
        TEST_SUCCESS=0
fi
```

Kapitola 6

Rozšířitelnost

Následující funkcionalita naší implementaci chybí a bylo by dobré ji v budoucnu implementovat.

6.1 Důkaz z předpokladů

V úvodní kapitole jsme v rámci popisu důkazových systémů definovali důkaz. Na straně 7 jsme se také zmínili o důkazu z předpokladů, který však tato práce neimplementuje. Implementaci bychom mohli rozšířit právě o tuto obecnou variantu důkazu. Následující návrh řešení snad ukazuje, že přidání této funkcionality lze pokládat za minimálně pracné.

6.1.1 Návrh

Přepínač `-P` rozšíříme o číselnou hodnotu n vyjadřující počet prvků množiny předpokladů. Aplikace tímto považuje prvních n výrokových formulí za předpoklady, ukládá je do zvláštního seznamu a při validaci výrokového důkazu kontroluje, zdali se mezi prvky důkazu nenacházejí právě tyto předpoklady.

6.2 Gentzenův systém

Hilbertův systém není jediný důkazový systém výrokové logiky, jeho pravidlo *modus ponens* je však považováno za nejvíce odpovídající našemu uvažování. Naší implementaci bychom mohli doplnit o důkazový systém Gentzenův. Aplikace by tak mohla ověřovat důkazy v tomto systému nebo dokonce důkazy mezi těmito systémy převádět.

6.2.1 Návrh

Zavedeme nové přepínače: `-s` a `-C`.

Přepínač `-s` s možnými hodnotami `hilbert` a `gentzen` určuje, v rámci kterého důkazového systému má aplikace operovat při kontrole výrokového důkazu, je-li zároveň použit přepínač `-P`, nebo do kterého z těchto systémů má aplikace důkaz převést, je-li zároveň použit přepínač `-C`. Úspěch převodu závisí pouze na korektnosti vstupního důkazu a aplikace jej indikuje návratovou hodnotou.

6.3 Nalezení důkazu

Tato práce nás naučila ověřovat důkazy výrokových formulí na základě vlastností důkazových systémů, ale nezabývali jsme se v ní, jakým způsobem důkazy hledat. Naší implementaci by velice obohatila funkcionalita nalezení důkazu k dané formuli, pokud takový důkaz existuje.

6.3.1 Návrh

Zavedeme nový přepínač `-F`.

Při použití přepínače `-F` je vstupní formule vnímána jako formule, ke které chceme nalézt Hilbertovský důkaz. Aplikace se jej pokusí nalézt a existuje-li, vypíše jej na výstup, pokud je zároveň použit přepínač `-e`. Úspěch nalezení důkazu aplikace indikuje návratovou hodnotou.

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++. Své dosavadní znalosti jsem také rozšířil. Nových zkušeností jsem konkrétně nabyl v oblasti výrokové logiky, jazyka C++, nástroje `make`, jazyka LaTeXa tvorby manuálové stránky pro UNIX. V poslední řadě jsem si poprvé zkusil takto rozsáhlou literární činnost. Jsem rád, že mě práce na tomto projektu těšila.

Literatura

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

[2] Švejdar, V.: *Logika: neúplnost, složitost a nutnost*. Praha: Academia, 2002, ISBN 80-200-1005-X.

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

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

[5] 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

HTML HyperText Markup Language

Příloha **B**

Obsah přiloženého CD

- `exe` ..... adresář se zkompilovanou formou implementace
- `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 textem práce