CHALMERS

$$\dfrac{\dfrac{p \to (q \to r) \quad \dfrac{[p \wedge q]}{p}\,\wedge_{e1}}{q \to r}\,\to_e \qquad \dfrac{[p \wedge q]}{q}\,\wedge_{e2}}{\dfrac{r}{p \wedge q \to r}\,\to_i}\,\to_e$$

Automated Reasoning

Automated theorem proving in Haskell

Kandidatarbete
Civilingenjörsprogrammet för datateknik

RICHARD FREDRIKSSON
DANIEL GUSTAFSSON
CHRISTOPHER ISSAL
MARTIN JOHANSSON
RASMUS JOHANSSON
ANDERS MÖRTBERG

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg 2008

Automated Reasoning
Automated theorem proving in Haskell

Richard Fredriksson, Daniel Gustafsson, Christopher Issal,
Martin Johansson, Rasmus Johansson, Anders Mörtberg

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
412 96 Göteborg

Omslag:
På omslaget presenteras ett bevis i naturlig deduktion för att $p \to (q \to r) \vdash p \wedge q \to r$.
Göteborg maj 2008

Abstract

Automated reasoning is a prerequisite for computers to be able to deduce logical conclusions, i.e. to make decisions on their own. It can for example be used to prove mathematical theorems. This report presents implementations of three automated theorem provers capable of proving theorems in first-order logic. They are based on resolution, narrowing and method of analytic tableau and are compared with respect to performance, readability of generated proofs and difficulty of implementation.

Sammanfattning

Automatiserat resonerande är en förutsättning för att datorer ska kunna härleda logiska slutledningar, d.v.s. att kunna ta egna beslut. Det kan också användas för att exempelvis bevisa logiska teorem. Denna rapport kommer att presentera implementationer av tre automatiska teorembevisare som klarar av att bevisa teorem i predikatlogik. De är baserade på resolution, narrowing och semantisk tableau och jämförs med avseende på prestanda, läsbarhet av genererade bevis och svårighetsgrad vid implementering.

Tack till

Vi vill tacka vår handledare Fredrik Lindblad för hans kontinuerliga stöd och idéer under arbetets gång. Vi vill även tacka Louise Holmer som har kommit med mycket konstruktiv kritik på rapporten och till sist ett tack till Anjelica Hammersjö, som spenderat sin fritid med korrekturläsning.

Innehåll

Kapitel 1

Inledning

1.1 Bakgrund

Detta avsnitt kommer att motivera varför automatiserad teorembevisning är av intresse att studera och vilka fördelar som finns i förhållande till mänsklig teorembevisning. Logikens ursprung kommer att presenteras tillsammans med anledningen till att den är en förutsättning för att området automatiserad teorembevisning existerar.

1.1.1 Logik

Logik är nödvändig för all vetenskap. Utan möjligheten att dra logiska slutledningar är det inte möjligt att veta om ens slutsatser är giltiga eller ej. Logik används på flera olika nivåer inom olika vetenskapliga områden, den spelar en mycket central roll i så skilda områden som matematik, sociologi eller historia.

Ordet logik kommer från grekiskans *logikos* med betydelserna 'som hör till talet' eller 'förnuftig'. Enligt Platon uppstod logiken ur utbytandet av argument och motargument, s.k. *dialektik*[1], och studerades för kunna föra en övertygande dialog. Som vetenskap anses logiken ha grundats av Aristoteles, som med sin lära om syllogismer undersökte giltigheten hos slutledningar utifrån de fyra logiska formlerna: "alla A är B", "några A är B", "inga A är B" och "några A är inte B". Utvecklingen har sedan dess gått mycket långsamt och det är först under senare hälften av 1800-talet som det sker några större förändringar. Dessa kom genom George Booles införande av klasslogik och Gottlob Freges kvantifieringar och predikatlogik.

Logiken kan delas upp i formell och informell logik. Informell logik berör logik i argumentationer i naturligt språk, där Platons dialoger är typiska exempel. Formell logik är mer svårdefinierat. Vi väljer att se formell logik som symbolisk logik, det vill säga studien av symboliska abstraktioner som beskriver de formella elementen i logisk inferens.

Traditionellt har logiken ansetts tillhöra filosofin, men under början av 1900-talet började man studera det som en grund till matematiken. Ett exempel på detta är Hilberts program vars mål bland annat var att formalisera alla då existerande matematiska teorier till en ändlig, komplett mängd axiom och i detta system bevisa att man inte kan deducera någon motsägelse, det vill säga att systemet är konsistent.

Under senare delen av 1900-talet har logiken spridit sig till fler områden där den blivit en mycket värdefull tillgång. Ett exempel är automatiserat resonerande, som detta projekt handlar om, vilket är ett subfält till artificiell intelligens.

[1] Plocka isär el. samtalskonst.

1.1.2 Automatisk teorembevisning

Det finns många argument för att automatisera teorembevisning. Till att börja med kan det vara ett enformigt arbete för en människa att bevisa väldigt många teorem. Detta fick Whitehead och Russel erfara under arbetet med Principia Mathematica, vilket var ett försök att härleda alla matematiska sanningar från en väldefinierad mängd axiom och inferensregler i symbolisk logik. Det fanns en planerad fjärde volym om geometri som aldrig blev färdigställd eftersom de blev utmattade av arbetet med de tidigare volymerna. Nästa argument är det faktum att människor gör misstag, en dator gör det däremot inte (givet att den har blivit korrekt programmerad av en människa). Bevis kan lätt bli oöverskådliga för en människa, vilket gör det väldigt svårt att arbeta med dem. Det kan även finnas ett värde i att det är just en maskin som drar slutsatsen eftersom den då kan börja dra självständiga rationella slutsatser. Detta är ett av målen för forskningen inom artificiell intelligens.

Ett exempel på användningsområden för automatiska teorembevisare är inom utveckling av processorer. Både Intel och AMD har använt automatiska teorembevisare för att verifiera att deras implementation av flyttalsdivision är korrekt [14]. Ett annat exempel är ett program skrivet av Alan Newell, Herbert Simon och J. C. Shaw redan 1955 som hette Logic Theorist. Detta program lyckades bevisa 38 av de första 52 teoremen i Principia Mathematica, det lyckades till och med hitta nya och mer eleganta bevis till vissa teorem [25].

1.2 Syfte

Projektets syfte är att studera olika implementationsmetoder för att verifiera, bevisa och visualisera logiska slutledningar samt att ta fram en representation av bevis. I arbetet studeras predikatlogik med naturlig deduktion som deduktionskalkyl.

Studien berör tre olika metoder för automatisk teorembevisning: resolution, semantisk tableau och narrowing. Dessa metoder har jämförts utifrån prestanda, implementationssvårighet, läsbarhet av genererade bevis samt vilka sorters problem de olika metoderna arbetar bäst med.

1.3 Metod

Tidigt under projektets gång beslutades att det funktionella programmeringsspråket Haskell skulle användas. En av anledningarna till detta var Haskells stöd för rekursiva datatyper vilket är ett väldigt naturligt sätt att representera olika strukturer inom predikatlogik. Detta eftersom många av strukturerna är rekursivt definierade.

För att direkt komma igång med att definiera det logiska språket beslutades att BNFC [20] skulle användas. BNFC är ett hjälpmedel vid kompilatorkonstruktion och genererar funktionalitet för lexikal- och syntaktisk analys utifrån en användardefinierad BNF[2]-grammatik. Meningen med detta var att snabbt få en stabil grund att bygga vidare på.

Arbetsgången har tidsmässigt varit indelad i tre perioder. Under den första definierades projektmål och programmeringsspråk samt övriga verktyg. Under denna period läste projektgruppen även in sig på predikatlogik.

I den andra perioden definierades grammatiken för det logiska språket. Vidare implementerades datastrukturer för att representera bevis samt konverteringsfunktioner för att konvertera mellan dessa. Slutligen för samma period konstruerades en bevis-

[2]Backus–Naur form

verifierare samt funktionalitet för visualisering av bevis på listform och som naturliga deduktionträd i LaTeX.

Tillsist i den tredje perioden delades projektgruppen upp i mindre delgrupper som tilldelades varsitt område inom teorembevisning. Delgrupperna implementerade varsin teorembevisare vilka slutligen jämfördes med hänsyn till prestanda, läsbarhet av genererade bevis och implementationssvårighet.

1.4 Projektmål

Målet med projektet är att implementera tre teorembevisare och dra slutsatser baserade på jämförelser av dessa. De baseras på olika metoder för teorembevisning och därför produceras bevis till teorem på olika sätt. Den första är baserad på inferensregeln resolution, den andra är semantisk tableau och den tredje är narrowing. För att kunna göra detta ska en representation av bevis tas fram och för verifiering av korrektheten hos genererade sådana implementeras även en bevisverifierare. Denna används till att både kontrollera bevis genererade av teorembevisarna och bevis som användaren skriver in. Till sist skall det vara möjligt att visualisera bevis på ett lättläsligt och tydligt sätt.

Här visas en komplett lista av projektmålen:

- Framtagning av en logik av typen (första ordningens) predikatlogik.
- Framtagning av en representation av bevis.
- Framtagning av en abstrakt syntax som tillåter översättning från text till den interna representationen av formler och bevis.
- Implementering av en bevisverifierare.
- Visualisering av bevis på listform samt trädform.
- Funktionalitet för konvertering av bevis mellan list- och trädform.
- Implementering av teorembevisare som bygger på inferensregeln resolution, semantisk tableau och sökalgoritmen narrowing.

Kapitel 2

Teori

I detta kapitel definieras och förklaras det logiska system som projektet berör. En deduktionskalkyl med tillhörande inferensregler beskrivs och allmänna egenskaper för logiska system definieras. Avslutningsvis beskrivs teorin för de tre bevismetoder som studien handlar om.

2.1 Predikatlogik

För att kunna dra otvetydiga slutsatser behövs ett formellt språk att göra detta i. Valet är en logik av första ordningen vilken har en stor uttryckbarhet[1] samtidigt som den är, till skillnad från naturliga språk, helt otvetydig. Den innehåller grundläggande konnektiver från satslogiken samt predikat och kvantifierare, vilket är förutsättningen för att en logik ska vara av första ordningen. Predikatlogik är tillräckligt kraftfullt att uttrycka och formalisera bland annat två av de viktigare matematiska grundstenarna. Den första är en mängdlära formulerad genom axiom kallad ZFC[2][12], där man försöker undvika klassiska motsägelser som t.ex. Russels paradox[24]. Den andra är Peano-aritmetik vilket är ett aritmetiskt system kraftfullt nog att representera de naturliga talen genom ett antal rekursiva axoim. Dock kan predikatlogik inte representera sådant som hur ofta ett predikat P håller.

Formler byggs upp rekursivt av literaler ihopkopplade med unära och binära konnektiver. Nedan konstrueras ett formellt språk där syntaxen och semantiken förklaras. Med detta språk används en deduktionskalkyl som beskrivs i avsnitt 2.2.

2.1.1 Termer

En term är ett uttryck som refererar till ett objekt och är antingen en konstant, variabel eller funktion. En variabel kan anta vilket värde som helst som en term kan anta. En funktion består av en identifierare samt en lista med parametrar. Dessa är i sin tur termer, vilket gör en funktion till en rekursiv struktur.

Definition 1 *Termer*

- Vilken variabel som helst är en term.
- En funktion med noll aritet[3] kallas för en konstant och är en term.

[1] Eng. Expressability.
[2] Eng. Zermelo-Fraenkel set theory, with the axiom of choice.
[3] Aritet är antalet inparametrar.

- Om $t_1, ..., t_n$ är termer och f är en funktion med aritet $n > 0$ så är $f(t_1, ..., t_n)$ en term.

Ett exempel på en funktion är $bror(x)$, vilket motsvarar objektet som är bror till x, där x är en variabel. Om funktionen skrivs om till $bror(John)$ så motsvarar detta Johns bror. Här är '$John$' en konstant. Genomförs ytterligare en ändring; $bror(far(John))$ så kommer denna funktion returnera objektet som är Johns farbror. Variabler, konstanter och funktionsidentifierare skrivs med små bokstäver.

2.1.2 Formler

En formel kallas välbildad omm nedanstående regler rekursivt kan appliceras tills varje delformel delats upp i atomer. De något avancerade byggstenarna för formler kommer först att förklaras kort och sedan följer en formell definition av en formel.

Ett predikat liknar en funktion, men istället för att returnera ett objekt, returnerar det istället ett boolskt värde. Absurdheten indikerar att något är absurt, t.ex. att φ och $\neg\varphi$ existerar samtidigt i samma kontext. Denna formel skrivs $\bot$. Här introduceras två kvantifierare, allkvantifiering och existenskvantifiering, som båda kvantifierar över en viss variabel x. Allkvantifiering innebär att en given formel gäller för samtliga x, medan en existenskvantifiering säger att formeln gäller för minst ett fall av x.

Definition 2 *Formler*

- Om φ är en formel , då är $\neg\varphi$ en formel.
- Om φ och ψ är formler, då är även $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$ formler.
- Om φ är en formel och x är en variabel så är både $\forall x.\, \varphi$ och $\exists x.\, \varphi$ en formel.
- Om P är en predikatsymbol med aritet $n \geq 1$ och om $t_1, ..., t_n$ är termer så är $P(t_1, ..., t_n)$ en formel.
- Vilken term t som helst är en formel.
- $\bot$ är en formel.

De två första reglerna innehåller konnektiver som beskrivs nedan:

- Konnektivet $\wedge$ kallas konjunktion och skall tolkas som 'och'. För att en formel med denna på toppnivå ska vara sann måste båda delformlerna vara sanna.
- Nästa konnektiv kallas disjunktion och skrivs $\vee$. Detta konnektiv tolkas som 'eller'. En formel med denna på toppnivå är sann om minst en av delformlerna är sanna.
- Konnektivet $\rightarrow$ kallas implikation och innebär att den vänstra delformeln medför den högra. En formel $p \rightarrow q$ utläses "om p så q". En formel med implikation är endast falsk om den vänstra delformeln är sann och den högra är falsk.
- Slutligen kommer konnektivet $\neg$ som kallas negation. En formel $\neg\phi$ är endast sann om ϕ är falsk.

Konnektiverna $\wedge$, $\vee$ och $\neg$ är triviala men $\rightarrow$ är inte lika lätt att förstå. Resomanget för den sistnämnda följer: om den vänstra delformeln är sann så borde även den högra vara sann, om den vänstra delformeln är falsk, spelar den högra ingen roll. Exempelvis om det regnar medför detta att marken är blöt. Om det inte regnar kan marken antingen vara torr eller blöt. Det kan ha varit uppehåll tillräckligt länge för att marken ska ha

torkat eller det kan nyss ha slutat regna vilket innebär att marken fortfarande är blöt. Faktumet att det inte regnar utesluter inte något av de två alternativen. Slutsatsen att marken är torr på grund av att det regnar, är dock felaktig.

Semantiken för dessa konnektiver kan enkelt beskrivas med sanningsvärdestabeller, vilket visas i tabell 2.1.2:

Tabell 2.1: *Sanningtabell över satslogiska konnektiver*

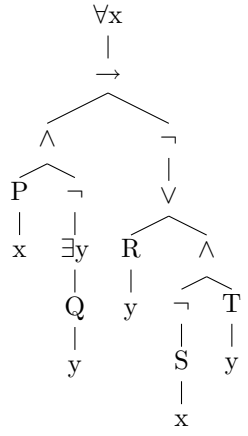
φ	ψ	$\varphi \wedge \psi$
Sant	Sant	Sant
Sant	Falskt	Falskt
Falskt	Sant	Falskt
Falskt	Falskt	Falskt

φ	ψ	$\varphi \vee \psi$
Sant	Sant	Sant
Sant	Falskt	Sant
Falskt	Sant	Sant
Falskt	Falskt	Falskt

φ	ψ	$\varphi \rightarrow \psi$
Sant	Sant	Sant
Sant	Falskt	Falskt
Falskt	Sant	Sant
Falskt	Falskt	Sant

φ	$\neg\varphi$
Sant	Falskt
Falskt	Sant

På samma sätt som matematiska operatorer så binder dessa konnektiver olika hårt. De som binder starkast är de unära konnektiverna $\neg$, $\forall$ och $\exists$. Därefter kommer $\wedge$, sedan $\vee$ och till sist $\rightarrow$, som binder svagast och är högerassociativ. Formeln $\forall x.(P(x) \wedge \neg\exists y.Q(y) \rightarrow \neg(R(y) \vee \neg S(x) \wedge T(y)))$ används i figur 2.1 för att visa bindning av de logiska konnektiverna:



Figur 2.1: *Syntaxträd för formeln* $\forall x.(P(x) \wedge \neg\exists y.Q(y) \rightarrow \neg(R(y) \vee \neg S(x) \wedge T(y)))$

Nedan följer definitionen av literaler vilket är formlers atomer.

Definition 3 *En literal är antingen en term, ett predikat eller* $\bot$*. Om* ϕ *är en literal, är även* $\neg\phi$ *en literal.*

Detta är definitionen av en klausul:

Definition 4 *En klausul är en icke tom mängd av literaler sammanbundna av disjunktioner.*

2.1.3 Predikat

Ett predikat liknar en funktion och består av en identifierare samt en lista med argument. Skillnaden mot en funktion är att ett predikat är en formel och returnerar därför ett boolskt värde, till skillnad från en funktion som refererar till ett objekt. Ett exempel på ett predikat ses nedan:

$$Yngre(x, John)$$

Predikatet kommer vara sant om x är yngre än $John$, annars falskt.

2.1.4 Allkvantifiering ($\forall$)

En allkvantifiering $\forall x.\phi$ motsvarar en formel där alla möjliga värden på x insatta i varsitt ϕ binds samman med konjunktioner. Ett exempel på denna kvantifierare visas nedan:

$$\forall x.(Katt(x) \rightarrow Djur(x))$$

Exemplet kan läsas som att för alla x sådant att x är en katt, gäller det också att x är ett djur. I ett system utan allkvantifiering skulle alla möjliga värden på x som är katt behöva räknas upp och sedan bindas samman med konjuktioner.

2.1.5 Existenskvantifering ($\exists$)

Om allkvantifiering kan ses som en konjunktion av formler, kan existenskvantifiering $\exists x.\phi$ istället ses som en disjunktion av formler. För att kvantifieringen ska vara sann räcker det alltså med att ϕ är sann för ett värde på x. Ett exempel visas nedan:

$$\exists x.(Hatt(x) \wedge Huvud(x, Erik))$$

Detta uttryck läses: det existerar minst ett x sådant att x är en hatt och att x är på Eriks huvud.

2.1.6 Logisk och semantisk konsekvens

För logiska system går det att definiera två olika relationer där den ena gäller på det syntaktiska planet och den andra på det semantiska. Logisk konsekvens är ett av de mest fundamentala koncepten inom fältet logik. Detta skrivs $\Gamma \vdash A$ och tolkas som att Γ är en mängd av formler och att A symboliskt kan deriveras ur Γ. Att *Aristoteles är dödlig* är en logisk konsekvens av att *alla människor är dödliga* och att *Aristoteles är en människa*.

Semantisk konsekvens skrivs $A \vDash B$ och betyder att A medför[4] B om en tolkning som gör alla formler i A sanna gör B sann. Skillnaden mot $\vdash$ är att den här relationen gäller semantiskt. För att det ska gälla så krävs det att alla formler i A är sanna och att B är sann.

2.1.7 Substitution

För att en formel innehållande variabler skall betyda något mer konkret, behövs ett sätt att substituera variabeln mot någon konkret information. Definitionen för hur denna substitution skall gå till kräver att både konceptet med scope samt konceptet med fria och bundna variabler är definierade.

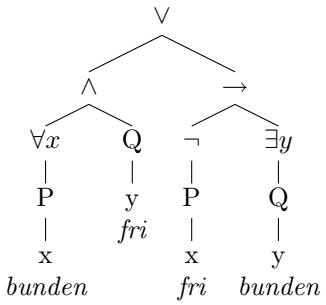
[4]Eng. Entails.

Definition 5 *För de två formlerna $\forall x.\phi$ och $\exists x.\phi$ så är scopet för $\forall x$ och $\exists x$ lika med ϕ minus alla subformler $\forall x.\psi$ och $\exists x.\psi$ i ϕ.*

Exempelvis så är scope för $\exists x$ i $\exists x.(P(x) \vee \forall x.Q(x))$ bara $P(x)$.

Definition 6 *En variabel kallas fri om den inte är i ett scope som binder den till en kvantifiering. Motsatsen till en fri variabel är en bunden variabel.*

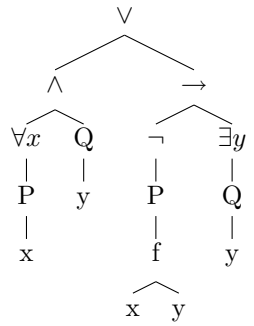
För att tydligt illustrera konceptet med fria och bundna variabler kan man rita ett parseträd där $\forall x$ och $\exists x$ är noder med endast ett subträd och där även predikat är noder med predikatsymbolen i noden och lika många subträd som den har aritet. I figur 2.2 visas ett syntaxträd för $(\forall x.P(x) \wedge Q(y)) \vee (\neg P(x) \rightarrow \exists y.Q(y))$ med information om variablerna är fria eller bundna.



Figur 2.2: *Syntaxträd för* $(\forall x.P(x) \wedge Q(y)) \vee (\neg P(x) \rightarrow \exists y.Q(y))$

Definition 7 *Om $v_1, ..., v_n$ är variabler och $t_1, ..., t_n$ är termer, så kallas en mängd av översättningar[5] från fria variabler till termer $\{t_1/\, v_1, ..., t_n/\, v_n\}$ för en substitution.*

Om substitutionen $\{f(x,y)/x\}$ appliceras på $(\forall x.P(x) \wedge Q(y)) \vee (\neg P(x) \rightarrow \exists y.Q(y))$ erhålls $(\forall x.P(x) \wedge Q(y)) \vee (\neg P(f(x,y)) \rightarrow \exists y.Q(y))$, parseträdet visas i figur 2.1.7:



Figur 2.3: *Syntaxträd för substitutionen* $\{f(x,y)/x\}$

Substitutioner kan dock leda till otrevliga sidoeffekter. Definitionen nedan löser dessa:

Definition 8 *Givet en term t, en variabel x och en formel ϕ så är t fri med avseende på x om inget fritt x i ϕ är i scope för $\forall y$ eller $\exists y$ för någon variabel y i t.*

[5] Eng. Mappings.

Ett exempel på ett fall som definition ovan löser är: Om subtitutionen $\{f(x)/y\}$ appliceras på $\forall x.(P(x) \wedge Q(y))$ erhålls en formel där den insatta termen blir bunden. För att eliminera denna sidoeffekt behövs det en definition som specificerar när en term är fri med avseende på en variabel.

Nedan följer en algoritm för hur en substitution appliceras på en formel:

1. Applicera substitutionen rekursivt över samtliga konnektiv i formeln. Om σ är en substitution så appliceras den enligt:

   - $(\phi \wedge \psi)\,\sigma \iff (\phi\,\sigma) \wedge (\psi\,\sigma)$
   - $(\phi \vee \psi)\,\sigma \iff (\phi\,\sigma) \vee (\psi\,\sigma)$
   - $(\phi \rightarrow \psi)\,\sigma \iff (\phi\,\sigma) \rightarrow (\psi\,\sigma)$
   - $(\neg\phi)\,\sigma \iff \neg(\phi\,\sigma)$
   - $(\forall x.\phi)\,\sigma \iff \forall x.(\phi\,\sigma)$
   - $(\exists x.\phi)\,\sigma \iff \exists x.(\phi\,\sigma)$.

2. Applicera substitutionen på predikat och funktioner rekursivt över inparametrarna enligt:

   - $P(t_1, ..., t_n)\,\sigma \iff P(t_1\,\sigma, ..., t_n\,\sigma)$
   - $f(t_1, ..., t_n)\,\sigma \iff f(t_1\,\sigma, ..., t_n\,\sigma)$

3. Om en fri variabel påträffas och den finns med i σ översätts den enligt substitutionen.

4. Inga andra formler eller termer berörs av substitutionen.

2.2 Naturlig deduktion

För att det logiska system som valts för projektet skall bli intressant, behövs det någon form av deduktionskalkyl för att kunna bevisa teorem. Valet som gjorts är att studera naturlig deduktion vilket är ett deduktionssystem som strävar efter att beskriva logiskt resonemang på ett sätt som känns naturligt. Med naturligt menas att det skall vara intuitivt och påminna så mycket som möjligt om mänskligt resonerande. Systemet introducerades som ett alternativ till axiombaserade system som exempelvis Principia Mathematica. Grunden till naturlig deduktion som det är känt idag lades av Gentzen i [7]. Prawitz vidareutvecklade det för modal och andra ordningens logik i [21].

En deduktion (eller ett bevis) kan formellt definieras som:

Definition 9 *En ändlig sekvens $\beta_1, ..., \beta_n$ av satser kallas för en deduktion av satsen α från en samling premisser $\sum$ om $\beta_n = \alpha$ och för alla $1 \leq i \leq n$ så gäller att $\beta_i \in \sum$ eller att β_i är resultatet av en inferensregel applicerad på en eller flera tidigare satser.*

Man börjar alltså med en samling premisser och applicerar sedan ett antal inferensregler tills man i slutändan har härlett sin slutsats. Man kan välja att ha ett deduktionssystem med färre regler än vad som valts för detta projekt eftersom många av dem går att härleda med hjälp av de andra. Färre deduktionsregler leder visserligen till längre och mer svårlästa bevis, men eftersom systemet är mindre är det lättare att implementera. Egenskapen att kunna producera tydliga bevis har prioriterats i detta projekt och därför har fler regler inkluderats än vad som är absolut nödvändigt.

Ett sätt att beskriva deduktionsregler är att ha ett vågrätt streck, med ett antal formler ovanför strecket och en formel nedanför. Till höger om strecket skrivs namnet

på den regel som appliceras. Formlerna ovanför strecket är de man vill applicera den angivna regeln på, och den under strecket är den resulterande formeln efter att regeln applicerats. På detta sätt kan man på ett enkelt sätt visualisera bevis genom att bara bygga vidare uppåt. De formler som man vill applicera sin inferensregel på är i sin tur resultatet av tidigare inferenser.

2.2.1 Konjunktion

De första regler som skall studeras är regler för att introducera och eliminera konjunktioner. För att introducera en konjunktion så behöver man två formler som man sedan skriver en konjunktion emellan. Detta skrivs:

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \wedge_i \tag{2.1}$$

Elimination av en konjunktion kan göras på två olika sätt. Antingen behåller man den vänstra delformeln (i förhållande till konjunktionen) och utelämnar den högra eller så behåller man den högra och utelämnar den vänstra. Dessa två olika alternativ kallas för konjunktionselimination 1 respektive 2 och skrivs:

$$\frac{\phi \wedge \psi}{\phi} \wedge_{e1} \qquad \frac{\phi \wedge \psi}{\psi} \wedge_{e2} \tag{2.2}$$

Med hjälp av dessa två regler är det möjligt att bevisa att $p \wedge q, r \vdash p \wedge r$. Detta görs genom att applicera $\wedge_{e1}$ på första premissen och sedan $\wedge_i$ på resultatet och den andra premissen. Detta visas i figur 2.4.

$$\frac{\dfrac{p \wedge q}{p} \wedge_{e1} \quad r}{p \wedge r} \wedge_i$$

Figur 2.4: *Exempel med elimination och introduktion av konjunktioner.*

2.2.2 Implikation

Att introducera en implikation är dock inte lika lätt. Om man vill applicera implikationsintroduktion på två formler ϕ och ψ så måste man visa att det verkligen går att härleda ψ ur ϕ. Kan man det så går det att säga att $\phi \rightarrow \psi$. Denna regel skrivs:

$$\frac{\begin{array}{c}[\phi] \\ \vdots \\ \psi\end{array}}{\phi \rightarrow \psi} \rightarrow_i \tag{2.3}$$

Eliminering av implikation kallas även *modus ponens,* vilken säger att om man har ϕ och $\phi \rightarrow \psi$ så kan man sluta sig till ψ. Detta är väldigt intuitivt, ett exempel på denna regel är det klassiska resonemanget:

- Alla människor är dödliga. Aristoteles är en människa.
- ∴Aristoteles är dödlig.

Regeln skrivs:

$$\frac{\phi \quad \phi \to \psi}{\psi} \to_e \tag{2.4}$$

Genom att använda dessa två regler tillsammans med introduktions- och eliminationsreglerna för konjunktioner går det att bevisa att $p \to (q \to r) \vdash p \wedge q \to r$. Detta görs genom att först anta att $p \wedge q$ gäller och utifrån det deducera r. När detta är gjort är det bara att applicera $\to_i$ och beviset är fullbordat. Detta visas i figur 2.5.

$$\dfrac{\dfrac{p \to (q \to r) \quad \dfrac{[p \wedge q]}{p} \wedge_{e1}}{q \to r} \to_e \quad \dfrac{[p \wedge q]}{q} \wedge_{e2}}{\dfrac{r}{p \wedge q \to r} \to_i} \to_e$$

Figur 2.5: *Exempel med introduktion och elimination av implikationer.*

2.2.3 Disjunktion

För introduktion av disjunktion finns det två olika versioner. Man introducerar en godtycklig formel antingen på höger eller vänster sida om en disjunktion där den motsatta sidan är en redan existerande formel. Reglerna skrivs som:

$$\frac{\phi}{\phi \vee \psi} \vee_{i1} \qquad \frac{\phi}{\psi \vee \phi} \vee_{i2} \tag{2.5}$$

Elimination av en disjunktion är mer komplex. Givet $\phi \vee \psi$ måste man visa att en formel χ är deriverbar ur både ϕ och ψ. Kan man göra detta så kan man sluta sig till χ. Detta kan kännas ointuitivt, men om man funderar över vad $\phi \vee \psi$ betyder så kan man inse att det är rimligt. Formeln $\phi \vee \psi$ är sann om antingen ϕ eller ψ är sann eller om båda är sanna. Om man kan härleda en formel χ från både ϕ och ψ så måste den gälla i alla de tre fallen, det vill säga när bara ϕ är sann, eller när bara ψ är sann eller när båda är det. Om χ gäller oberoende av vilken av ϕ eller ψ som är sann så kan man tryggt sluta sig till att den måste gälla och då kan man eliminera disjunktionen. Regeln skrivs som:

$$\frac{\phi \vee \psi \quad \begin{matrix}[\phi] \\ \vdots \\ \chi\end{matrix} \quad \begin{matrix}[\psi] \\ \vdots \\ \chi\end{matrix}}{\chi} \vee_e \tag{2.6}$$

Med hjälp av denna regel går det att bevisa att $p \vee q \vdash q \vee p$. Beviset går att härleda i fyra steg. Antag p, deducera $q \vee p$, antag sedan q och deducera $q \vee p$. När detta är gjort är χ i regeln ovan deducerad från både p och q vilket gör att det är möjligt att applicera regeln. Beviset visas i figur 2.6:

$$\dfrac{p \vee q \quad \dfrac{[p]}{q \vee p} \vee_{i2} \quad \dfrac{[q]}{q \vee p} \vee_{i1}}{q \vee p} \vee_e$$

Figur 2.6: *Exempel med introduktion och elimination av disjunktioner.*

2.2.4 Negation

Negationsintroduktion är även den en komplicerad regel. Den påminner om implikationsintroduktion, men istället för att härleda en godtycklig formel så ska man kunna härleda $\bot$. Detta kan motiveras med att $\neg\phi$ är definierat som $\phi \to \bot$. Man kan även motivera det med följande resonemang: om $\neg\phi$ är sann så måste ϕ leda till $\bot$. Denna regel skrivs:

$$\dfrac{\begin{array}{c}[\phi]\\ \vdots \\ \bot\end{array}}{\neg\phi}\ \neg_i \tag{2.7}$$

Elimination av en negation sker på följande sätt. Om man i en given kontext har en formel ϕ och samtidigt dess logiska motsats $\neg\phi$, så kan man derivera $\bot$. Detta eftersom de två aldrig kan vara giltiga samtidigt. Detta skrivs:

$$\dfrac{\phi \quad \neg\phi}{\bot}\ \neg_e \tag{2.8}$$

2.2.5 Allkvantifiering

Allkvantifierarelimination innebär att man reducerar den kvantifierade mängden variabler till en specifik term. Om formeln ϕ gäller för alla x, gäller den även för termen t, eftersom t är en delmängd av alla x. Detta är givet att t är fri i ϕ. Termen t kan här ses som en mer konkret instans av x. Denna regel skrivs:

$$\dfrac{\forall x.\phi}{\phi\{t/x\}}\ \forall x_e \tag{2.9}$$

Introduktionen av en allkvantifierare går till på följande sätt: kan man i scopet av en helt ny variabel x_0 derivera en formel ϕ som beror av av den nya variabeln, så kan man även anta att formeln gäller för alla x. Detta eftersom inga tidigare antaganden gjorts om x_0 och den anses vara en helt godtycklig variabel. Gäller ϕ för en godtycklig variabel x_0, gäller den också för samtliga x. Detta skrivs:

$$\dfrac{\begin{array}{c}[x_0]\\ \vdots \\ \phi\{x_0/x\}\end{array}}{\forall x.\phi}\ \forall x_i \tag{2.10}$$

Med hjälp av dessa båda regler går det att bevisa att $\forall x.(P(x) \to Q(x)), \forall x.P(x) \vdash \forall x.Q(x)$. Detta kan göras genom att välja en variabel x_0 och sedan eliminera allkvantifierarna i premisserna. Sedan är det bara att applicera *modus ponens* och erhålla $Q(x_0)$, detta är då ett bevis för att $Q(x)$ gäller för godtyckliga värden på x och allkvantifieraren kan då introduceras. Beviset visas i figur 2.7:

$$\dfrac{\dfrac{\dfrac{\forall x.(P(x) \to Q(x))}{P(x_0) \to Q(x_0)}\ \forall x_e \quad \dfrac{\forall x.P(x)}{P(x_0)}\ \forall x_e}{Q(x_0)}\ \to_e}{\forall x.Q(x)}\ \forall x_i$$

Figur 2.7: *Exempel med elimination och introduktion av allkvantifierare.*

2.2.6 Existenskvantifiering

Givet en formel ϕ som beror av en term t så är det möjligt att införa en existenskvantifiering, där x ersätter t. Resonemanget bygger på att det finns minst ett x sådant att ϕ är sann, nämligen termen t. Regeln för detta skrivs :

$$\frac{\phi\{t/x\}}{\exists x.\phi}\ \exists x_i \tag{2.11}$$

För att eliminera en existenskvantifiering $\exists x.\phi$ gör man ett antagande att ϕ gäller för den generella variabeln x_0. Är det sedan möjligt att från antagandet derivera en formel ψ, som inte beror av x_0, så går det att sluta sig till att ψ gäller. Detta är ännu ett exempel på en regel som kan verka ointuitiv vid en första anblick, men som är rimlig om man funderar över vad den egentligen säger. Formeln $\exists x.\phi$ säger att det finns minst ett värde på x sådant att ϕ är sann. Kan man härleda en formel ψ som inte beror på den generella variabeln x_0, så gäller ψ oavsett vilket värde eller värden på x som gör att ϕ är sann. Denna regel skrivs:

$$\frac{\exists x.\phi \qquad \begin{array}{c}[\phi\{x_0/x\}]\\ \vdots\\ \psi\end{array}}{\psi}\ \exists x_e \tag{2.12}$$

Med hjälp av dessa två regler går det att bevisa att $\forall x.(P(x) \rightarrow Q(x)), \exists x.P(x) \vdash \exists x.Q(x)$. För att bevisa detta antar man att $P(x)$ gäller för ett godtyckligt värde x_0 på x. Nästa steg är att eliminera allkvantifieraren i den första premissen och det är sedan möjligt att applicera *modus ponens* och då erhålla $Q(x_0)$. I och med att $Q(x)$ gäller för ett värde x_0 på x så är det möjligt att introducera en existenskvantifierare och därmed är $\exists x.Q(x)$ deducerat från $P(x)\{x_0/x\}$ vilket gör det möjligt att applicera existenskvantifierarelimination. Beviset visas i figur 2.8:

$$\dfrac{\exists x.P(x) \qquad \dfrac{\dfrac{[P(x_0)] \qquad \dfrac{\forall x.(P(x) \rightarrow Q(x))}{P(x_0) \rightarrow Q(x_0)}\ \forall x_e}{Q(x_0)}\ \rightarrow_e}{\exists x.Q(x)}\ \exists x_i}{\exists x.Q(x)}\ \exists x_e$$

Figur 2.8: *Exempel med introduktion och elimination av existenskvantifierare.*

2.2.7 Absurdhet

Eftersom $\perp$ är absurdheten så kan man härleda *allt* ur den. Detta skrivs som:

$$\frac{\perp}{\phi}\ \perp_e \tag{2.13}$$

2.2.8 Motsägelsebevis

En väldigt användbar regel är den för motsägelsebevis (lat. *reducio ad absurdum*). Denna regel är själva kärnan i bevismetoderna resolution och semantisk tableau. Det finns väldigt många bevis som bygger på denna regel, bland annat Euklides välkända bevis

att det finns oändligt många primtal[6]. Denna regel skrivs enligt:

$$\dfrac{\begin{array}{c}[\neg\phi]\\ \vdots\\ \bot\end{array}}{\phi}\ RAA \tag{2.14}$$

2.2.9 Modus Tollens

En regel som påminner väldigt mycket om *modus ponens* är *modus tollens*, den säger att om man har $\neg\psi$ och $\phi \rightarrow \psi$ så gäller $\neg\phi$. Ett exempel på denna regel är:

- Alla människor är dödliga. En sten är inte dödlig.
- ∴En sten är inte en människa.

Detta skrivs:

$$\dfrac{\neg\psi \quad \phi \rightarrow \psi}{\neg\phi}\ MT \tag{2.15}$$

2.2.10 Dubbelnegation

Två andra intuitiva regler är introduktion och elimination av dubbelnegation. Dessa skrivs:

$$\dfrac{\phi}{\neg\neg\phi}\ \neg\neg_i \tag{2.16}$$

$$\dfrac{\neg\neg\phi}{\phi}\ \neg\neg_e \tag{2.17}$$

2.2.11 Lagen om det uteslutna tredje

Den sista regeln i vår deduktionskalkyl är lagen om det uteslutna tredje[7]. Ett exempel på denna regel är Shakespeares ”Att vara eller icke vara” som formaliseras $\phi \vee \neg\phi$. Detta är en uppenbar tautologi och man behöver alltså inte ha något underlag för att kunna härleda det. Regeln skrivs:

$$\dfrac{}{\phi \vee \neg\phi}\ LEM \tag{2.18}$$

Intuitionistisk logik är ett exempel på ett logiskt system där varken elimination av dubbelnegation, lagen om det uteslutna tredje eller motsägelsebevis gäller. Anledning är att intuitionistisk logik har sina rötter i intuitionismen som säger att objekt måste kunna konstrueras mentalt innan det går att resonera om dem. Detta går emot den traditionella logiken som säger att existensen av ett objekt kan bevisas genom att motbevisa dess icke-existens. Detta är anledningen till att motsägelsebevis inte accepteras av intuitionistisk logik. Av detta följer att bevis i intuitionistisk logik endast kan anses giltiga om det finns en metod som kan skapa de objekt beviset behandlar.

Lagen om det uteslutna tredje inkluderas inte i den intuitionistiska logiken eftersom det går att konstruera ett matematiskt påstående som varken kan bevisas eller motbevisas.

Elimination av dubbelnegation tillåts ej i intuitionistisk logik eftersom synen på negation här skiljer sig från traditionell logik, som säger att negationen av ett sant påstående innebär att detta är falskt. I den intuitionistiska logiken ses detta som ett

[6]Beviset återfinnes i de flesta böcker om elementär talteori eller exempelvis i [22].
[7]Eng. Law of the excluded middle.

bevis av att påståendet inte går att bevisa. Om ett påstående P går att bevisa, så är det omöjligt att bevisa att det inte går att bevisa. Dock så innebär avsaknaden av ett bevis för att det inte finns något bevis för P, att det faktiskt finns ett bevis för P och därför är P ett starkare påstående än $\neg\neg P$.

2.3 Normalformer

I detta avsnitt presenteras två olika normalformer, nämligen disjunktiv- och konjunktiv normalform. Alla formler i första ordningens logik kan konverteras till en ekvivalent formel på konjunktiv- eller disjunktiv normalform.

En formel är i konjunktiv normalform omm den är en konjunktion av klausuler. Exempelvis är $\neg\phi \wedge (\psi \vee \varphi)$ på konjunktiv normalform, men $(\neg\phi \wedge \psi) \vee \varphi$ är det inte. Konjunktiv normalform är nödvändig för inferensregeln resolution, som bara kan appliceras på klausuler.

En formel är i disjunktiv normalform omm den är en disjunktion av konjunktiva klausuler. En konjunktiv klausul är en mängd literaler sammankopplade med konjunktioner. Exempelvis är $(\neg\phi \wedge \psi) \vee \varphi$ på disjunktiv normalform, men $\neg\phi \wedge (\psi \vee \varphi)$ är inte det. Disjunktiv normalform är lämplig för semantisk tableau eftersom disjunktioner hamnar på toppnivå i formler och därmed blir färre till antalet. Detta kommer att förklaras närmare senare i rapporten.

Algoritmen för att konvertera en godtycklig formel av första ordningens logik till dessa normalformer beskrivs nedan:

1. Eliminera implikationer genom att byta ut $\phi \rightarrow \psi$ mot $\neg\phi \vee \psi$.

2. Flytta negationer inåt i formeln genom att applicera De Morgans lagar:

   - $\neg(\phi \vee \psi) \iff \neg\phi \wedge \neg\psi$
   - $\neg(\phi \wedge \psi) \iff \neg\phi \vee \neg\psi$
   - $\neg\forall x.\phi \iff \exists x.\neg\phi$
   - $\neg\exists x.\phi \iff \forall x.\neg\phi$
   - $\neg\neg\phi \iff \phi$

   Till exempel blir $\neg(\neg\phi \wedge \psi)$ konverterat till $\phi \vee \neg\psi$.

3. Standardisera variabelnamn för att slippa tvetydigheter när man vid ett senare steg eliminerar kvantifierare. I formeln $\forall x.P(x) \vee \exists x.Q(x)$ representerar de två kvantifierade variablerna olika x. För att komma ifrån detta problem kan formeln skrivas om till $\forall x.P(x) \vee \exists y.Q(y)$.

4. Flytta alla kvantifierare längst till vänster i formeln, $\forall x.P(x) \vee \exists y.Q(y)$ blir $\forall x.\exists y.(P(x) \vee Q(y))$. Eftersom alla variabler nu är standardiserade så finns det inga variabler med samma namn, vilket medför att denna transformering kan utföras. Om variablerna inte hade varit standardiserade så hade varje fri variabel med samma namn som en bunden, efter transformering blivit bunden och formelns logiska innebörd hade då förändrats.

5. Eliminera existenskvantifierare genom att byta ut alla variabler som de kvantifierar mot skolemkonstanter eller skolemfunktioner. En skolemfunktion är en funktion $f(x_1, ..., x_n)$ där funktionsymbolen f inte redan förekommer i formeln och variablerna $x_1, ..., x_n$ är de variabler som är allkvantifierade. En skolemkonstant är en skolemfunktion med noll aritet.

6. Eftersom alla allkvantifierare nu ligger längst till vänster i formeln och de existenskvantifierade variablerna har ersatts med skolemkonstanter eller skolemfunktioner, är det säkert att anta att alla de kvarvarande variablerna i formeln är allkvantifierade. Man kan därför förkasta de kvarstående kvantifieringarna.

7. För konvertering till konjunktiv normalform utför steg (a), för disjunktiv normalform utför steg (b).

   (a) Distribuera disjunktioner över konjunktioner. $(\phi \wedge \psi) \vee \varphi$ blir $(\phi \vee \varphi) \wedge (\psi \vee \varphi)$ och $\phi \vee (\psi \wedge \varphi)$ blir $(\phi \vee \psi) \wedge (\phi \vee \varphi)$.

   (b) Distribuera konjunktioner över disjunktioner. $(\phi \vee \psi) \wedge \varphi$ blir $(\phi \wedge \varphi) \vee (\psi \wedge \varphi)$ och $\phi \wedge (\psi \vee \varphi)$ blir $(\phi \wedge \psi) \vee (\phi \wedge \varphi)$.

2.4 Unifiering

Denna metod utvecklades av Robinson i [23] för att användas inom automatisk teorembevisning. Unifiering utnyttjas frekvent inom logiska programspråk bland annat för att hantera instansiering av variabler. Den problematik som unifiering löser är att avgöra om det för en mängd formler finns en substitution som får dem att se likadana ut.

Definition 10 *En substitution $\sigma = \{t_1/\, v_1, ..., t_n/\, v_n\}$ kallas för en unifierare för mängden av formler $\{E_1, ..., E_m\}$ om $E_1\sigma = E_2\sigma = ... = E_m\sigma$.*

Om detta leder till någon motsägelse är formlerna inte unifierbara och ett negativt resultat returneras. Om formlerna däremot är unifierbara returneras en unifierare. För alla mängder av formler så finns det en unifierare som är mer generell än alla andra, denna kallas för den mest generella unifieraren.

Definition 11 *En substitution σ kallas den mest generella unifieraren om det inte finns något par av substitutioner (σ', τ) sådana att $\sigma' = \sigma\tau$.*

För de två substitutionerna $\sigma_1 = \{f(g(a,\, h(x)))/x,\, g(h(x),\, b)/y,\, h(x)/z\}$ och $\sigma_2 = \{f(g(x,\, y))/x,\, g(z,\, b)/y\}$ är σ_2 mer generell än σ_1 eftersom det finns en substitution $\tau = \{a/x,\, h(z)/y,\, h(x)/z\}$ sådan att $\sigma_1 = \sigma_2\tau$.

Ett exempel på formler som inte går att unifiera är $P(x,\, 1)$ och $P(2,\, x)$. Jämför man den första variabeln i de båda formlerna får man unifieraren $\{2/x\}$. När man sedan går vidare till den andra variabeln skall man lägga till $\{1/x\}$ till unifieraren, men eftersom detta leder till en motsägelse så kan det konstateras att det inte går att unifiera formlerna.

Ett specialfall som är värt att uppmärksamma inträffar när man försöker unifiera en formel f med en variabel x, där f innehåller x. Unifieringsalgoritmen kommer då att fastna i en oändlig loop. Det blir en cyklisk struktur som bara är giltig om f är identitetsfunktionen. För att implementera en komplett unifieringsalgoritm måste det testas om f innehåller x, det enda problemet med detta är att unifieringen blir mindre effektiv, närmare bestämt går komplexiteten från $\mathcal{O}(min(size(t_1),\, size(t_2)))$ till $\mathcal{O}(max(size(t_1),\, size(t_2)))$. Detta test kallas kontroll av variabelförekomst[8].

[8] Eng. Occurence check.

2.5 Metalogik

Metalogik eller metateori för logik är studiet av egenskaper som logiska teorier och system har. Nedan definieras några relevanta egenskaper för logiska system, dessa kopplas sedan till predikatlogik.

Definition 12 *Ett logiskt system kallas konsistent omm det inte finns några formler sådana att* $\phi_1, \ldots, \phi_n \vdash \psi$ *samtidigt som* $\phi_1, \ldots, \phi_n \vdash \neg\psi$

Konsistens är en intuitiv, väldigt viktig egenskap, om man har bevisat ψ så bör det inte vara möjligt att bevisa $\neg\psi$. I ett system med motsägelser går allt att bevisa (se formel 2.13).

Definition 13 *Om det med hjälp av systemets inferensregler enbart går att hitta bevis för semantiskt sanna följder säges systemet vara sunt. Med andra ord* $\phi_1, ..., \phi_n \vdash \psi \Rightarrow \phi_1, ..., \phi_n \vDash \psi$

Dess signifikans inses snabbt genom att föreställa sig meningslösheten hos ett inferenssystem där det går att finna bevis för falska påståenden.

Definition 14 *Ett inferenssystem säges vara fullständigt om det för varje semantiskt giltig följd existerar ett bevis* $\phi_1, ..., \phi_n \vDash \psi \Rightarrow \phi_1, ..., \phi_n \vdash \psi$.

I teorier eller system som har denna egenskap finns det bevis för samtliga teorem, vilket kan tyckas betryggande. Detta bevisades för predikatlogiken av Gödel i hans fullständighetsteorem[8]. Problemet är att så fort en teori blir tillräckligt komplicerad kommer detta inte nödvändigtvis gälla. Om en konsistent teori Γ uppfyller kravet att den kan uttrycka och bevisa enkla[9] aritmetiska sanningar, samtidigt som den kan uttrycka vissa sanningar om teorin själv kan man konstruera ett påstående som är sant i Γ men obevisbart. Även detta resultat bevisades av Gödel i [9] där även de exakta kraven på Γ presenteras. Konsekvenser som detta innebär för matematiken är att om man vill hålla den konsistent får man helt enkelt leva med att vissa saker inte går att bevisa.

Definition 15 *Ett problem säges vara avgörbart om det existerar en metod som på ett ändligt antal beräkningssteg kan sluta sig till ett svar.*

Predikatlogiken har visats vara oavgörbar vilket innebär att en sökning efter ett bevis kan i teorin hålla på i all oändlighet. Detta kan visas genom att exempelvis reducera problemet till en instans av "Post correspondence problem" vilket görs i [13].

2.6 Automatisk teorembevisning

Som det nämns i inledningen så finns det många motiveringar till att automatisera bevisningen av teorem. Detta projekt fokuserar på tre olika metoder. Den första är resolution, vilken bygger på att man applicerar en inferensregel med samma namn ända tills man härlett $\perp$ vilket leder till ett motsägelsebevis. Nästa metod är tableau. Denna unyttjar också motsägelsebevis för att derivera fram slutsatsen. Tableau arbetar sig fram genom att bryta isär konjunktioner och förgrena sökningen vid varje disjunktion. Detta fortsätter tills alla förgreningar leder fram till $\perp$ med vilket man kan härleda slutsatsen. Den tredje och sista metoden kallas för narrowing som är en generell metod för att lösa ekvationer i omskrivningssytem.

[9] Addition och multiplikation över naturliga tal.

2.6.1 Resolution

Denna metod introducerades 1965 av Robinson i [23] och kan delas upp i två delar. Den första delen är en inferensregel som härefter kommer att benämnas ”resolutionregeln”. Den andra delen kommer benämnas ”resolution” och syftar till en teknik för att bevisa teorem.

För att kunna applicera resolution krävs det att indata är i konjunktiv normalform (se avsnitt 2.3). Inferensregeln appliceras på två klausuler vilka har minst ett par komplementära literaler. Klausulerna knyts samman av en disjunktion, dubbletter av literaler tas bort och sedan cancelleras de komplementära literalerna. Detta eftersom en disjunktion av komplementära literaler, t.ex. $p \vee \neg p$, är en tautologi (vilket lätt inses om man studerar dess sanningsvärdestabell). Resultatet är en klausul som bara innehåller unika literaler. Är fallet sådant att samtliga literaler efter borttagningen av dubbletter ingår i ett komplementärt par, har man deriverat $\perp$. Nedan visas definitionen av resolutionregeln, där literalerna a_i och b_j är varandras komplement:

$$\frac{a_1 \vee ... \vee a_i \vee ... \vee a_n, \quad b_1 \vee ... \vee b_j \vee ... \vee b_m}{a_1 \vee ... \vee a_{i-1} \vee a_{i+1} \vee ... \vee a_n \vee b \vee ... \vee b_{j-1} \vee b_{j+1} \vee ... \vee b_m} \tag{2.19}$$

Nedan följer ett enklare exempel av resolutionregeln samt en tillhörande förklaring:

$$\frac{a \vee b, \quad \neg a \vee c}{b \vee c} \tag{2.20}$$

De två klausulerna har ett par komplementära literaler, nämligen a och $\neg a$. Om a är sann, så är $\neg a$ falsk och därför måste även c vara sann. Om a däremot är falsk, så måste b vara sann. Av dessa två slutsatser kan man komma fram till att, oberoende av värdet på a, så måste antingen b eller c vara sann, därav $b \vee c$.

Den andra delen är, som nämnt ovan, en teknik för att bevisa teorem som bygger på ett motsägelsebevis. Premisser samt den negerade slutsatsen konverteras till konjunktiv normalform och därefter bryts samtliga konjunktioner upp. Resolutionregeln appliceras sedan gång på gång tills att $\perp$ är deriverad.

För att tydligt illustrera hur resolution fungerar följer ett exempel. Uppgiften för teorembevisaren är att bevisa att:

$$(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge p \vdash r \tag{2.21}$$

Premissen är redan i konjunktiv normalform så det behövs inte göras någon konvertering. Det första steget är då att bryta upp formeln i klausuler och lägga in negationen av slutsatsen, $\neg r$, i klausullistan. Denna blir då:

$$[\neg p \vee \neg q \vee r, \neg p \vee q, p, \neg r]$$

Resolutionregeln appliceras sedan mellan första och andra formeln vilket ger:

$$[\neg p \vee \neg q \vee r, \neg p \vee q, p, \neg r, \neg p \vee r]$$

Nästa steg är att först applicera resolutionregeln mellan p och $\neg p \vee r$ vilket ger bara r. Därefter är att det bara att applicera regeln mellan r och $\neg r$ vilket ger:he

$$[\neg p \vee \neg q \vee r, \neg p \vee q, p, \neg r, \neg p \vee r, r, \perp]$$

Absurdheten är då deducerad och man har alltså funnit en motsägelse, vilket betyder att r är bevisad från $(\neg p \vee \neg q \vee r) \wedge (\neg p \vee q) \wedge p$.

2.6.2 Semantisk tableau

För att bevisa en formel ϕ med tableau, antas $\neg\phi$, varpå försök att derivera en motsägelse görs. Motsägelsen produceras genom att omvandla resultatet till ett tableauträd så som θ. Det består först bara av en gren men för varje disjunktion i θ som behandlas förgrenas trädet i två nya grenar. Uppgiften att producera en motsägelse för hela trädet kallas för att *stänga* trädet. Att stänga ett träd gör man genom att stänga trädets grenar.

Definition 16 *En gren är stängd när ϕ och $\neg\phi$, eller $\perp$ existerar i grenen.*

Expansionen av tableauträdet sker genom att applicera olika regler på trädets grenar. En av dessa regler är redan definierad och det är regeln för elimination av dubbelnegation, de andra är dock inte definierade och visas nedan i tabell 2.2 och 2.3.

Tabell 2.2: *Reglerna för expansion av α- och β-formler.*

Konjunktiv			Disjunktiv		
α	α_1	α_2	β	β_1	β_2
$x \wedge y$	x	y	$\neg(x \wedge y)$	$\neg x$	$\neg y$
$\neg(x \vee y)$	$\neg x$	$\neg y$	$x \vee y$	x	y
$\neg(x \rightarrow y)$	x	$\neg y$	$x \rightarrow y$	$\neg x$	y

Dessa regler låter oss omvandla ett tableauträd med logiska formler som noder till ett nytt tableauträd, expanderat enligt reglerna. Arbetsättet är enligt följande: välj en gren θ och en formel ϕ som inte är en literal och utför sedan:

- Om ϕ är $\neg\neg x$ så lägg till noden x till i θ.
- Om ϕ är en α-formel skall α_1 och α_2 läggas till som noder i θ.
- Om ϕ är en β-formel skall det skapas två nya barnnoder till den sista noden med β_1 och β_2, det vill säga två nya grenar.

På samma sätt som regler delas upp i konjunktiva (α-formler) och disjunktiva (β-formler) regler, kan det för predikatlogik göras ytterligare en uppdelning mellan allkvantifierande och existenskvantifierande formler γ och δ.

Tabell 2.3: *Regler för utveckling av γ-* och δ-formler.

Universell		Existensiell	
γ	$\gamma(t)$	δ	$\delta(t)$
$\forall x.\Phi$	$\Phi\{t/x\}$	$\exists x.\Phi$	$\Phi\{t/x\}$
$\neg\exists x.\Phi$	$\neg\Phi\{t/x\}$	$\neg\forall x.\Phi$	$\neg\Phi\{t/x\}$

Tabellen ovan visar hur γ-formler och δ-formler används. Det är reglerna för γ-formler som gör att de flesta implementationer av tableau är långsammare än implementationer av resolution. I tableau räcker det för alla andra regler att appliceras en gång per formel i trädet. Dessa regler kan dock behöva användas flera gånger på samma formel och det går inte att i förväg att veta exakt hur många gånger detta kommer att behöva ske.

När de historiskt sett första implementeringarna av tableau gjordes användes regeln en gång för varje stängd term [5], men eftersom detta antal kan bli oändligt [5] är det ingen effektiv strategi. Ett försök att lösa problemet som används i många nyare implementeringar är användandet av unifiering och ersättandet av reglerna för γ-formlerna med en regel:

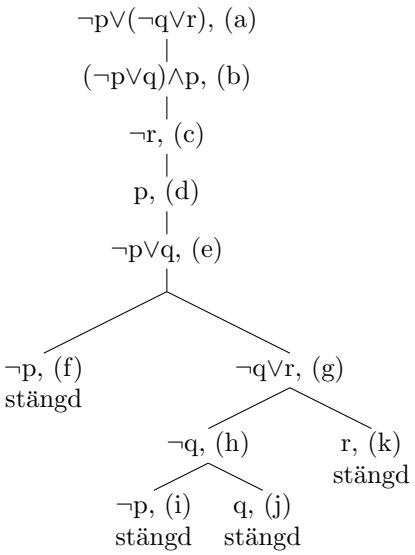
$$\frac{\gamma}{\gamma(x)}$$

Denna regel gör en substution på samma vis som förut men istället för att införa $\gamma(t)$ där t är en konstant, införs $\gamma(x)$ med den nya fria variabeln x. Värdet på x sätts senare med hjälp av unifiering till något som har möjlighet att stänga en gren.

För att bättre visa hur tableau jobbar sig fram för att finna en lösning följer ett exempel. Uppgiften för teorembevisaren är att bevisa formel 2.21 från stycket ovan om resolution. Det första steget är att anta den negerade slutsatsen, $\neg r$, och lägga den till premisserna med hjälp av en konjunktion:

$$(\neg p \vee (\neg q \vee r)) \wedge (\neg p \vee q) \wedge p \wedge \neg r$$

Figur 2.9 visar den följande expansionen av tableuträdet:



Figur 2.9: *Exempel på tableauträd*

Noderna (d) och (e) bildas från konjunktionen i (b) som bryts upp genom applicerande av reglerna för α-formler från tabell 2.2. Då det ännu inte finns någon motsägelse i trädet, börjar tableau använda reglerna för β-formler från samma tabell för att expandera det till flera grenar. Den vänstra grenen går att stänga genom motsägelsen (f) och (d). Den högra grenen måste dock expanderas i ytterligare grenar innan trädet kan stängas i sin helhet. Detta kan göras med hjälp av motsägelser i (d) och (i), (h) och (j) samt (c) och (k).

2.6.3 Narrowing

Narrowing är en samling beräkningsstrategier för att reducera okända termer i omskrivingssystem[10]. Narrowing används idag som en del av kärnan i så kallade funktionella logikprogrammeringsspråk, exempelvis Curry[11] och $\mathcal{TOY}$[4] m.fl.[10]. Lite informellt kan narrowing sägas unifiera vänsterledet i en omskrivningsrege l med en term för att sedan applicera regeln på den instansierade termen. Vad narrowing egentligen är, kan lättast åskådliggöras med ett exempel på en sökning efter tilldelningar till okända delar av en ekvation i ett rekursivt definierat talsystem.

[10]Eng. Rewrite system.

Betrakta ett talsystem likt de naturliga talen. Det finns ett tal 0. För varje tal n finns ett efterföljande tal $s(n)$. I ett sådant system blir exempelvis talet 3 definierat rekursivt genom $s(s(s(0)))$. Likhet följer genom omskrivningsreglerna:

$$\begin{aligned} &0 = 0 \rightarrow sant \\ &0 = s(x) \rightarrow falskt \\ &s(x) = 0 \rightarrow falskt \\ &s(x) = s(y) \rightarrow x = y \end{aligned}$$

I denna miljö är det möjligt att införa operatorn för addition rekursivt genom:

$$\begin{aligned} &0 + n \rightarrow n \\ &m + s(n) \rightarrow s(m + n) \end{aligned}$$

Vi kan i detta systemet försöka lösa en ekvationen nedan med en serie omskrivningar:

$$s(0)+?_0 = s(s(0))$$

Man kan börja med att applicera regel 2 för addition $s(0)+?_0 \rightarrow s(0+?_0)$ på vänsterledet för att få:

$$s(0+?_0) = s(s(0))$$

Detta kallar man för reduceringssteg. Vidare kan man använda likhetsregel 4 $s(0+?_0) = s(s(0)) \rightarrow 0+?_0 = s(0)$ och sedan applicera den första regeln för addition $0+?_0 \rightarrow ?_0$ på vänsterledet. Kvar får vi då ekvationen:

$$?_0 = s(0)$$

Här är det möjligt att instansiera $?_0 \rightarrow s(?_1)$, vilket ger:

$$s(?_1) = s(0)$$

Ytterligare en likhetsreduktion och den sista okända termen kan tilldelas $?_1 \rightarrow 0$ för att lösa ekvationen. Lösningen av ekvationen ovan presenterade en möjlig kombination av de steg som krävdes för en lösning. Hur många omskrivningsregler som egentligen användes innan lösningen kunde finnas beror på vilken narrowingstrategi som används. En av de (sämre) strategierna är att försöka unifiera varje omskrivningsregel med varje oinstansierad term vid varje narrowingsteg, vilket resulterar i en uttömmande sökning. De vanligaste strategierna man använder idag kan samlas under namnen lat eller nödvändig narrowing[11] vilka försöker att enbart göra steg som är oundvikliga. Dessa och andra behandlas översiktligt tillsammans med deras egenskaper i [1]. Något mer formellt kan man se narrowing som ett verktyg som i omskrivningssystem kan lösa ekvationer genom att beräkna unifierare med avseende på dess ekvationssteori. Den försöker uppnå *sant* och skriver om termen tills detta uppnåtts eller termen är så avsmalnad[12] att alla regler evaluerar till $falskt$. Nödvändig narrowing visar sig ha goda resultat för system baserade på definitionsträd[13][2].

Inspirationen till att konstruera en bevisletare baserad på narrowing kommer från [17], där en bevisverifierare appliceras på en lat nödvändig narrowingimplementation för att hitta bevis av högre ordningen. Implementationen gjordes i Haskell och utnyttjar en del finurliga tricks för att undvika att generera egna definitionsträd genom insikt i hur GHC[14] fungerar internt.

[11] Eng. Needed narrowing.
[12] Eng. Fully narrowed.
[13] Eng. Definitional tree.
[14] The Glasgow Haskell Compiler.

Latheten kommer väl till pass om narrowing skulle ställas inför en situation likt:

$$?_0 \wedge ?_1$$

Om den här försöker instansiera $?_0 \rightarrow falskt$ behöver vi inte ens evaluera resten eftersom $falskt \wedge ?_1 \rightarrow falskt$ oavsett vad $?_1$ antar. Detta gör att den delen av lösningsträdet kan kapas bort och därmed minska sökrymden. Detta gör att sökningen i praktiken blir effektivare än en uttömmande sökning samtidigt som alla korrekta lösningar kan beräknas.

Vi kan omformulera problemet att hitta bevis för en följd av formler i termer av omskrivning genom att låta slutsatsformeln vara fix och bevisstrukuren det okända objektet. I detta fall kan vi lätt skära bort delar av sökrymden för varje introduktionsregel som inte kan appliceras på den yttersta konnektiven. En bevisverifierare av denna formen kan till exempel se ut något liknande:

```
check :: Proof -> Formula -> Bool
```

Vi låter formeln vandra uppåt rekursivt genom verifieraren och manipulerar enbart den baserat på de inferensregler som narrowing ger oss. Nedan följer ett exempel på hur `check` kan implementeras:

```
check p f = case p of
    AndI a b -> case f of
        And x y -> (check a x) && (check b y)
        _ -> False
    OrI1 a -> case f of
        Or x y  -> check a x
        _ -> False
    [...]
```

Idén att använda narrowing för bevisletning är dock inte helt ny, det upptäcktes nämligen för att användas till detta ändamål i [26], fast på ett annat sätt än det som presenterats här. För att en narrowingstrategi skall bli effektiv[15], sund och fullständig krävs oftast att man gör begränsningar på hur sökningen sker och vilka omskrivningar man tillåter.

[15] Minimalt antal steg som inte leder till en lösning.

Kapitel 3

Implementation

Detta kapitel beskriver projektets upplägg och vilka val som gjorts under implementeringen. Först kommer en översiktlig presentation av hela projektet med ett moduldiagram. Sedan beskrivs hur vi valt att representera predikatlogik och naturlig deduktion som abstrakta datatyper i Haskell. Avslutningsvis beskrivs hur bevis verifieras och de val som gjorts under implementeringen av teorembevisarna.

3.1 Översiktligt om implementation

Detta avsnitt kommer gå igenom de olika modulerna i implementationen, hur de samarbetar med varandra samt viktiga datatyper som används i stor utsträckning. De moduler som nämns nedan kommer förklaras senare i detta kapitel.

Hela applikationen byggs på grammatiken, de datatyper som genererats av BNFC från den abstrakta syntaxen. Dessa datatyper inkluderar den interna representationen för formler, teorem, bevis och referenser till inferensregler. Det finns tre representationer för bevis i applikationen: en generell representation som genereras av BNFC och två andra som är mer specialiserade, en för bevis på liststruktur samt en för trädstruktur. Ett generellt bevis fås när text tolkas av BNFC. Dessa bryts sedan ner så att lämpliga parametrar kan skickas till de olika teorembevisarna alternativt bevisverifieraren. Det består av en lista med premisser, ett antal inferensregler samt en slutsats.

Ett bevis på liststruktur innehåller en lista med triplar som var och en består av ett referensnummer, en formel och en applicerad inferensregel. Strukturen är starkt influerad av exemplen i [13]. I tabell 3.1 ses ett enkelt exempel av ett sådant bevis:

Tabell 3.1: *Bevis på listform*

1	p	Premiss
2	$\neg\neg(q \wedge r)$	Premiss
3	$\neg\neg p$	$\neg\neg_i\, 1$
4	$q \wedge r$	$\neg\neg_e\, 2$
5	r	$\wedge_{e2}\, 4$
6	$\neg\neg p \wedge r$	$\wedge_i\, 3, 5$

Ett bevis på trädstruktur är rekursivt uppbyggt och varje nivå är ett bevis av en formel givet en inferensregel samt ytterligare ett bevis för varje delträd som inferensregeln behöver för att derivera den givna formeln. Figur 3.1 visar samma bevis som i tabell 3.1, men på trädstruktur:

$$\dfrac{\dfrac{p}{\neg\neg p}\,\neg\neg_i \qquad \dfrac{\dfrac{\neg\neg(q \wedge r)}{q \wedge r}\,\neg\neg_e}{r}\,\wedge_e 2}{\neg\neg p \wedge r}\,\wedge_i$$

Figur 3.1: *Bevis på trädform*

Det har implementerats en modul som kan konvertera en godtycklig formel till en av två normalformer. Antingen konjunktiv- eller disjunktiv normalform. Denna modul används både av resolution och tableau. I implementationen av denna modul så byter de standardiserade variablerna namn till $\{x_0, x_1, ...\}$ och skolemkonstanterna och skolemfunktionerna heter $\{f_0, f_1, ...\}$.

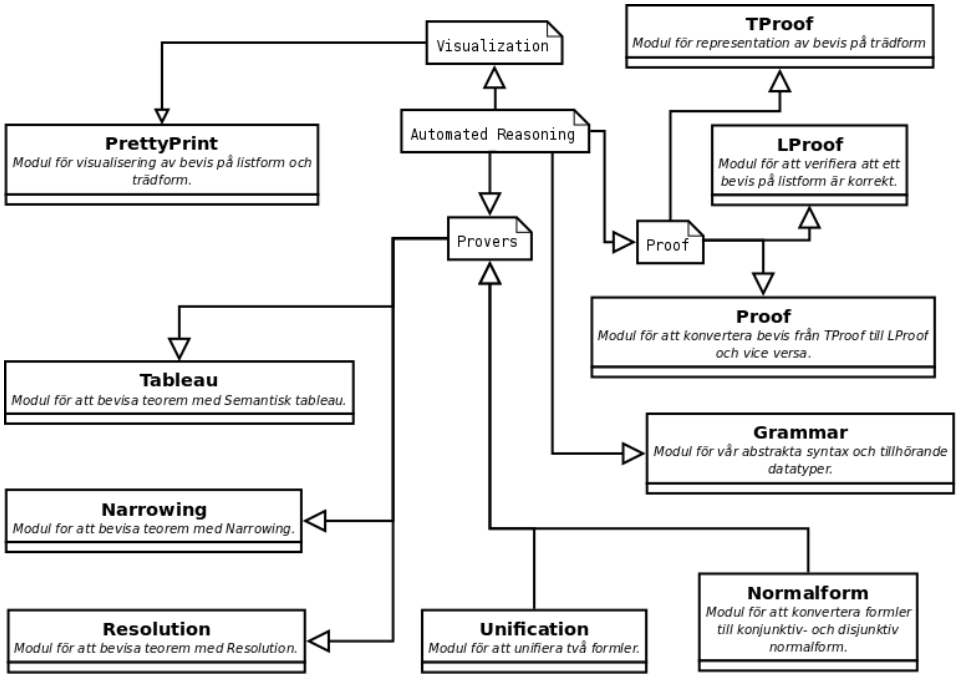
Ytterligare en modul som flitigt används av de två ovannämnda teorembevisarna är den för unifiering. Denna modul behandlar endast literaler, eftersom det inte finns behov i applikationen av att unifiera formler på en högre nivå.

Resolution genererar ett generellt bevis medan tableau och narrowing genererar bevis på trädstruktur.

Bevisverifieraren tar antingen in ett bevis i den generella strukturen eller ett bevis i liststrukturen. Den kan även ta in ett bevis på den generella strukturen och konvertera detta till ett bevis på liststruktur.

3.1.1 Moduldiagram

För att lättare se relationerna mellan delarna i applikationen visas ett moduldiagram i figur 3.2. Som detta indikerar har den tre huvuddelar, nämligen en för bevisverifiering, en för teorembevisare och en för grafisk representation av bevis.



Figur 3.2: *Moduldiagram för projektet.*

3.2 Abstrakt syntax

För detta projekt har det tagits fram en abstrakt syntax med hjälp av BNFC[20], som också bygger de datatyper för formler, och till viss utsträckning bevis och inferensregler som används i applikationen. Detta innebär att bevis och teorem skrivna på denna syntax kan återanvändas om och om igen. Detta har visat sig vara en mycket värdefull egenskap vid testning under utvecklingsfasen. Det innebär också att användare inte behöver sätta sig in i den interna representationen av bevis utan istället kan skriva dem på den abstrakta syntaxen. Bevisen och teoremen blir mer lättläsliga, går snabbare att skriva och blir mer intuitionistiska eftersom syntaxen påminner om den notation som används i avsnittet om predikatlogik.

I det här avsnittet visas och förklaras de grammatiska regler som ingår i den abstrakta syntaxen. Förklaringen kommer börja på en hög nivå för att ge en bra överblick och sedan arbeta sig nedåt. Det kommer sedan visas konkreta exempel på hur bevis och teorem skrivs i denna syntax och även hur ett syntaxträd[1] av ett bevis ser ut. För att ta del av detta avsnitt så bör läsaren tagit del av avsnitten om predikatlogik och naturlig deduktion.

3.2.1 Bevis

Ett bevis är en lista av premisser, ett antal inferensregler applicerade på premisserna samt en slutsats. Nedan visas definitionen av ett bevis på den abstrakta syntaxen:

$$PProof.\,Proof\ ::=\ [Formula]\text{"}\ \texttt{==>}\ \text{"}\,Rule\text{"}\,\texttt{|-}\ \text{"}\,Formula\,;$$

Ovan representeras premisserna av en lista med formler separerad med semikolon. Det finns naturligtvis en regel som säger beskriver att det skall vara semikolon som separator, men eftersom den är trivial tas den inte uppe i detta dokument. Inferensreglerna representeras av datatypen, "Rule". Notationen säger att detta element inte är en lista, till skillnad från premisserna, vilket tyder på att detta är en rekursiv datatyp. En "Rule" innehåller en inferensregel och ännu en "Rule". Bevisets slutsats representeras av en formel. Tecknen som separerar dessa element är till för BNFC vid översättningen till de interna datatyperna och skickas inte vidare till applikationen.

Denna datatyp är även den som används för att representera ett teorem - man utelämnar bara inferensreglerna. Detta på grund av en begränsning i BNFC som medför att man inte kan ha flera s.k. "entry points" i grammatiken. Detta innebär i korthet att det inte får finnas tvetydigheter i vad som ska tolkas och översättas. Skickar man in ett teorem till bevisverifieraren, så kommer den att tolka det som ett ogiltigt bevis på grund av avsaknaden av inferensregler. Skickar man in ett bevis till någon av våra teoremsbevisare, kommer den att förkasta inferensreglerna och försöka bevisa det på egen hand.

3.2.2 Formler

Nedan visas de definitioner för formlerna som består av en binär operator som binder ihop två formler:

$$Impl.\,Formula\ ::=\ Formula\text{"}\,\texttt{->}\ \text{"}\,Formula1\,;$$

$$Or.\,Formula1\ ::=\ Formula1\text{"}\,\texttt{|}\,\text{"}\,Formula2\,;$$

$$And.\,Formula2\ ::=\ Formula2\text{"}\,\&\,\text{"}\,Formula3\,;$$

[1] Eng. Parse tree.

Definitionerna ovan säger att en formel kan vara två formler som binds samman av antingen en implikation, konjunktion eller disjunktion. Siffrorna efter ordet "Formula" anger hur hårt operatorn binder. Ju lösare bindning, desto lägre nummer (inget nummer kan tolkas som siffran 0).

Här följer de definitioner som beskriver kvantifieringar över en viss formel:

$$All.\,Formula3 \;::=\; \text{"!"}\;\text{"["}\;TIdent\;\text{"]"}\;\text{" :"}\;Formula3\,;$$

$$Exist.\,Formula3 \;::=\; \text{"?"}\;\text{"["}\;TIdent\;\text{"]"}\;\text{" :"}\;Formula3\,;$$

Definitionerna säger att en formel kan vara en allkvantifiering eller en existenskvantifiering av en given variabel över en formel. Observera att hakparenteser inom citationstecken och "vanliga" hakparenteser inte betyder samma sak. Vanliga hakparenteser tyder på att elementet är en lista, medan hakparenteser inom citationstecken är till för BNFC vid översättningen till de interna datatyperna.

Nedan följer definitionen av en negation:

$$Neg.\quad Formula3 \;::=\; \text{" }\sim\text{ "}\;Formula3\,;$$

Definitionen säger helt enkelt att en formel kan vara en negation av en annan formel.

Till sist visas definitionerna av de olika atomerna i datatypen formel:

$$Pred.\,Formula4 \;::=\; PIdent\;\text{"("}\;[Term]\;\text{")"}\,;$$

$$Bottom.\,Formula4 \;::=\; \text{" \_|\_"}\,;$$

$$Term.\,Formula4 \;::=\; Term\,;$$

Definitionerna ovan säger att en formel kan vara ett predikat, $\bot$ eller en term. Ett predikat består av en identifierare samt en kommaseparerad lista med termer som argument, omslutna av ett par parenteser.

3.2.3 Termer

Det finns tre olika varianter av termer: variabler, funktioner och konstanter. Dessa visas nedan:

$$Var.\,Term \;::=\; TIdent\,;$$

$$Const.\,Term \;::=\; TIdent\;\text{"("}\;\text{")"}\,;$$

$$Func.\,Term \;::=\; TIdent\;\text{"("}\;[Term]\;\text{")"}\,;$$

En variabel är en enkel identifierare, en konstant är en funktion som inte tar emot några argument och en funktion är en identifierare med en kommaseparerad lista av termer som argument, omslutna av ett par parenteser.

3.2.4 Identifierare

Det finns två olika typer av identifierare i den abstrakta syntaxen:

$$token\,TIdent\,(\,lower\,(\,letter\,|\,digit\,|\,'\_'\,)*\,)\,;$$

$$token\,PIdent\,(\,upper\,(\,letter\,|\,digit\,|\,'\_'\,)*\,)\,;$$

En "TIdent" är en godtycklig sekvens av små bokstäver, siffror och understreck. Denna identifierare används till variabler, funktioner, konstanter och kvantifieringar. En "PIdent" är en godtycklig sekvens av stora bokstäver, siffror och understreck. Denna identifierare används endast till predikat. Två begränsningar som BNFC lägger på de båda är att identifierarna måste börja på en bokstav, samt att sekvensen av tecken måste vara minst ett tecken lång.

3.2.5 Inferensregler

Det finns fyra olika typer av inferensregler: antaganden, introduktionsregler, eliminationsregler samt deriverade regler som kan liknas vid makron. Det finns ännu en regel, den tomma regeln, men eftersom den endast används internt i vår applikation och därför är osynlig för användaren kommer den inte tas upp i detta dokument.

3.2.6 Antaganden

Ett antagande är egentligen inte en inferensregel så mycket som en märkning av ett antagande som görs i ett bevis. Regeln finns där för att flagga att speciella förhållanden gäller tills man går ur scopet som definierar antagandet. Nedan visas syntaxen för ett antagande:

$$ARule. \quad Rule ::= "\{" Formula ";" Rule "\}" Rule;$$

Antagandets scope definieras av måsvingarna och "Formula" är antagandet som introduceras. Regeln innanför måsvingarna innehåller de inferensregler som görs i antagandets scope och regeln utanför innehåller resten av inferensreglerna i beviset.

3.2.7 Introduktionsregler

Nedan beskrivs syntaxen för introduktionsreglerna tagna från naturlig deduktion. Det första ordet inom citationstecken är identifieraren för regeln, medan de efterföljande siffrorna är referenser till formler i beviset.

$$AndI.IntroRule ::= "andI" Integer "," Integer;$$

Detta är syntaxen för konjunktionsintroduktion, där de två siffrorna refererar formler i beviset som skall kopplas samman med en konjunktion.

$$OrI1.IntroRule ::= "orI1" Integer "," Formula;$$

$$OrI2.IntroRule ::= "orI2" Integer "," Formula;$$

Ovan visas syntaxen för de två varianterna av disjunktionsintroduktion. Siffran refererar till den ena av två formler som skall bindas samman med en disjunktion, "Formula" är den andra. Infereras den första regeln så hamnar den givna formeln på höger sida, infereras den andra så hamnar den på vänster sida.

$$NotI.IntroRule ::= "notI" Integer "-" Integer;$$

Detta är definitionen för negationsintroduktion. De två siffrorna refererar till början och slutet på ett antagande som till slut deriverar $\perp$.

$$NotNotI.IntroRule ::= "notnotI" Integer;$$

Ovan visas syntaxen för introduktion av dubbel negation. Siffran refererar till formeln som skall bli dubbelt negerad.

$$ImpI.IntroRule ::= "implI" Integer "-" Integer;$$

Syntaxen för implikationsintroducering visas ovan. De två siffrorna refererar till ett interval av formler, som är en derivering av en formel från en annan.

$$ForAllI.IntroRule ::= "!" "[" TIdent "/" TIdent "]" "I" Integer "-" Integer;$$

Detta är syntaxen för allkvantifierarintroduktion, den är en av de mest komplicerade i den abstrakta syntaxen. Den första identifieraren är den helt nya variabeln som skall ersättas med den andra identifieraren, vilket är kvantifieringsvariabeln. Siffrorna anger intervallet för den nya variabelns scope.

$$ExistI.IntroRule ::= \text{"?"}\ \text{"["}\ Term\ \text{"/"}\ TIdent\ \text{"]"}\ \text{"}I\text{"}\ Integer;$$

Ovan visas syntaxen för existenskvantierarintroduktion. Siffran refererar till den formel introduktionen införs på.

3.2.8 Eliminationsregler

Nedan introduceras syntaxen för eliminationsreglerna i den abstrakta syntaxen. Samma generella regler som gäller för introduktionsregler, gäller även för eliminationsreglerna.

$$AndE1.ElimRule ::= \text{"}andE1\text{"}\ Integer;$$

$$AndE2.ElimRule ::= \text{"}andE2\text{"}\ Integer;$$

Ovan visas de regler som infererar konjunktionseliminering. Siffran refererar till en formel i beviset med en konjunktion på högsta nivå.

$$OrE.ElimRule ::= \text{"}orE\text{"}\ Integer\text{","}\ Integer\ \text{"}-\text{"}\ Integer\text{","}\ Integer\ \text{"}-\text{"}\ Integer;$$

Syntaxen för disjunktionseliminering är relativt komplex jämfört med de andra inferensreglerna. Den första siffran refererar till en formel med en disjunktion på högsta nivån. De två återstående paren av siffror refererar till varsitt intervall av formler som båda är antaganden. De startar med den högra respektive vänstra delformeln av disjunktionen som refereras av första siffran och deriverar till slut samma formel.

$$NotE.ElimRule ::= \text{"}notE\text{"}\ Integer\text{","}\ Integer;$$

Ovan visas syntaxen för negationselimination, där de två siffrorna refererar formler som är varandras logiska motsats.

$$NotNotE.ElimRule ::= \text{"}notnotE\text{"}\ Integer;$$

Detta är syntaxen för dubbel negationselimination. Siffran refererar till en formel med en dubbel negation på högsta nivån.

$$ImpE.ElimRule ::= \text{"}implE\text{"}\ Integer\text{","}\ Integer;$$

Syntaxen för implikationselimination visas ovan. Den första siffran refererar till en formel med en implikation på högsta nivå och den andra refererar till en formel som är logiskt ekvivalent till vänstersidan av implikationen.

$$BottomE.ElimRule ::= \text{"}bottomE\text{"}\ Integer\text{","}\ Formula;$$

Ovan visas syntaxen för eliminering av $\bot$. Siffran refererar till $\bot$ och formeln är vad man vill introducera.

$$ForAllE.ElimRule ::= \text{"!"}\ \text{"}E\text{"}\ Term\ Integer;$$

Detta är syntaxen för allkvantifierarelimination. Siffran refererar till en formel med en allkvantifiering på högsta nivån och termen är vad man vill ersätta den kvantifierade variabeln med.

$$ExistE.ElimRule ::= \text{"?"}\ \text{"}E\text{"}\ Integer\text{","}\ Integer\ \text{"}-\text{"}\ Integer;$$

Ovan visas syntaxen för existenskvantifierarelimination. Den första siffran refererar till en formel med en existenskvantifiering på högsta nivån. Den andra och tredje siffran refererar till ett interval som är ett scope för ett antagande av kvantifieringen.

3.2.9 Deriverade regler

Inferensregler som tas upp nedan kan härledas med hjälp av ett antal andra regler, men eftersom dessa används relativt ofta har valet tagits att inkludera dessa i den abstrakta syntaxen. Användandet av deriverade regler kan bl.a. korta ner bevis och på så sätt även underlätta läsbarheten.

$$LEM.\,DerivedRule\ ::=\ "\,LEM"\ Formula\,;$$

Detta är syntaxen för den deriverade regeln lagen om det uteslutna tredje[2]. Om man t.ex. anger p som formel, så kommer regeln införa $p \vee \neg p$ i kontexten.

$$RAA.\,DerivedRule\ ::=\ "\,RAA"\ Integer\ -"\ Integer\,;$$

Ovan visas syntaxen för motsägelsebevis. Siffrorna refererar till ett interval av formler som är ett antagande och slutar i $\perp$.

$$MT.\,DerivedRule\ ::=\ "\,MT"\ Integer"\,,"\ Integer\,;$$

Syntaxen för regeln modus tollens visas ovan. Den första siffran refererar till en formel med en implikation på högsta nivån och den andra siffran refererar till en formel som är negationen av implikationens högra sida.

$$CNF.\,DerivedRule\ ::=\ "\,CNF"\ Integer\,;$$

Detta är syntaxen för att åberopa konvertering av en refererad formel till konjunktiv normalform.

$$RES.\,DerivedRule\ ::=\ "\,RES"\ Integer"\,,"\ Integer\,;$$

Ovan visas syntaxen för att applicera resolutionregeln på två refererade formler, vilka refereras av de två siffrorna.

3.2.10 Exempel och syntaxträd

I tabell 3.2.10 visas två exempel som är skriva på den abstrakta syntaxen, det första exemplet är ett bevis och det andra är ett teorem.

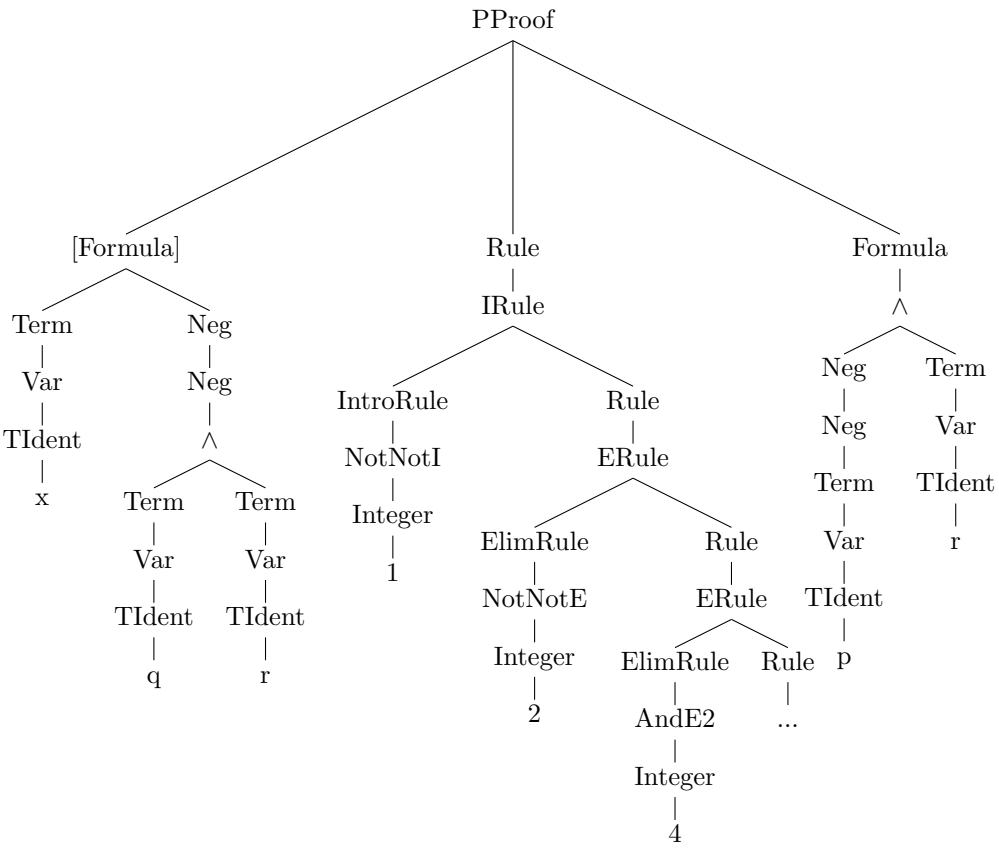
Tabell 3.2: *Till vänster visas ett bevis och till höger ett teorem.*

```
1  p;
2  ~~(q & r)
3  ==>
4  notnotI 1;
5  notnotE 2;
6  andE2 4;
7  andI 3,5;
8  |-
9  ~~p & r
```

```
1  p -> q
2  ==>
3  |-
4  ~p | q
```

För att ge läsaren en uppfattning om hur ett bevis ser ut efter översättning från den abstrakta syntaxen när det kommer till själva applikationen, visas i figur 3.3 ett sådant parseträd.

[2]Eng. Law of the excluded middle.



Figur 3.3: *Parseträd för beviset i tabell 3.2.10*

Delträdet med inferensregler har kortats av för att syntaxträdet ska få plats på sidan. Det finns dock tillräckligt mycket information kvar för att man ska kunna förstå strukturen.

3.3 Bevisverifiering

Detta avsnitt kommer att förklara bevisverifierarens arbetsgång samt hur det har säkerställts att den verkligen gör korrekta bedömningar av bevis. Det kommer också att presenteras metoder för visualisering av bevis.

3.3.1 Bevisverifieraren

Denna modul tar in ett bevis och avgör om det är korrekt eller ej. Beviset innehåller en lista med premisser, ett antal inferensregler samt en slutsats. Bevisverifieraren applicerar helt enkelt de angivna inferensreglerna och ser om den själv kommer fram till en formel som matchar den angivna slutsatsen.

Detta ger dock inte alltid mängden information som önskas. Därför kan också bevisverifieraren endast utföra de nödvändiga beräkningarna för beviset, utan att kontrollera om det är giltigt eller ej. Den applicerar då de angivna inferensreglerna som vanligt, men utför inte den sista kontrollen. Om någon inferensregel applicerats på ett felaktigt sätt, kommer dock endast ett felmeddelande att returneras. Vid visualisering av detta okontrollerade bevis kan man följa de beräkningar som gjorts och se var beviset brister.

Som det kort nämndes ovan så kan man få ytterligare information, vid kontroll av ogiltiga bevis, om någon inferensregeln applicerats fel. Exempel på sådana fall kan vara

att en konjunktionseliminering refererar till en formel med en disjunktion på högsta nivå eller att en allkvantifieringsintroduktion inte sker på en helt ny variabel.

3.3.2 Visualisering av bevis

Att kunna visualisera bevis är av stort intresse, ur användarens perspektiv men också för utvecklarna. Denna funktion är till stor hjälp om exempelvis bevisverifieraren dömer ett bevis som ogiltigt och man vill se var det gick fel (givet att alla inferensregler applicerats korrekt). Det kan också vara av intresse att se hur de olika implementerade teorembevisarna löser ett teorem. Det beslutades tidigt i utvecklingsfasen att visualisering av bevis skulle kunna ske på två olika sätt. Den första är en textbaserad version som tar ett bevis i linjär form, vilket är en tabell där varje rad är ett steg i beviset. Den andra visualiseringsmetoden ritar bevis på trädform med hjälp av LaTeXsom slutsteg vilket bygger upp ett naturligt deduktionsträd. Exempelbevis visualiserade med dessa metoder visas i tabell 3.3 respektive figur 3.4.

Tabell 3.3: *Resolutionsbevis på tabellform*

1.	p | q | r	Premise
2.	˜(p | q | r)	Assumption
3.	˜p & ˜q & ˜r	CNF 2
4.	˜p & ˜q	AndE1 3
5.	˜r	AndE2 3
6.	˜p	AndE1 4
7.	˜q	AndE2 4
8.	p | q	RES 1 5
9.	q	RES 8 6
10.	_|_	RES 9 7
11.	p | q | r	RAA 2 10

$$\dfrac{\dfrac{p \vee q \vee r \qquad \dfrac{[\neg(p \vee q \vee r)]}{\neg p \wedge \neg q \wedge \neg r}\,CNF}{\dfrac{p \vee q}{\neg r}\,\wedge e_2}\,RES \qquad \dfrac{\dfrac{[\neg(p \vee q \vee r)]}{\neg p \wedge \neg q \wedge \neg r}\,CNF}{\dfrac{\neg p \wedge \neg q}{\neg p}\,\wedge e_1}\,\wedge e_1}{q}\,RES \qquad \dfrac{\dfrac{[\neg(p \vee q \vee r)]}{\neg p \wedge \neg q \wedge \neg r}\,CNF}{\dfrac{\neg p \wedge \neg q}{\neg q}\,\wedge e_2}\,\wedge e_1}{\dfrac{\bot}{p \vee q \vee r}\,RAA}\,RES$$

Figur 3.4: *Resolutionsbevis på trädform*

3.4 Implementation av teorembevisare

Detta avsnitt behandlar hur de olika metoderna för teorembevisning tolkats till konkreta implementationer och vilka eventuella optimeringar som implementerats. Det finns även exempel i vart och ett av delavsnitten som visar hur de olika teorembevisarna skapar bevis för teorem.

3.4.1 Resolution

Denna teorembevisare bygger upp ett bevis i linjär form. Misslyckas den att konstruera ett komplett bevis för ett givet teorem, kommer det returneras ett bevis med en tom lista av inferensregler.

Hjärtat i teorembevisaren är implementationen av resolutionsregeln. Funktionen tar in två klausuler som kopplas samman av en disjunktion och konverterar sedan till en lista av literaler. Listan sorteras i lexikografisk ordning. Detta medför att komplementära literaler, t.ex. p och $\neg p$, hamnar bredvid varandra, vilket underlättar för nästa steg där först dubbletter tas bort och sedan komplementära literaler. Eftersom listan är sorterad kommer sökningen i värsta fall vara $\mathcal{O}(n^2)$. För att avgöra om två literaler är kopior av varandra används likhetsoperatorn och för att avgöra om ett par literaler är komplementära används unifiering. De kvarvarande literalerna konverteras sedan tillbaka till en disjunktiv formel och returneras.

För att konstruera ett bevis för ett teorem så läggs först den negerade slutsatsen till som ett antagande. Därefter konverteras alla formler som inte redan är i konjunktiv normalform och konjunktioner bryts upp så att varje klausul hamnar på en varsin rad. I tabell 3.4 ses ett exempel som visar förberedelserna som görs inför konstruktionen av ett bevis. Notera att det inte är komplett.

Tabell 3.4: *Förberedelser inför bevis med resolution*

1	$p \vee (q \wedge r)$	Premiss
2	$q \wedge r$	Premiss
3	$\neg r$	Antagande
4	$(p \vee q) \wedge (p \vee r)$	CNF 1
5	$p \vee q$	$\wedge_{e1}$ 4
6	$p \vee r$	$\wedge_{e2}$ 4
7	q	$\wedge_{e1}$ 2
8	r	$\wedge_{e2}$ 2

I tabell 3.4 konverteras formler, som inte redan är i korrekt form, till konjunktiv normalform och sedan bryts eventuella konjunktioner upp.

Samtliga formler i beviset som är klausuler läggs till i en separat lista, en kunskapsbas[3] (KB). Det är dessa resolutionregeln appliceras på, och förhoppningsvis deriveras en motsägelse.

Nästa steg matchar varje formel mot resten av formlerna i KB och sparar resultatet av appliceringar av resolutionregeln mellan varje par. Finns redan några av resultaten i KB så kommer dessa filteras bort för att hålla resultatlistan så liten som möjligt. Resultatlistan itereras sedan igenom varpå elementet med minst antal literaler väljs ut. Denna optimering kan liknas vid "Unit Preference" som tas upp i [25] och innebär att resolutionregeln endast appliceras där minst en av formlerna är en enkel literal. På detta sätt produceras endast resultatklausuler som är kortare än den längsta inparametern, vilket dramatiskt förbättrar prestandan vid teorembevisning. Om det utvalda resultatet saknar literaler leder detta till $\perp$ som läggs till i KB och en motsägelse har funnits. Om resultatet däremot innehåller minst en literal, läggs det till i KB. Resultatlistan kommer också utökas med resultat från applicering av resolutionregeln mellan det nya resultatet och resten av formlerna i KB.

Algoritmen avbryts när antingen $\perp$ deriverats från en applicering av resolutionregeln eller när listan med resultat är tom, vilket innebär att alla möjliga kombinationer av literaler finns i KB.

[3] Eng. Knowledge base.

För att skapa ett mer lättläst och minimalt bevis kommer baklängesstegning att ske. Denna behåller endast de regler som användes för att derivera slutsatsen. Detta görs främst för att visualiseringen av ett sådant bevis blir lättare att förstå för användaren.

I tabell 3.5 visas ett bevis utfört med naturlig deduktion följt av samma bevis utfört med resolution som visas i tabell 3.6.

Tabell 3.5: *Bevis utfört med naturlig deduktion*

1.	$\forall x.(P(x) \rightarrow Q(x))$	Premiss
2.	$P(x_0)$	Premiss
3.	$P(x_0) \rightarrow Q(x_0)$	$\forall x_e$ 1
4.	$Q(x_0)$	$\rightarrow_e$ 3,2
5.	$\forall x.Q(x)$	$\forall x_i$ 2-4

Tabell 3.6: *Bevis utfört med resolution*

1.	$\forall x.(P(x) \rightarrow Q(x))$	Premiss
2.	$P(x_0)$	Premiss
3.	$\neg\forall x.Q(x)$	Antagande
4.	$\neg P(x_0) \vee Q(x_0)$	CNF 1
5.	$\neg Q(f_0)$	CNF 3
6.	$Q(x_0)$	Resolution 2,4
7.	$\perp$	Resolution 5,6
8.	$\forall x.Q(x)$	RAA 3-8

I tabell 3.6 syns det t.ex. att unifiering används då resolutionregeln appliceras på raderna 5 och 6. Formlerna $\neg Q(f_0)$ och $Q(x_0)$ är logiskt ekvivalenta givet att den allkvantifierade variabeln x_0 har värdet f_0, vilket i detta fallet är en skolemkonstant.

3.4.2 Semantisk tableau

Algoritmen för semantisk tableau kan brytas ner i ett antal steg. Indata (premisserna och den negerade slutsatsen) konverteras inledningsvis till disjunktiv normalform (som förklarats i avsnitt 2.3). Detta gör att algoritmen behöver hantera mycket färre regler, men hämmar utskriften av beviset som då blir mer svårläst.

Väl konverterat går informationen igenom en funktion som delar upp den i tre delar. Första urskiljningen är att alla termer (förutom funktioner) skrivs till en egen lista. Detta medför också att alla termer sammankopplade av konjunktioner bryts isär och placeras i tidigare nämnde lista. Andra mängden blir de predikat och funktioner som förekommer i teoremet. På grund av dess mer komplexa hantering i samband med allkvantifiering (som togs upp i avsnitt 2.6.2) skiljs de åt till en egen lista. Den sista delen är alla disjunktioner som är underlag för de förgreningar som görs. Ett ytterligare steg i denna del av processen är att alla variabler som förekommer från allkvantifiering sparas undan separat vars anledning blir mer konkret nedan i samma avsnitt.

När väl informationen är uppdelad börjar algoritmen med att försöka hitta en motsägelse bland variablerna och konstanterna. Om det inte finns någon motsägelse där görs nästa försök bland predikaten och funktionerna. Om den inte finner två komplementära predikat eller funktioner (t.ex. $P(x)$ och $\neg P(x)$) kommer sökningen istället att, med hjälp av unifiering, finna kvantifieringar som tillsammans med befintliga predikat och funktioner kan leda till en motsägelse. Om inte heller detta leder till en motsägelse kommer istället den första disjunktionen att brytas isär och användas som underlag för en fortsatt sökning. Algoritmen kommer då som tidigare att bryta isär information och söka

vidare efter motsägelser. Detta fortsätter tills att antingen en motsägelse är funnen i varje gren (se avsnitt 2.6.2) eller tills det i en gren inte går att finna en motsägelse, vilket terminerar algoritmen.

Om sökningen lyckas kommer varje söksteg och en lista av använda regler att skrivas ut, i andra fall visas de söksteg som togs fram tills att sökningen terminerats.

I fall där allkvantifiering finns med i problemet sker en del extra steg utöver den vanliga sökningen. Det viktigaste är att varje allkvantifiering sparas för att användas som underlag när algoritmen söker efter möjligheter att finna en motsägelse med predikat och funktioner. När sökningen vänder sig till predikat och funktioner för att stänga en gren görs en ny instansiering av den allkvantifierade formeln. Instansieringen används sedan för att försöka unifieras med övrig information och på så vis finna en motsägelse.

De regler som genereras vid en lyckad sökning används för att skapa en representation av lösningen i trädform (se 4.1). En stor fördel med att skriva om lösningen till en trädrepresentationen är att alla de regler som applicerats men inte används för att finna lösningen kan ignoreras, vilket gör det resulterande trädet mycket mer läsbart. Trädet kan slutligen användas för att skapa en LaTeX-representation som kan ses i figur 3.5. Samma bevis visas även i listform i tabell 3.7.

Tabell 3.7: *Ett bevis i listform.*

1.	r	*Premiss*
2.	$p \wedge q$	*Premiss*
3.	$\neg(q \wedge r)$	*Antagande*
4.	$\neg q \vee \neg r$	$DNF\ 3$
5.	$\neg r$	*Antagande*
6.	$\bot$	$\neg_e\ 1-4$
7.	q	$\wedge_{e2}\ 2$
8.	$\neg q$	*Antagande*
9.	$\bot$	$\neg_e\ 6-7$
10.	$\bot$	$\vee_e\ 3,\ 4-5,\ 7-8$
11.	$q \wedge r$	$RAA\ 3-9$

$$\dfrac{\dfrac{\dfrac{[\neg(q \wedge r)]}{\neg q \vee \neg r}\ DNF \quad \dfrac{r \quad [\neg r]}{\bot}\ \neg e \quad \dfrac{\dfrac{p \wedge q}{q}\ \wedge e_2 \quad [\neg q]}{\bot}\ \neg e}{\bot}\ \vee e}{q \wedge r}\ RAA$$

Figur 3.5: *Trädrepsentation av ett bevis i* LaTeX.

3.4.3 Narrowing

Att utföra narrowingberäkningar utan begränsningar är ganska meningslöst eftersom sökrymden är stor. Det visar sig dock att trädstrukturen hos bevis konstruerade genom naturlig deduktion passar ganska bra eftersom varje regel har sitt eget förutsättningskrav. Man kan alltså kontrollera dessa bevis lokalt genom att för varje regel kontrollera att den deriverade formeln passar med de förutsättande formlerna uppåt. Representationer som t.ex. en linjär form medför merarbete då narrowing både måste gissa reglerna och vilka intervall de skall appliceras på innan dess riktighet kan kontrolleras. Lokaliteten hos trädrepresentationen betyder också att det passar för narrowing då vissa delar av beviset kan vara partiellt instansierade men ändå gå igenom checkern.

Ett bevis för följden $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ byggs nedifrån med slutsatsen ψ som grund varpå regler appliceras "baklänges" och på detta vis manipulerar denna tills hypoteser eller premisser ϕ_i nås. Detta fungerar alldeles utmärkt för de flesta deduktionsregler. Dock finns det några som ställer till problem som t.ex.

$$\frac{\phi \vee \psi \quad \begin{array}{c}[\phi] \\ \vdots \\ \chi\end{array} \quad \begin{array}{c}[\psi] \\ \vdots \\ \chi\end{array}}{\chi} \vee_e$$

I detta fall närmar vi oss regeln underifrån enbart med information om formeln χ. Att verifiera så att grenarna:

$$\phi \vee \psi \quad \begin{array}{c}[\phi] \\ \vdots \\ \chi\end{array} \quad \begin{array}{c}[\psi] \\ \vdots \\ \chi\end{array}$$

stämmer lokalt skapar problem då vi inte känner ϕ och ψ utan enbart att χ är deriverad från dessa. En möjlig lösning är att låta narrowing instansiera formlerna ϕ och ψ så att bevisverifierare kan kontrollera riktigheten lokalt utan att instansiera hela trädet. Detta visar sig vara något kostsamt då den fullständiga begränsningen på dessa formler inte kommer fram förrän ett löv nåtts.

Den speciella narrowingstrategi som används är en variant av lazy narrowing, utvecklad av Lindblad et al [15], att användas för programverifiering av haskellprogram i verktyget Lazy SmallCheck[19]. Fullständigheten för sökningen följer av fullständigheten för strategin [18] samt motsvarande i naturlig deduktion som är vår omskrivningsteori. Narrowingimplementationen tillhandahåller begränsningar enligt nedanstående haskelldatatyp.

```
data Prop = Ok Int
          | Fail
          | AndS Prop Prop
          | AndR Prop Prop
          | EqInt Int Int
          | IntervalInt Int Int Int
```

En egen datastruktur för bevis deriverade av narrowing fick utvecklas på grund av de begränsningar i lokal information som finns att tillgå vid lokal verifiering samt att narrowingbiblioteket enbart instansierar heltal.

```
data Proof = AndI Proof Proof
           | OrI1 Proof
           | OrI2 Proof
           | HypRef Int [Elim]
           [..]
```

Vid de flesta eliminationsregler används en elegantare form av regel HypRef där man direkt refererar till en premiss eller deriverat antagande och därmed slipper problem i de flesta regler som uppenbarades i $\vee_e$. Då kan dessa regler istället elimineras ovanifrån utan att narrowing måste instansiera hela formeln explicit. För att detta skall fungera så konverteras alla identifierare för variabler, predikat och funktioner till heltal. Även hypotesreferenser i löv görs med heltalsindexerade listor likt de Bruijn-index för lambda kalkyl [3]. Rent tekniskt sker sökningen genom att verifieringsfunktionen bygger upp ett träd av dessa begränsningar som måste gälla för att just det verifierade beviset skall fungera. Exempel på detta kan vara intervall för vilka variabler som är tillåtna att instansiera i någon term. Hur begränsningarna som narrowingbiblioteket tillhandahåller

kan användas kan ses för $\wedge_i$:

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \wedge_i$$

som kan verifieras genom att kräva att båda delgrenarna ϕ och $\phi \rightarrow \psi$ går igenom okej. För narrowingbiblioteket motsvarar detta att båda grenarna måste evaluera till `Ok Int` (där heltalet beskriver en kostnad för den instansieringen). Detta kan göras med begränsningen `AndS` som uttrycker en konjunktion. Sjävla verifieringen kan gå till på detta vis:

```
check f (AndI a b) = case f of
    And x y -> AndS (check x a) (check y b)
    _ -> Fail
```

Kapitel 4

Resultat och diskussion

I detta kapitel kommer resultaten och erfarenheterna från projektet att presenteras och diskuteras. Först kommer en analys av implementationen av projektets kärna och sedan jämförs teorembevisarna med avseende på hastighet, implementationssvårighet och tydlighet av genererade bevis. Därefter kommer en lista med möjliga vidareutvecklingar och slutligen följer ett stycke som knyter ihop rapporten och återkopplar till syfte och projektmål.

4.1 Analys av projektets kärna

De moduler som utgör kärnan av projektet är representationerna av bevis, bevisverifieraren och visualiseringen av bevis. Som nämnts i avsnitt 1.3 skrevs dessa i ett tidigt skede av projektet för att ha en stabil grund att bygga vidare på.

De abstrakta representationerna av bevis har alla olika för- och nackdelar. En fördel med Proof är att scope för antaganden är inbyggt i strukturen vilket är smidigt när man skall verifiera att bevis är korrekta. LProof har som fördel att den är väldigt enkel att skriva ut och att det blir lättlästa och tydliga bevis. Den har dock inte något inbyggt stöd för att representera scope för antaganden, vilket leder till en del extra arbete när man verifierar dem. Med extra arbete menas att antagandens giltighet och huruvida exempelvis en implikationsintroduktion verkligen börjar med ett antagande måste verifieras. Det måste också kontrolleras att antaganden endast refereras till i sin helhet utanför dess scope. Både Proof och LProof har som nackdel att man i de automatiska teorembevisarna måste hantera alla nummerreferenser vid generering av bevis. Eftersom TProof inte innehåller några nummerreferenser är den lämplig att använda i teorembevisarna. Denna representation är den som mest påminner om naturliga deduktionsträd vilket gör den lämpad för att användas i LaTeX-ritaren.

Visualiseringen av bevis gick förhållandevis lätt att implementera, då funktionaliteten för att skriva ut listbevis utnyttjar autogenererade funktioner från BNFC. Funktionen som skriver ut naturliga deduktionsträd i LaTeXvar dock svårare att implementera. Att skriva ut bevisen var inte speciellt svårt eftersom LaTeX-ritaren tar in ett TProof, vilket är en väldigt intuitiv struktur att skriva ut på detta sätt. Att få dem att se bra ut var dock inte lika enkelt, exempelvis var det förhållandevis svårt att undvika utskrift av redundanta paranteser. Resultatet blev dock tillfredställande och bevisen är tydliga och lättlästa.

4.2 Analys av automatiska teorembevisare

För att kunna analysera och dra slutsatser rörande implementationen av teorembevisarna har det utförts ett prestandamätning. De olika bevisarna har fått bevisa 22 olika teorem 1000 gånger och tiden har mätts. Med denna data som grund jämförs de implementerade metoderna.

Även läsbarheten för var och en av teorembevisarna jämförs. Läsbarhet har haft en hög prioritet i projektet. Graden av läsbarhet för ett bevis beror på flera olika faktorer; hur stort det är, hur många olika inferensregler som använts, hur lätta formlerna i beviset är att förstå samt om beviset i sig utförts på ett, för människan, intuitivt sätt. Ett exempelproblem bevisas med var och en av bevismetoderna och presenteras på trädform för jämförelse. Slutligen diskuteras även svårighetsgraden på implementationen för vardera metod.

4.2.1 Resultat av prestandamätning

Som det nämnts ovan har de implementerade teorembevisarna genomgått en mätning för att bilda en uppfattning om hur snabbt var och en av dem bevisar olika teorem. Dessa teorem valdes till en mängd som alla teorembevisarna klarade av att bevisa. I tabell 4.1 visas resultatet av denna mätning. Det har utförts på en dator med en Intel(R) Core(TM) 2 Duo 2.0GHz processor med 4096KB cache och 4GB dualchannel 667MHz RAM. Applikationen har inför mätningen kompilerats med GHC 6.8.2 och följande kommandorad har använts: ”ghc –make Main.hs -O2 -optc=03”.

Tabell 4.1: *Resultat av prestandamätning.*

Nr.	Problem	Resolution	Tableau	Narrowing
1.	$\vdash \neg p \vee p$	1,012	0,892	18,217
2.	$a, b \vdash a \rightarrow b$	1,192	0,912	1,388
3.	$p, q \vdash p \wedge q$	1,288	1,116	1,56
4.	$p, \neg\neg(q \wedge r) \vdash \neg\neg(p \wedge r)$	1,444	1,42	11,637
5.	$p \vdash p$	0,86	0,768	0,948
6.	$p \wedge q, r \vdash q \wedge r$	1,336	1,288	2,156
7.	$p \wedge q \vdash p$	1,044	0,976	1,232
8.	$p \wedge q \wedge r, s \wedge t \vdash q \wedge s$	1,428	1,3	2,772
9.	$p \wedge q \rightarrow r \vdash p \rightarrow q \rightarrow r$	1,724	1,448	5,412
10.	$p \rightarrow q, p \vdash q$	1,192	1,248	1,416
11.	$p \rightarrow q \vdash \neg p \vee q$	1,476	1,228	39,762
12.	$p \rightarrow q \rightarrow r, p \rightarrow q, p \vdash r$	1,584	1,34	3,064
13.	$p \vee q \vee r \vdash p \vee q \vee r$	1,66	1,316	1,104
14.	$a \vee b \vdash b \vee a$	1,412	1,148	9,673
15.	$a \vee b \vee a \vdash a \vee b$	1,3	1,496	3,952
16.	$p \rightarrow q \vdash \neg q \rightarrow \neg p$	1,308	1,32	3,324
17.	$\neg q \rightarrow \neg p \vdash p \rightarrow \neg\neg q$	1,424	1,272	4,488
18.	$\forall x.(P(x) \rightarrow Q(x)), P(t) \vdash Q(t)$	1,396	1,44	2,2
19.	$\forall x.(P(x) \rightarrow \neg Q(x)), P(t) \vdash \neg Q(t)$	1,328	1,328	2,532
20.	$\forall x.P(x) \vdash \forall x.P(x)$	1,244	1,18	1,284
21.	$\exists x.P(x) \vdash \exists x.P(x)$	1,244	1,128	1,424
22.	$\forall y.(y \wedge b), a \vee c \vdash b$	1,444	1,116	1,684
Totalt:		29,34	26,68	121,229

Tiderna har mätts med unix-kommandot *time* och det är tiden *”user”* som valts. Den-

na tid beskriver hur lång tid processorn tar på sig för att exekvera just det angivna kommandot, utan att ta hänsyn till tid nedlagd på andra program som körs samtidigt. Resultatet är inte tänkt att ge en rättvis bild av hur snabbt varje enskild teorembevisare jobbar, utan meningen är att man ska kunna jämföra dem sinsemellan.

4.2.2 Thousands of Problems for Theorem Provers (TPTP)

TPTP[27] tillhandahåller en stor mängd problem som är menade att utgöra en rigorös bas för testning av automatiska teorembevisare. Problemen är indelade i kategorier och har i många fall en viss bevisalgoritm i åtanke.

För att undersöka hur bra våra olika teorembevisarna arbetar på mer komplexa och intressantare problem har två problem från TPTP testats, nämligen pusslen: PUZ001+1.p och PUZ061+1.p. Resolution lyckades bevisa båda dessa på mycket kort tid och det senare beviset blir hela 38 steg långt. Tableaumetoden finner ett bevis för ett av pusslen (PUZ061+1.p), men inte för det andra. Det är möjligt att narrowing klarar att lösa pusslen, men det tar i sådana fall mycket lång tid.

Det är både roligt och intressant att någon av bevisarna klarade av att hitta bevis eftersom detta test verkligen mäter var gränsen går för vårt system. Nedan visas pusslet PUZ001+1.p (fritt översatt) och man kan se att det inte är helt trivialt:

> Någon som bor på Hovs Herrgård har dödat Agatha. Agatha, betjänten och Charles bor på Hovs Herrgård och är de enda som bor där. Alla mördare hatar sina offer och är inte rikare än sina offer. Charles hatar ingen som Agatha hatar. Agatha hatar alla utom betjänten. Betjänten hatar alla som inte är rikare än Agatha. Betjänten hatar alla som Agatha hatar. Ingen hatar alla. Agatha hatar inte alla. Vem mördade Agatha?

Teoremet som skall bevisas är att Agatha mördade sig själv, det vill säga att Agatha begick självmord.

4.2.3 Resolution

Eftersom all indata konverteras till konjunktiv normalform och därmed skolemiseras, tas kvantifierare helt bort. När sedan alla konjunktioner brutits ner i klausuler återstår endast en delmängd av det logiska språket som använts i projektet. För att deducera ett bevis används endast 4 (av totalt 22) inferensregler vilket tillsammans med den enkla och effektiva resolutionsregeln gör det mycket lätt att bevisa teorem.

Denna teorembevisare har dock en stor svaghet; många onödiga beräkningar görs för att deducera ett bevis. Varje klausul i kunskapsbasen testas mot resten, vilket innebär att antalet beräkningar som görs är kvadratiskt mot antalet klausuler i kunskapsbasen. Om det sedan finns många literaler i klausulerna så kommer även själva appliceringen av inferensregeln bli mycket beräkningstung. Detta leder till att det tar mycket lång tid att bedöma om ett problem är satisfierbart.

I tabell 4.1 går det att avläsa att resolution är lite långsammare än tableau på de flesta av bevisen. Hastigheten är dock bra eftersom alla bevis tar ungefär lika lång tid. Det är svårt att dra några djupare slutsatser från mätdata, men man kan spekulera i att resolution skulle vara snabbare än tableau på mer avancerade problem med fler kvantifierare. Detta eftersom tableau då måste hålla reda på unifierare i olika grenar, vilket resolution inte behöver hantera. Ett sådant resultat har presenterats i [6] där resolution och tableau jämförts, och resolution visades vara den mer effektiva av de två.

Läsbarhet av genererade bevis

Bevisning av teorem med resolution är passande för datorer. Detta eftersom bevisning sker med en mycket liten uppsättning inferensregler och att dessa appliceras i samma ordning för alla teorem. Metoden är därför inte lika intuitiv som naturlig deduktion.

Bevis blir i de flesta fall större än om de gjorts för hand. Detta eftersom ett antal steg måste tas för att förbereda indata innan appliceringen av resolutionregeln kan börja. En förutsättning för att förstå dessa bevis är att man har kunskap om hur konvertering av formler till konjunktiv normalform går till samt hur applicering av resolutionregeln går till.

I detta projekt har det valts att ha resolutionregeln som en enkel inferensregel, men den kan även ses som ett makro bestående av inferensregler från naturlig deduktion som producerar samma resultat. Om inte resolutionregeln används som ett makro, utan ersätts av regler från naturlig deduktion, leder detta till att bevis blir nästintill omöjliga att förstå. En applicering av resolutionregeln på två klausuler, innehållande två literaler var och endast ett par komplementära literaler, innehåller 20-talet steg.

$$\dfrac{r \quad \dfrac{\dfrac{[\neg(q \wedge r)]}{\neg q \vee \neg r}\, CNF}{\neg q}\, RES \qquad \dfrac{p \wedge q}{q}\, \wedge e_2}{\dfrac{\bot}{q \wedge r}\, RAA}\, RES$$

Figur 4.1: *Teorem bevisat med resolution och utskrivet som ett naturligt deduktionsträd.*

Generellt sett producerar resolution dock bevis som går att läsa och förstå, givet att antalet klausuler och literaler inte är alltför stora. Det används också få inferensregler vilket leder till att det är mindre att ta i beaktning. Ett exempel på ett bevis deriverat med resolution återfinnes i figur 4.1 där problem 6 i tabell 4.1 bevisats.

Implementation

Resolution var den lättaste teorembevisaren att implementera. Själva kärnfunktionaliteten skrevs på mycket kort tid i och med den effektiva inferensregeln och det begränsade logiska språk som användes.

Modulen korrigerades bara några få gånger, t.ex. då unifiering implementerades, men eftersom korrigeringarna bara begränsades till inferensregeln gick även det snabbt. Majoriteten av de buggar som berört resolution har istället upptäckts i modulerna som sköter konvertering till konjunktiv normalform och unifiering.

Det fanns till och med tid att implementera ett antal optimeringar och i det området har en stor majoritet av tiden investerats. Bland annat implementerades en optimering för att förbättra prestandan och en för förbättring av läsbarheten av genererade bevis. Den senare nämnda optimering tog nästan längre tid att implementera än den första versionen av hela modulen och var därför den mest tidskrävande delen.

En sak som underlättade implementeringen av denna modul var att det redan fanns en färdig representation av ett linjärt bevis och att bevisverifieraren hade en linjär struktur. Detta kan jämföras med de båda andra teorembevisarna som bygger på trädstrukturer och därför haltades något av att funktionaliteten för konvertering mellan de två representationerna länge krånglade.

Implementeringen av denna teorembevisare har också varit mycket underhållande eftersom man till stor del får fria händer, med undantaget från konverteringen till kon-

junktiv normalform och användandet av resolutionregeln. Det finns många dokumenterade strategier [25] för att förbättra prestanda, t.ex. vilka klausuler man ska applicera resolution på samt hur man väljer att dela upp kunskapsbasen.

4.2.4 Semantisk Tableau

Enligt tabell 4.1 är tableau den av de tre metoderna som arbetar snabbast på de test problemen som använts. Det kan också noteras att sökrymden för tableau ökar mycket snabbt när en stor mängd disjunktioner innefattas i teoremet. Detta kan beskådas i resultaten för problem nummer 4, 9 och 15 som alla leder till många disjunktioner efter att ha konverterats till DNF. Andra mätvärden som står ut är ett par som innehåller allkvantifiering, nämligen problem 18 och 19. Hanteringen av just predikat och allkvantifieringar innefattar en mängd fler beräkningssteg som annars inte utförs. Som teorin tagit upp måste teorembevisaren börja leta efter möjliga nya instansieringar av variabler och detta leder längre exekveringstid. Enligt testet som utförts klarar tableau de flesta problemen relativt väl, men detta kan till stor del bero på att många av dem varken innehåller några kvantifierare eller predikat.

Läsbarhet av genererade bevis

Algoritmen är designad för att effektivt avgöra om ett problem går att bevisa och inte för att producera lättlästa lösningar. Läsbarheten av teoremen lider av att alla formler översätts till DNF, och att ett motsägelsebevis eftersträvas istället för att direkt visa slutsatsen. I de flesta fall består lösningen av ett antal konjuktionseliminationer i början, några antagande som avslutas med en negationselimination och en disjunktionselimination som binder ihop allt. Eftersom ett motsägelsebevis deriveras så avslutas det alltid med att RAA-regeln appliceras. I figur 4.2 visas det sjätte problemet ur tabell *4.1* bevisat med semantisk tableau.

$$\dfrac{\dfrac{\dfrac{[\neg(q \wedge r)]}{\neg q \vee \neg r}\ DNF \quad \dfrac{r \quad [\neg r]}{\bot}\ \neg e \quad \dfrac{\dfrac{p \wedge q}{q}\ \wedge e_2 \quad [\neg q]}{\bot}\ \neg e}{\bot}\ \vee e}{q \wedge r}\ RAA$$

Figur 4.2: *Teorem bevisat med semantisk tableau och utskrivet som ett naturligt deduktionsträd.*

Implementation

Tableau-implementering har, för en delmängd av det logiska systemet som inte innehåller kvantifierare eller predikat, haft rätt struktur sedan starten. Men på grund av otillräcklig teoretisk grund har det gjorts om flera gånger för att på ett bra sätt bevisa problem som innefattar allkvantifiering. Detta har varit den svåraste aspekten av implementationen. Slutsatsen är att tableaumetoden till en början är enkel att implementera, men när allkvantifiering introduceras dyker många problem upp. Ett råd för den som vill göra en egen implementation är att förbereda sig väl teoretiskt med någon bra bok, som till exempel [5] skriven av Fitting.

4.2.5 Narrowing

Denna bevismetod visar sig vara den långsammaste av de tre vilket också var väntat. Detta beror på att metoden bara är en indirekt sökning, den ”förstår” inte språket på samma sätt som resolution eller tableau-metoderna. Vid varje narrowingsteg körs bevisverifieraren på problemet för att verifiera att allting fortfarande gäller och återge de nya begränsningarna. Längst tid att bevisa tog problemen 1, 4 och 11 enligt tabell 4.1. Alla tre har gemensamt att de bevisas genom att derivera $\perp$ vilket ger längre bevis än i de andra fallen vilket också leder till en längre sökning. Jämfört med de andra två metoderna så verkar narrowingsökningen rätt oanvändbar, men eftersom den jobbar på partiellt instansierade termer kan den fortfarande vara till nytta i en interaktiv bevisassistent.

I [16] undersöks hur väl narrowing lämpar sig i Agda, en bevisassistent för Martin-Lövs intuitionistiska typteori. Även här kommer författaren till samma slutsats att det är långsamt, men snabbt nog för små bevis där människan är flaskhalsen. I stycket 4.3 diskuteras en grafiska teorembevisare lite ytterligare som tilllämpning för teorembeviare.

Läsbarhet av genererade bevis

Bevis hittade med narrowingsökningen kan se ut precis hur man önskar genom att ändra utseendet på inferensreglerna i verifieraren. Människor uppskattar troligen bevis funna med narrowing då de innehåller ett rikare språk av inferensregler (jämfört med resolution och semantisk tableau) som liknar hur människor drar slutledningar. I implementationen används hela språket av naturlig deduktion utom vissa deriverade regler som MT (formel 2.15 på sidan 14) och LEM (formel 2.18 på sidan 14). I figur 4.3 visas det sjätte problemet i tabell 4.1.

$$\dfrac{\dfrac{p \wedge q}{q}\ \wedge e_2 \qquad r}{q \wedge r}\ \wedge i$$

Figur 4.3: *Teorem bevisat med narrowing och utskrivet som ett naturligt deduktionsträd.*

Implementation

I teorin borde narrowingsökningen vara den enklaste metoden av de tre att implementera då den enbart består av en bevisverifierare på trädform. Tanken var att använda en färdig verifierare och med några små modifikationer få den att bli en fullständig sökning med minimal arbetsinsats. I praktiken visade det sig dock vara ganska svårt att få till sökningen då man enbart har indirekt kontroll genom de begränsningar som återges till algoritmen. Svårigheterna i sökningen ligger att finna kreativa men effektiva sätt att begränsa hur omskrivningsreglerna appliceras på slutsatsen. En djupare förståelse för omskrivningssystem och definitionsträd är troligen till mycket stor nytta vid designen av dessa. Ett exempel på en sådan begränsning är t.ex. att en kostnad sätts på regeln för motsägelsebevis och ökas ordentligt för varje gång den används i samma gren. En sådan kostnad ”uppmuntrar” sökningen att försöka med andra regler före den använder samma regel igen då den görs med ett ökande djup på den maximala kostnaden.

4.3 Vidareutveckling

Detta avsnitt kommer handla om potentiella vidareutvecklingar på projektet. Eftersom det är ett väldigt brett ämne finns det väldigt många olika tillämpningar som vore

intressanta att implementera. Detta avsnitt berör dock endast utökningar som vore relevanta för just detta projekt.

Rikare språk

För att få ett rikare språk vore det intressant att utöka det med fler konnektiv. Detta skulle exempelvis kunna innebära att man lägger till ekvivalens (dubbelriktad implikation) och likhet.

Fler bevisverifierare

Bevisverifieraren kan endast verifiera korrektheten hos bevis av typen Proof. Detta innebär att konvertering från LProof respektive TProof måste göras för att verifiera bevis genererade av teorembevisarna. Om bevisverifierare som hanterar dessa datatyper implementeras slipper man detta och det går då även att testa dessa mot varandra för att försäkra sig om att både konverteringen och verifierarna sinsemellan är korrekta.

Parser för TPTP

Det var otroligt arbetsamt att manuellt skriva in problemen från TPTP direkt i vår egen syntax och därmed ett hinder för att testa mer en ett fåtal problem från denna databas. Då den innehåller ett tusental problem kan detta ses som en nödvändig utökning för effektiv optimering av metoderna.

Optimerad teorembevisare

Det vore intressant att försöka optimera någon av teorembevisarna och försöka lösa större problem. Exempelvis kunde man försöka bevisa fler teorem från TPTP.

Noggrannare testning

Eftersom det är ett stort system med många möjliga indata, är det svårt att testa att allt i systemet fungerar som det ska. Det har både funnits en uppsättning med problem som bevisverifieraren testats mot och en uppsättning som teorembevisarna testats mot. Samlingen med problem hade behövts utökas för att göra en riktigt rigorös testning och optimering av algoritmerna. En möjlig lösning på detta hade varit att skriva en parser för TPTP enligt ovan eller kanske generera problem automatiskt.

Grafisk bevisassistent

För studenter och lärare kan en interaktiv teorembevisare med ett grafiskt gränsnitt vara ett bra verktyg vid inlärningen av predikatlogik. Denna skulle kunna fungera så att man interaktivt försöker bevisa ett teorem och att programmet då kan hjälpa en och säga till om man gör fel eller visa möjliga lösningar.

4.4 Avslutande reflektioner

Studien var ämnad att beröra olika implementationsmetoder för att verifiera, bevisa och visualisera logiska slutledningar. De olika teorembevisarna skulle även jämföras utifrån prestanda, implementationssvårighet och hur tydliga och lättlästa de deducerade bevisen är. Allt detta har gjorts och många slutsatser har dragits angående fördelar och nackdelar med de olika representationerna samt designvalen i implementationen av teorembevisarna. Teorembevisarna har jämförts och man kan sluta sig till att:

- Vår implementation av semantisk tableau är den snabbaste.
- Resolution var enklast att implementera och var den enda som klarade båda av de två problemen från TPTP.
- Narrowing producerar mest lättlästa bevis.

Litteraturförteckning

[1] S. Antoy och S. Lucas. Demandness in rewriting and narrowing. I: *Proc. of the 11th International Workshop on Functional and (constraint) Logic Programming (WFLP'02)*, ss 35–43, Grado, Italy, June 2002.

[2] Sergio Antoy, Rachid Echahed och Michael Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[3] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[4] R. Caballero (ed), J. Sánchez (ed), P. Arenas, A. Fernández, A. Gil, F.J. López Fraguas, M. Rodríguez Artalejo och F. Sáenz. $\mathcal{TOY}$. A Multiparadigm Declarative Language. Version 2.3.1. Available at `http://www.fdi.ucm.es/profesor/fernan/toy/`, October 2007.

[5] Melvin Fitting. *First Order Logic and Automated Theorem Proving*. Springer, 1996.

[6] Andreas L.E. Folkler. Automated theorem proving: Resolution vs. tableaux, March 2002.

[7] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39(1):405–431, 1935.

[8] Kurt Gödel. Die vollständigkeit der axiome des logischen funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930.

[9] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[10] Michael Hanus. Implementations of functional logic languages. `http://www.informatik.uni-kiel.de/~mh/FLP/implementations.html`.

[11] Michael Hanus. Multi-paradigm declarative languages. I: *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, ss 45–75. Springer LNCS 4670, 2007.

[12] Karel Hrbacek, Thomas Jech och Hrbacek Hrbacek. *Introduction to Set Theory, Third Edition, Revised and Expanded*. CRC, Juni 1999.

[13] Michael Huth och Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2004.

[14] Matt Kaufmann och J Strother Moore. Some key research problems in automated theorem proving for hardware and software verification. *Real Academia de Ciencias Serie A. Mathemáticas: Ciencias de la computación*, 98(1–2):181–195, 2004.

[15] Fredrik Lindblad. Property directed generation of first-order test data. The Eighth Symposium on Trends in Functional Programming, New York, 2007.

[16] Fredrik Lindblad. A tool for automated theorem proving in agda. Types for Proofs and Programs, post-proceedings of TYPES 2004, LNCS volume 3839/2006, 2007.

[17] Fredrik Lindblad. Higher-order proof construction based on first-order narrowing. *Electronic Notes in Theoretical Computer Science*, 196:69–84, 2008.

[18] Aart Middeldorp, Satoshi Okui och Tetsuo Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167(1–2):95–130, 1996.

[19] Matthew Naylor, Fredrik Lindblad och Colin Runciman. Lazy SmallCheck: exhaustive, demand-driven testing in Haskell. Talk at Fun in the Afternoon, York, see `http://www.cs.york.ac.uk/~mfn/lazysmallcheck`, November 2007.

[20] Michael Pellauer, Markus Forsberg och Aarne Ranta. BNF Converter Multilingual Front-End Generation from Labelled BNF Grammars. *Computer Science Chalmers University of Technology and Gothenburg University*, 2004.

[21] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 1965.

[22] P. Ribenboim. *The New Book of Prime Number Records*. Springer, NY, 3:e utgåvan, Februari 1996.

[23] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[24] Bertrand Russell. *Principles of Mathematics*. Cambridge University Press, 1903.

[25] Stuart Russell och Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[26] James R. Slagle. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *J. ACM*, 21(4):622–642, 1974.

[27] G. Sutcliffe och C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.