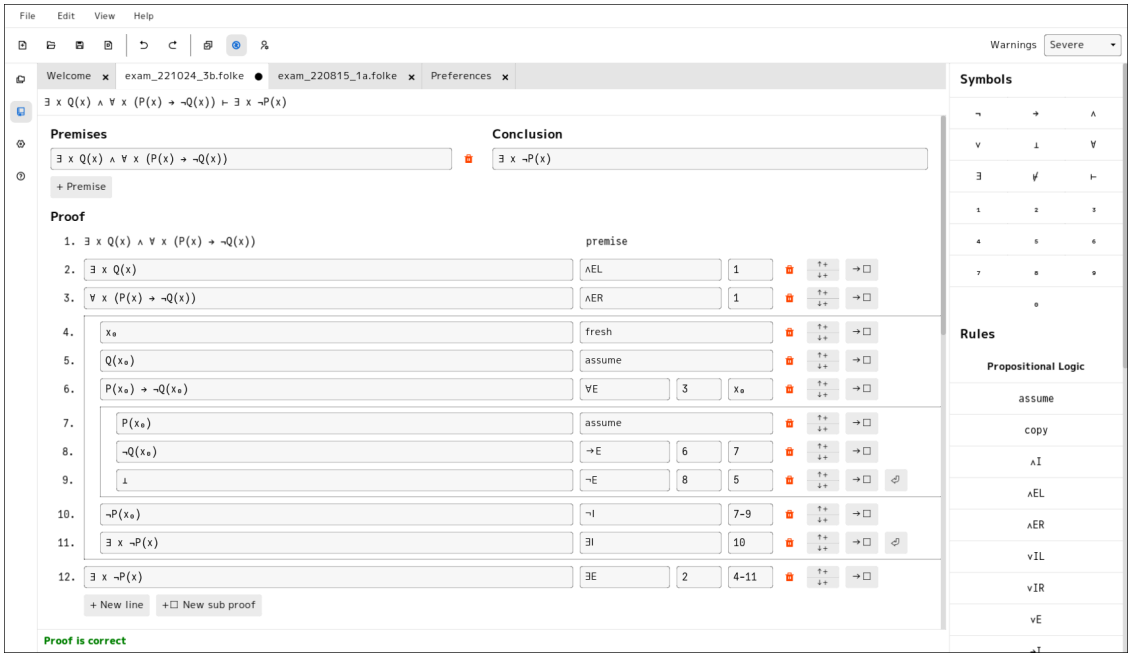
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Folke: En bevisredigerare för naturlig deduktion

Utvecklingen av ett pedagogiskt verktyg för studenter

Kandidatarbete vid Data- och informationsteknik

JAKOB FREDBY, LUCAS MÖLLER, ALFONS NILSSON, ANTON NORDBECK, ALBIN SCHMIDT

Institutionen för Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2025

Kandidatarbete 2025

Folke: En bevisredigerare för naturlig deduktion

Utvecklingen av ett pedagogiskt verktyg för studenter

JAKOB FREDBY
LUCAS MÖLLER
ALFONS NILSSON
ANTON NORDBECK
ALBIN SCHMIDT

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för Data- och informationsteknik
Chalmers tekniska högskola
Göteborgs Universitet
Göteborg, Sverige 2025

Folke: En bevisredigerare för naturlig deduktion
Utvecklingen av ett pedagogiskt verktyg för studenter
JAKOB FREDBY LUCAS MÖLLER ALFONS NILSSON ANTON NORDBECK
ALBIN SCHMIDT

Projektlänk: `https://github.com/lambducas/folke`

Handledare: Ana Bove, Institutionen för Data- och informationsteknik
Examinatorer: Patrik Jansson, Institutionen för Data- och informationsteknik
Rättande lärare: Wolfgang Ahrendt, Institutionen för Data- och informationsteknik

Kandidatarbete 2025
Institutionen för Data- och informationsteknik
Chalmers tekniska högskola och Göteborgs universitet
SE-412 96 Göteborg
Telefon +46 31 772 1000

Omslag: Ett korrekt bevis för $\exists x Q(x) \land \forall x(P(x) \to \neg Q(x)) \vdash \exists x \neg P(x)$

Typsättning i LaTeX
Göteborg, Sverige 2025

Abstract

This report introduces Folke, a natural deduction proof editor for first-order logic. Logic provides a framework for formal reasoning by constructing new statements from previous ones, beginning from a set of initial statements called premises. A proof is a correct sequence of statements that, for some given premises, reaches a given conclusion. Folke was designed to help students in logic formulate and verify their proofs. Previous software solutions have a tendency to be difficult to use and unintuitive for beginners owing to their complexity. To address this, Folke was developed with a focus on ease of use, featuring an interactive and beginner-friendly interface with clear error messages. The program was implemented in Haskell with a user interface built using the component-based GUI library Monomer. Folke is distributed as a native application available for Linux, Windows, and Mac OS. Future enhancements could include a hint/suggestion system for proof steps, improved parsing for more informative error messages, and a user study to guide further development and uncover potential bugs.

Keywords: proof editor, natural deduction, pedagogy, ux

Sammandrag

Denna rapport introducerar Folke, en bevisredigerare för naturlig deduktion för första ordningens logik. Logik tillhandahåller ett ramverk för formellt resonerande där man konstruerar nya påståenden utifrån initiala påståenden som kallas premisser. Ett bevis är en korrekt formulerad sekvens av påståenden som utifrån givna premisser når en önskad slutsats. Folke var utformad för att hjälpa studenter i logik att formulera och verifiera sina bevis. Tidigare mjukvarulösningar har en tendens att vara svåranvända och ointuitiva för nybörjare på grund av deras komplexitet. För att bemöta detta har Folke utvecklats med fokus på användarvänlighet, med ett interaktivt och nybörjarvänligt gränssnitt med tydliga felmeddelanden. Programmet har implementerats i Haskell, och användargränssnittet har byggts med hjälp av det komponentbaserade GUI-biblioteket Monomer. Folke distribueras som ett program tillgängligt för Linux, Windows och Mac OS. Framtida förbättringar kan inkludera ett hjälpsystem för bevissteg, förbättrad tolkning för mer informativa felmeddelanden samt en användarstudie för att vägleda fortsatt utveckling och identifiera eventuella buggar.

Nyckelord: bevisredigerare, naturlig deduktion, pedagogik, ux

Förord

Vi vill rikta ett stort tack till vår handledare Ana Bove. Utan hennes stöd, goda kommunikation och vägledning genom projektets gång skulle det inte vara möjligt att genomföra projektet.

Jakob Fredby, Lucas Möller, Alfons Nilsson, Anton Nordbeck, Albin Schmidt,
Göteborg, juni 2025

Innehåll

Figurer

Tabeller

Ordlista

Arbetskatalog	En mapp som associeras med en process. Filsökvägar är relativa till arbetskatalogen.
Aritet	Antal argument för en funktion eller operand
BNF	Backus-Naur Form
BNFC	Backus-Naur Form Converter
GUI	Grafiskt gränssnitt (Graphical User Interface)
IDE	Integrerad utvecklingsmiljö (Integrated Development Environment)
JSON	JavaScript Object Notation
LBNF	Labeled Backus-Naur Form
Markdown	Ett språk för formatering av ren text
Open-source	Källkod som är tillgänglig att använda och ändra
Syntaktiskt socker	Tillägg till ett programmeringsspråk som gör det enklare att använda utan att ändra funktionaliteten

1

Inledning

Logik är en viktig del av många olika tekniska och vetenskapliga områden såsom matematik och datavetenskap. Logik används inom matematiken som en fundamental byggsten för att formulera och bevisa satser. Inom datavetenskap används logik för att bekräfta korrektheten av algoritmer. Därför finns det nytta i att kunna logik för flertalet olika vetenskapliga discipliner. Det finns olika system inom logik och sätt att bedriva resonemang utifrån dessa. Ett av systemen för att göra detta kallas naturlig deduktion. Det påminner om ett vardagligt eller naturligt sätt att resonera. Det kan liknas vid matematiken, men istället för att siffror manipuleras, manipuleras istället påståenden. Resonemanget utgår från ett antal antagna påståenden och sker i steg för att nå en slutsats. För detta projekt kommer naturlig deduktion att användas så som det presenteras av Huth och Ryan i *Logic in Computer Science* [1].

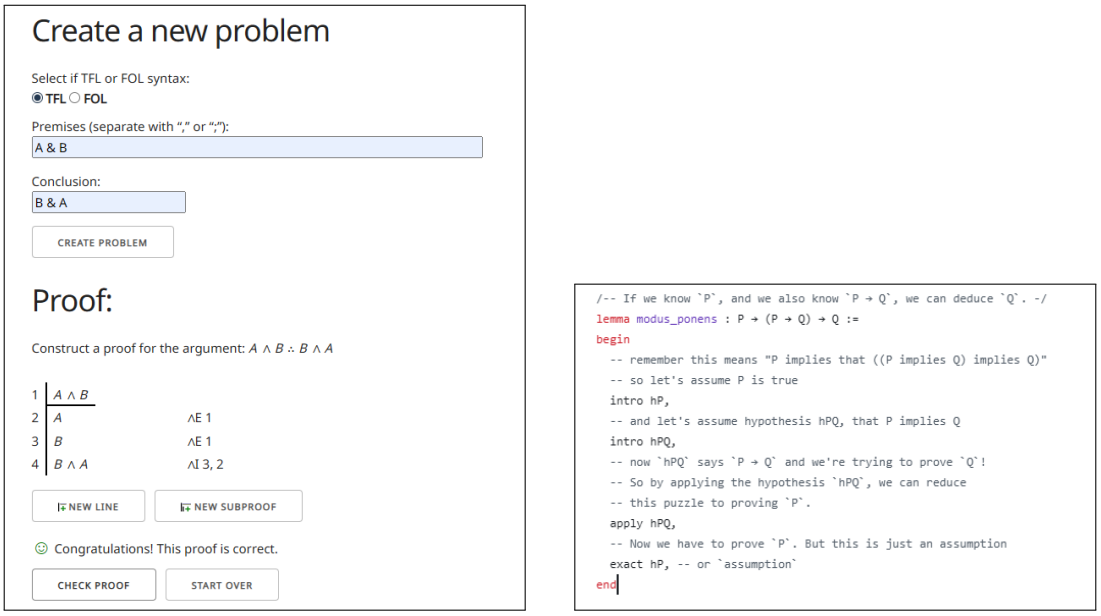
Med hänsyn till bredden av logik som koncept finns det därför intresse i att underlätta författandet av logiska bevis, särskilt inom ramar av pedagogiska ändamål. Således implementerar vi en bevisredigerare som skall underlätta lärandet av naturlig deduktion. Målgruppen förväntas ha lämplig förkunskap inom logik och matematiska resonemang. Med detta som mål har vi identifierat den primära målgruppen att vara logikstudenter på masternivå. I synnerhet de som finner behov av att författa bevis på ett mer interaktivt, pedagogiskt assisterande och intuitivt sätt. En bevisredigerare gör det möjligt för studenten att snabbt formulera, korrigera och få omedelbar återkoppling på sina bevis vilket är något som traditionella analoga metoder inte kan matcha. Redigeraren är utformad som ett användargränssnitt med grafisk representation av bevis. Bevisredigeraren ska för användarens angivna antaganden och slutsats, granska stegen som görs för att nå slutsatsen. Den ska då identifiera fel och förklara dem samt konstatera om slutsatsen är bevisad. Fel kan till exempel vara att grammatikens härledningsregler inte har följts, fel regel har använts eller att regeln har använts på ett inkorrekt sätt.

1.1 Syfte

Syftet med projektet är att implementera en bevisredigerare för naturlig deduktion, med fokus på satslogik och första ordningens logik, som gör det enkelt att formalisera och strukturera bevis. Främst är redigeraren avsedd att användas som ett pedagogiskt verktyg. Målgruppen är logikstudenter på masternivå. Bevisredigeraren ska erbjuda en snabb, intuitiv och interaktiv upplevelse för användaren med funktionalitet som stödjer och underlättar självständiga studier.

1.2 Nuvarande programvara

Det finns sedan tidigare ett flertal bevisredigerare för första ordningens logik. Ett exempel är Open Logic Project's *Natural deduction proof editor and checker* [2] vilket är ett grafiskt verktyg för bevisföring inom satslogik och första ordningens logik. Det finns även mer avancerade, textbaserade bevisassistenter. Några exempel är Agda [3], Rocq [4] och Lean [5]. Dessa är mer allmänna verktyg som används för att bevisa satser inom flera vetenskapliga områden. I figur 1.1 visas skillnaden mellan grafiska och textbaserade gränssnitt.



(a) Open Logic Project's bevisredigerare **(b)** Bevisassistenten Lean

Figur 1.1: Jämförelse mellan grafiskt och textbaserat gränssnitt.

Bevisredigerare med textbaserat gränssnitt, såsom Agda och Lean, efterliknar utvecklingsmiljöer för programmering där användaren skriver och strukturerar all text manuellt. Fördelarna är att det ger användaren mycket större frihet till hur bevis struktureras, samt tillåter användaren att utöka funktionaliteten av verktyget genom att exempelvis importera kodpaket. Nackdelen är att det oftast krävs mer grundkunskap om hur syntaxen ser ut samt hur programmet fungerar innan det kan börja användas.

Bevisredigerare med grafiskt användargränssnitt, såsom Open Logic Project's bevisredigerare, har en mer explicit grafisk miljö för skrivandet av bevisen. Användaren förfogas här textrutor som ger struktur till bevisets rader samt knappar för hanterandet av dessa. Detta har fördelen att bevisredigerarens bruk blir mer självförklarlig, tydlig och intuitiv. Nackdelar kan förekomma i att strukturen kan vara begränsande eller ineffektiv, särskilt för en användare som är mer kunnig inom bevisförfattande.

Projektets syfte är att skapa ett program för studenter som precis har påbörjat sina logikstudier och som därmed inte har mycket kunskap inom ämnet. Med respektive gränssnitts för- och nackdelar i åtanke ansågs en bevisredigerare med ett grafiskt användargränssnitt mer lämpad för detta syfte. Med detta gäller det även att erbjuda lösningar utöver det som andra grafiska bevisredigerare redan gör. I detta fallet vill vi alltså särskilt åtgärda vissa av de brister vi anser att Open Logic Project's bevisredigerare har. Framför allt att man måste klicka med musen för att skapa nya rader, samt att det inte går att spara eller exportera bevis.

1.3 Avgränsningar

Satslogik samt första ordningens logik ansågs som de mest inledande för ämnet för masterstudenter, därför ska bevisredigeraren endast hantera dessa. Bevisredigeraren ska inte automatiskt skriva bevis, då syftet är att studenter ska skriva dem själva. För att använda bevisredigeraren måste den installeras lokalt på användarens dator och den har inget stöd för att användas via en webbläsare eller mobiltelefon. Detta för att undvika driftkostnader och serverunderhåll. Bevisredigeraren ska inte heller ha onlinefunktionalitet, såsom kollaboration mellan användare, av samma skäl som tidigare. Då projektet fokuserar mer på det tekniska utvecklandet av bevisredigeraren kommer inga användarstudier att genomföras.

2

Logik och Naturlig deduktion

Logikformen som projektet handlar om kallas naturlig deduktion vilket är ett formellt system för logisk bevisföring. Den uppkom utifrån ett missnöje med dåtidens axiomatiserade härledningssystem och dess moderna form föreslogs av Gerhard Gentzen i början av 1930-talet [6].

Naturlig deduktion liknar matematiken och medan matematik bekymrar sig om abstrakta antal bekymrar sig naturlig deduktion om abstrakta påståenden. Dessa påståenden kan likt nummer kombineras under olika operationer och bindemedel för att bilda formler och således få ny mening. Naturlig deduktion är ett rigoröst ramverk för logiskt tänkande och finner i första hand nytta inom matematik och datavetenskap för att formulera bevis.

2.1 Satslogik

Satslogik är ett språk för att beskriva grundläggande logiska påståenden samt resonemang och från dessa kunna dra logiska slutsatser. Formler i satslogiken uttrycks med hjälp av atomära propositioner och konnektiv. En atomär proposition är definierad som ett påstående som är odelbart och har ett sanningsvärde, sant eller falskt. Dessa kan till exempel vara "det regnar" eller "jag har mitt paraply". Man brukar beteckna dessa påståenden med bokstäver och fortsättningsvis låter vi

p beteckna påståendet "det regnar" och
q beteckna påståendet "jag har mitt paraply".

Alla atomära propositioner är välbildade formler. Förutom detta består välbildade formler även av motsägelse som betecknas $\bot$. Detta uppkommer till exempel när vi har både q och $\neg q$, som kan stå för "det regnar" och "det regnar **inte**", alltså två formler som motsäger varandra. Välbildade formler kan även bestå av andra välbildade formler kombinerade under konnektiv.

Namn	**Symbol**	**Exempel på en sats**
Negation	$\neg p$	"Det regnar **inte**"
Konjunktion	$p \wedge q$	"Det regnar **och** jag har mitt paraply"
Disjunktion	$p \vee q$	"Det regnar **eller** jag har mitt paraply"
Implikation	$p \rightarrow q$	"**Om** det regnar **så** har jag mitt paraply"

Tabell 2.1: Logiska konnektiv i satslogik med exempel.

Konnektiv är väldefinierade logiska operatorer som kombinerar välbildade formler och bildar nya formler. I satslogiken finns det konnektiven $\neg, \wedge, \vee, \rightarrow$. Av dessa binder $\neg$ bara en formel och står för en negation av formeln. De andra konnektiven binder två formler där $\wedge$ står för konjunktion, $\vee$ står för disjunktion och $\rightarrow$ för implikation. Om vi skulle använda exemplen från ovan uttrycker då $p \wedge q$ som "det regnar **och** jag har mitt paraply". Se tabell 2.1 för en översikt av konnektiven tillsammans med exempel.

De atomära propositionerna tillsammans med konnektiven ger följande rekursiva definition av välbildade formler:

$$\phi, \psi ::= \bot \mid p \mid \neg\phi \mid (\phi \wedge \psi) \mid (\phi \vee \psi) \mid (\phi \rightarrow \psi)$$

där p är en atomär proposition, samt ϕ och ψ är båda välbildade formler.

2.1.1 Naturlig deduktion för satslogik

Naturlig deduktion är ett sätt att resonera kring satser. Vi gör detta genom att använda bevisregler som vi applicerar på påståenden för att härleda nya påståenden. En sekvent betecknas

$$\phi_1, ..., \phi_n \vdash \psi$$

där $\phi_1, \phi_2, ..., \phi_n$ är antaganden, även kallat premisser, och ψ är en slutsats som härleds från dessa premisser. Ett bevis av en sekvent består av ett antal steg där man härleder nya formler från premisserna och slutligen kan bevisa slutsatsen. En sekvent är giltig om det finns ett bevis från premisserna till slutsatsen.

Härledning av nya formler sker genom att tillämpa regler på redan givna formler. Det finns axiomatiska regler samt härledda regler. Härledda regler kan bevisas utifrån de axiomatiska reglerna. Under naturlig deduktion kan man använda olika system där olika härledda regler är tillåtna. I denna text följer vi det system som används av Huth och Ryan [1] med en lista av regler i tabell 2.2.

	Introduktion	Elimination
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\wedge_i$	$\dfrac{\phi \wedge \psi}{\phi}\wedge_{e1}$ $\dfrac{\phi \wedge \psi}{\psi}\wedge_{e2}$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\vee_{i1}$ $\dfrac{\psi}{\phi \vee \psi}\vee_{i2}$	$\dfrac{\phi \vee \psi \quad \begin{array}{c}[\phi]\\ \vdots\\ \varphi\end{array} \quad \begin{array}{c}[\psi]\\ \vdots\\ \varphi\end{array}}{\varphi}\vee_e$
$\rightarrow$	$\dfrac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \rightarrow \psi}\rightarrow_i$	$\dfrac{\phi \rightarrow \psi \quad \phi}{\psi}\rightarrow_e$
$\neg$	$\dfrac{\begin{array}{c}[\phi]\\ \vdots\\ \bot\end{array}}{\neg\phi}\neg_i$	$\dfrac{\phi \quad \neg\phi}{\bot}\neg_e$
$\bot$		$\dfrac{\bot}{\phi}\bot_e$
$\neg\neg$	$\dfrac{\phi}{\neg\neg\phi}\neg\neg_i$	
	Extra regler	
	$\dfrac{\begin{array}{c}[\neg\phi]\\ \vdots\\ \bot\end{array}}{\phi}$ PBC	$\dfrac{\neg\neg\phi}{\phi}\neg\neg_e$
	$\dfrac{\phi \rightarrow \psi \quad \neg\psi}{\neg\phi}$ MT	$\dfrac{}{\phi \vee \neg\phi}$ LEM

Tabell 2.2: Regler för introduktion och elimination av konnektiv i satslogik samt extra regler som krävs för den klassiska logiken.

Det finns främst två typer av regler, introduktionsregler och eliminationsregler. Introduktionsregler används för att introducera nya konnektiv och eliminationsregler för att eliminera dessa. Ett exempel på en introduktionsregel är $\wedge_i$, som säger att om vi har ett bevis för ϕ samt ett bevis för ψ har vi ett bevis för formeln $\phi \wedge \psi$, vi kan skriva detta som regeln

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

De motsvarande eliminationsreglerna till $\wedge_i$ är $\wedge_{e1}$ och $\wedge_{e2}$, som säger att om vi har ett bevis för $\phi \wedge \psi$ så har vi ett bevis för ϕ respektive ψ. Detta skrivs som följande

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

Stegen i ett bevis av en sekvent kan antingen vara applicering av regler, likt $\wedge_i$ ovan, eller införandet av nya antaganden. I rapporten och bevisredigeraren använder vi även Huth and Ryan-stilen för att visa dessa bevis. Där skriver man ett radnummer på varje steg och sedan när man applicerar regler så refererar man till dessa radnummer tillsammans med regelförkortningen. Här nedan är beviset av sekventen $A, B \vdash A \wedge B$. På rad 3 används $\wedge_i$ regeln vi visade ovan samt referenser till raderna för bevisen till A samt B.

1.	A	premiss
2.	B	premiss
3.	$A \wedge B$	$\wedge_i$ 1, 2

I ett steg som utgör ett antagande förutsätter man att det finns ett bevis för en formel. När man gör detta antagande öppnas ett nytt kontext där det beviset får användas inom. Kontextet kan sedan stängas vilket markerar när påståendena som bevisades inom kontextet slutar gälla. Individuella påståenden kan alltså inte refereras till utanför sitt kontext, men vissa regler tillämpas på och refererar till hela stängda kontext. För att ett bevis av en sekvent ska vara giltigt krävs det att alla kontext är stängda. En av reglerna som använder dessa antaganden är $\rightarrow_i$ som vi uttrycker nedan som

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

Regeln säger då att om vi antar att vi har ett bevis för ϕ och utifrån detta kan härleda ett bevis för ψ inom samma kontext så har vi ett bevis för $\phi \rightarrow \psi$. I bevis av en sekvent representeras dessa antaganden och kontext med lådor. Denna låda omringar ett kontext och innehåller antagandet samt bevis som kan dras med hjälp utav det antagandet.

1.	$q \to r$	premiss
2.	$p \to q$	antagande
3.	p	antagande
4.	q	$\to_e$ 2, 3
5.	r	$\to_e$ 1, 4
6.	$p \to r$	$\to_i$ 3-5
7.	$(p \to q) \to (p \to r)$	$\to_i$ 2-6

Figur 2.1: Ett bevis av sekventen $q \to r \vdash (p \to q) \to (p \to r)$. Notera hur antaganden har används för att applicera $\to_i$.

I figur 2.1 har vi beviset av sekvensen $q \to r \vdash (p \to q) \to (p \to r)$, som använder ett inbäddat antagande. I detta bevis kan till exempel p och q bara refereras till i innersta kontexten på rad 3-5. Utanför detta kontext på raderna 6 och 7 ser vi inte dem längre, det vill säga att vi inte har ett bevis för dem. Lådan på rad 3-5 representerar underbeviset med antagandet p och slutsatsen r. Genom att avsluta antagandet och därmed stänga lådan kan vi applicera $\to_i$ på lådan. Med hjälp av $\to_i$ regeln och lådan på rad 3-5 har vi skapat ett bevis av $p \to r$.

2.2 Predikatlogik

Predikatlogiken är en utvidgning av satslogiken där man använder termer, predikat och kvantifikatorer för att tydligare kunna presentera mer komplexa påståenden och resonemang. Till exempel kan en formel uttryckas som $P(x)$ där P är predikatet och x är en term. Predikatet betyder en sorts egenskap för termerna, exempelvis kan P representera "är jämn", vilket ger att $P(x)$ betyder att x är jämnt.

I predikatlogik definieras en term som

$$t ::= x \mid c \mid f(t_1, t_2, ..., t_n)$$

där x står för variabler, c konstanter och f som en funktion av n termer t_1,...,t_n.

I predikatlogiken ersätts atomära propositioner från satslogiken med atomära formler. Det finns två typer av atomära formler:

1. $t_1 = t_2$, som för två termer t_1, t_2 säger att de är lika med varandra och är ett speciell sorts predikat.

2. $P(t_1, t_2...t_n)$, för termer t_k där aritet noll är tillåtet och motsvarar propositioner i satslogiken.

Utöver detta införs även två kvantifikatorer eller kvantorer som binder en variabel x i formeln ϕ. Kvantifikatorerna som finns $\forall$ som står för "för alla" samt $\exists$ som står för "det existerar".

$\forall$	$\forall x\phi$	För alla x gäller det att ϕ
$\exists$	$\exists x\phi$	Det existerar ett x sådan att ϕ

En variabel är bunden om den är "uppfångad" av antingen $\forall$ eller $\exists$. Exempelvis i $\forall x(P(x) \land Q(y))$ är x bunden i $P(x)$ för att den finns i $\forall x$ medan y är fri eftersom det inte finns någon kvantifikator som binder den.

I predikatlogiken består välbildade formler av atomära formler, kvantifikatorer samt konnektiv applicerade på välbildade formler. Vi får således följande rekursiva definition av välbildade formler:

$$\phi, \psi ::= \bot \mid P(t_1, t_2, ..., t_n) \mid t_1 = t_2 \mid \neg\phi \mid (\phi \land \psi) \mid (\phi \lor \psi) \mid (\phi \to \psi) \mid (\forall\, x\phi) \mid (\exists\, x\phi)$$

där P är en atomär proposition, t_i termer samt ϕ och ψ är välbildade formler.

2.2.1 Naturlig deduktion för predikatlogik

Med predikatlogiken introduceras fler regler som gäller $\forall$ och $\exists$ kvantifikatorerna samt $=$, vilket ses i tabell 2.3. Förutom steg från satslogiken behöver vi nu även hantera fräscha variabler. En fräsch variabel är en variabel som inte förekommit tidigare i dess kontext. I bevis introduceras fräscha variabler likt antaganden genom att ett nytt kontext öppnas.

	Introduktion	Elimination
$=$	$\dfrac{}{t = t}\,{=_i}$	$\dfrac{t_1 = t_2 \qquad \phi[t_1/x]}{\phi[t_2/x]}\,{=_e}$
$\forall$	$\begin{array}{c}(x_0)\\ \vdots\\ \phi[x_0/x]\end{array}\,{\forall_i}$	$\dfrac{\forall x\phi}{\phi[t/x]}\,{\forall_e}$
$\exists$	$\dfrac{\phi[t/x]}{\exists x\phi}\,{\exists_i}$	$\dfrac{\exists x\phi \qquad \begin{array}{c}(x_0)\\ [\phi[x_0/x]]\\ \vdots\\ \psi\end{array}}{\psi}\,{\exists_e}$

Tabell 2.3: Regler för introduktion och elimination i predikatlogik.

1.	$\exists x \forall y P(x, y)$	premiss
2.	y_0	fräsch
3.	x_0	fräsch
4.	$\forall y P(x_0, y)$	antagande
5.	$P(x_0, y_0)$	$\forall_e$ 4, y_0
6.	$\exists x P(x, y_0)$	$\exists_i$ 5
7.	$\exists x P(x, y_0)$	$\exists_e$ 1, 3-6
8.	$\forall y \exists x P(x, y)$	$\forall_i$ 2-7

Figur 2.2: Bevis av sekventen $\exists x \forall y P(x, y) \vdash \forall y \exists x P(x, y)$.

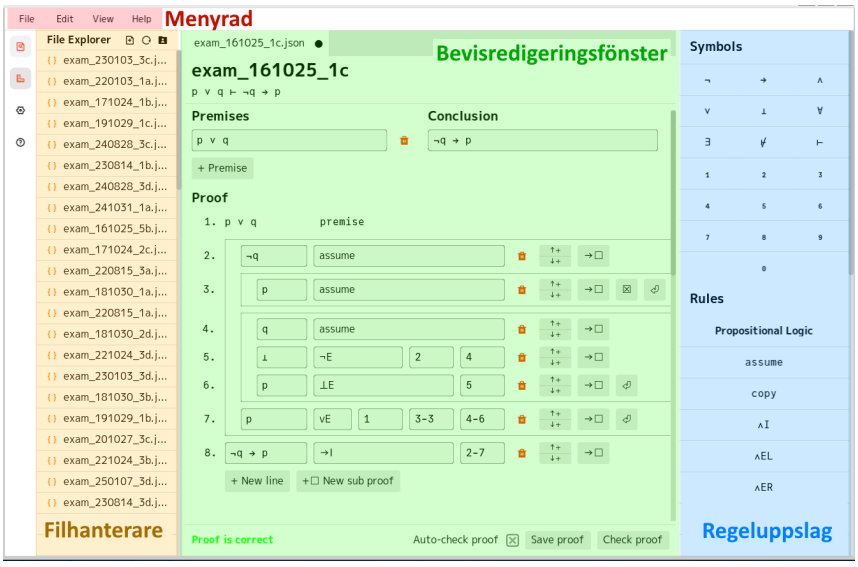
Vissa av reglerna för predikatlogiken använder även en notation $\phi[t/x]$. I uttrycket är x en variabel, t en term samt ϕ en formel. Från detta definieras $\phi[t/x]$ som formeln där man byter ut varje fri instans av x i ϕ med t. Viktig detalj är att detta endast får göras om t är fri för x i ϕ, vilket kan förklaras som att inga fria variabler t blir bundna av en kvantifikator som en följd av substitutionen.

I figur 2.2 visas ett exempel på beviset för sekventen $\exists x \forall y P(x, y) \vdash \forall y \exists x P(x, y)$ med regler för $\forall$ och $\exists$. På rad 2 introducerar vi en fräsch variabel vilket öppnar ett nytt kontext. På rad 3 och 4 ser vi hur ett steg med en fräsch variabel kan följas av ett antagande baserat på den fräscha variabeln. På rad 5 tillämpas regeln $\forall_e$ med referenser till rad 4 och variabeln y_0. Med detta görs utbytet $\phi[y_0/y]$ med $\phi(y) = P(x_0, y)$ där alla förekomster av y byts ut med y_0 och härleds ett bevis för $P(x_0, y_0)$.

3

Bevisredigerarens gränssnitt

Vi har utvecklat ett program som kan installeras på Windows, Linux och Mac OS med hjälp av en installatör[1]. Programmet kräver inte att användaren installerar något annat program eller verktyg för att fungera. Bevisredigeraren ska erbjuda en snabb och intuitiv upplevelse för användaren med funktioner som stödjer och underlättar undervisning och studier. Sammanställningen av funktionaliteter intressant i avseende till ovanstående presenteras nedan.



Figur 3.1: Bevisredigerarens olika delar.

Gränssnittet delar upp arbetsytan i fyra distinkta sektioner: menyrad, filsystemskolumn, bevisredigeringsfönster och ett regeluppslag. Sektionerna är markerade i figur 3.1. Designen divergerar inte meningsfullt från traditionell design så att användare som redan är bekanta med integrerade utvecklingsmiljöer (IDE) snabbt kan lära sig vårt program. Nedan följer en beskrivning av hur programmets struktur ser ut och fungerar.

[1]Programmet är tillgängligt på GitHub: `https://github.com/lambducas/folke`

3.1 Menyrad

Högst upp i fönstret finns en menyrad bestående av fyra knappar. Här finns alla kommandon och åtgärder för programmet samlade.

- **File**: ger alternativen "New Proof", "Save File", "Close File", "Export to LaTeX", "Export to PDF", "Set Working Directory" och "Exit". Detta hanterar filer.
- **Edit**: ger alternativen "Undo", "Redo" och "Validate Proof". Detta agerar på själva bevisredigerandet.
- **View**: ger alternativet "Toggle File Explorer", "Open Preferences" och "Open Guide". Detta öppnar olika menyer.
- **Help**: ger alternativet "Open Guide". Detta öppnar en guide med instruktioner för programmet.

Varje alternativ visar även korresponderande tangentbordsgenväg för att användaren snabbt ska kunna utfärda åtgärden utan att behöva öppna menyn.

Två särskilda alternativ att notera under **File**: är "Export to LaTeX" och "Export to PDF". Dessa tillåter bevis i programmet att kompileras till LaTeX-kod respektive en PDF.

3.2 Filhanteringskolumn

I denna kolumnen kan användaren välja en lokal mapp som arbetskatalog (working directory) samt interagera med innehållet i mappen. Filhanteringskolumnen har följande knappar med respektive funktionalitet:

- **Create new proof**: Skapar en tom bevisfil i den nuvarande arbetskatalogen.
- **Refresh files**: Uppdaterar arbetskatalogen så att den är enig med externa förändringar som kan ha gjorts sedan den öppnades.
- **Set working directory**: Öppnar upp en filutforskare som tillåter en söka upp och välja en mapp som aktuell arbetskatalog.

Detta tillåter smidig navigering, organisering och arkivering av primärt användarens bevisfiler. Det är möjligt att kollapsa denna kolumn för att ge mer plats åt bevisredigeringsfönstret.

3.3 Regeluppslag

Kolumnen till höger om bevisredigeringsfönstret tjänar två funktioner. Den övre delen erbjuder knappar som användaren kan trycka på för att skriva specialtecken i den aktuella textrutan i bevisredigeringsfönstret. Den nedre delen är ett uppslagsverk med alla tillgängliga regler inom naturlig deduktion samt första ordningens logik. Trycker användaren på en regel visas det en kort definition på hur den regeln används och vilka argument den förväntas ta in. Det är möjligt att kollapsa denna kolumn för att ge mer plats åt bevisredigeringsfönstret.

3.4 Bevisredigeringsfönstret

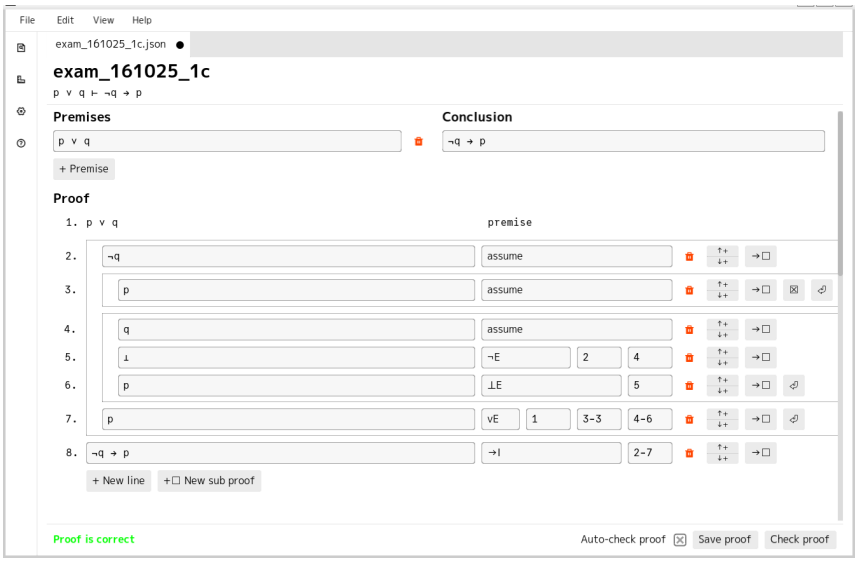
Användaren börjar skriva bevis genom att skapa ett nytt, tomt bevis via menyraden eller med tangentbordsgenvägen *Ctrl+N*. Alternativt går det att öppna sparade bevis från filhanteringskolumnen eller genom att välja ett exempelbevis från menyraden. Beviset visas i bevisredigeringsfönstret i en egen flik. Alla öppnade bevis visas som flikar högst upp i fönstret, vilket tillåter smidig navigering mellan alla öppnade bevis. Här visas även vilka bevis som har ändrats men ännu inte har sparats och varnar när användaren försöker stänga en flik med ett osparat bevis.

När ett bevis har öppnats kan användaren ange eventuella premisser samt slutsats för den sekvens som önskas bevisas, samt ett bevis för sekvensen. I figur 3.2 visas ett exempelbevis. Textfältet högst upp till vänster är där användaren skriver in bevisets premiss. Flera premisser kan skrivas in genom att trycka på *+ Premise* som skapar ett nytt textfält under den senaste tillagda premissen. Till höger om premisserna skriver användaren in bevisets slutsats. Under premisserna skriver sedan användaren själva beviset. Premisserna fylls automatiskt in högst upp i beviset. Användaren kan sedan börja skriva beviset genom att ange ett påstående samt en korrekt regel som ger påståendet tillsammans med eventuella referenser till föregående rader.

Användaren kan använda sig av följande åtgärder för att bland annat infoga rader, infoga underbevis eller ta bort rader:

- **Infoga ny rad:** Infoga en ny rad under nuvarande rad eller i slutet av beviset.
- **Infoga underbevis:** Infoga ett underbevis under nuvarande rad eller omvandla nuvarande rad till ett underbevis.
- **Stäng underbevis:** Stäng nuvarande underbevis och infoga en ny rad under.
- **Ta bort rad:** Ta bort den nuvarande raden.

För att se om hela beviset är korrekt trycker användaren på *Check proof* längst ner till höger eller *Ctrl+R*. Om beviset är inkorrekt visas fel samt eventuella varningar vid raderna de uppstår på. Om användaren skriver in något som inte är tillåtet rödmarkeras raden och användaren uppmanas att korrigera raden. I figur 3.3 visas

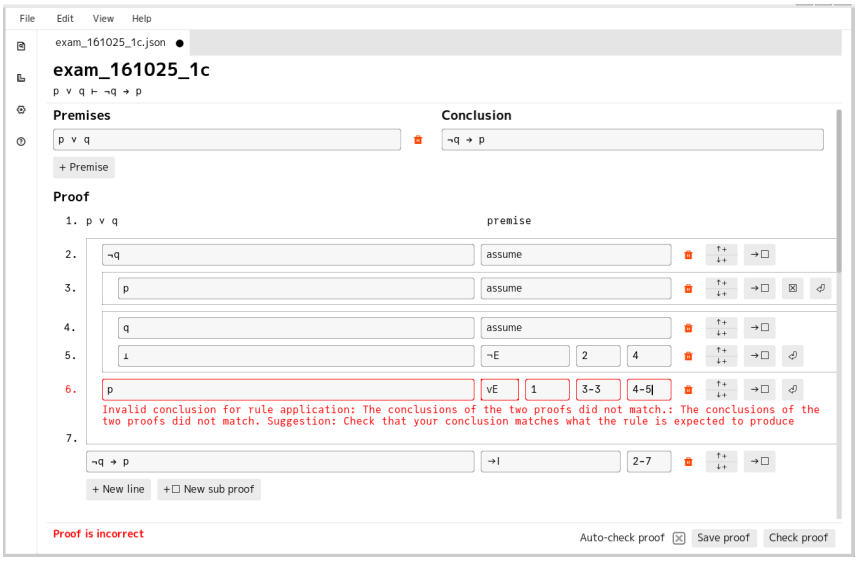


Figur 3.2: Bevisredigeringsfönstret med ett korrekt bevis för $p \vee q \vdash \neg q \rightarrow p$.

ett inkorrekt bevis. Raden där felet uppstår är rödmarkerad och felmeddelandet visas under raden. Otillåtna saker som användaren kan skriva är:

- **Inkorrekt syntax**: Programmet kan inte tolka det användaren skrev in.
- **Okänd regel**: Användaren försöker använda en regel som inte existerar.
- **Felanvänd regel**: Fel regel används för att dra en slutsats.
- **Inkorrekt referens**: Regeln refererar till fel rad eller delbevis.

När alla fel har åtgärdats får användaren ett korrekt bevis och detta visas med grön text ner till vänster i fönstret. Om användaren önskar kan beviset nu även användas som en regel i andra bevis.



Figur 3.3: Ett inkorrekt bevis för $p \vee q \vdash \neg q \to p$. Regeln på rad 6 används på ett otillåtet sätt.

3.5 Anpassa programmet

Bevisredigeraren erbjuder inställningar för att anpassa användarupplevelsen. Följande egenskaper går att anpassa:

- **Text- och fönsterstorlek:** Användaren kan dynamiskt variera textstorleken samt skalningen på hela gränssnittet.
- **Varierbar texttjocklek:** För att göra text enklare att läsa kan användaren välja mellan tunn, normal och tjock texttjocklek.
- **Val av typsnitt:** Användaren kan välja mellan diverse fonter för att göra texten enklare att läsa.
- **Ljus- och mörkerläge:** Välj mellan att ha svart text på ljus bakgrund eller vit text på mörk bakgrund.

4

Design och teknikval

I början av projektet skapades en generell plan för hur programmet skulle designas och det identifierades vilka redan existerande verktyg och bibliotek som behövdes för att genomföra projektet. Projektet separerades i två delar, den första delen är det grafiska gränssnittet och den andra delen är en bevismotor som är ansvarig för att validera bevisen som användaren matar in via användargränssnittet.

4.1 Val av programmeringsspråk

Programmeringsspråk har olika styrkor och svagheter som kan göra att de är mer eller mindre lämpade för olika ändamål. Detta betyder att valet av språk ett viktigt första steg i att utveckla ett digitalt verktyg, i detta fall en bevisredigerare. Bevismotorn och gränssnittet har skilda behov och därför olika krav på vilket programmeringsspråk som bäst passar den tjänst programmet behöver utföra.

Vi valde att använda Haskell för att implementera bevisredigeraren. Beslutet påverkades huvudsakligen av att vi ansåg att den hade ett antal egenskaper som var väl ämnade för utvecklingen av vår bevismotor. Detta eftersom mycket av de algoritmer vi planerade att implementera kändes naturliga att beskriva rekursivt vilket ett funktionellt programmeringsspråk såsom Haskell är väl lämpat för. Vi övervägde flera alternativ för att utveckla det grafiska gränssnittet men landade i att även där använda Haskell. Detta val gjordes framförallt för att undvika behovet av flera olika programmeringsspråk i projektet.

```c
int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) {
        fact *= i;
    }
    return fact;
}
```

```haskell
factorial :: Integral a => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Figur 4.1: Fakultet funktion implementerad i C till vänster och i Haskell till höger.

4.1.1 Funktionell programmering

För funktionella programmeringsspråk såsom Haskell konstruerar man program genom att bygga upp matematiska funktioner som sedan evalueras. Detta till skillnad från imperativa programmeringsspråk där program konstrueras som en sekvens av operationer som sedan utförs. Då Haskell, likt många andra funktionella programmeringsspråk, inte har loopar eller liknande koncept är det vanligt att man definierar funktioner rekursivt likt den högra Haskell-koden i figur 4.1 där "factorial" kallar sig själv. Rekursion är när något definieras i termer av sig själv, vilket i funktionella programmeringsspråk tillsammans med if-satser och liknande kontrollstrukturer kan användas för att uppnå liknande resultat som man vanligtvis skulle uppnå med loopar. Då det ofta är naturligt att beskriva datastrukturer rekursivt är det också naturligt att beskriva algoritmer som arbetar på sådana datastrukturer rekursivt. Sådan kod kan ofta vara estetiskt tillfredsställande men kan leda till vissa tekniska utmaningar beroende på hur de underliggande programmeringsspråket är designat och är därför sällan använt i industrin. Detta är delvis löst i Haskell genom implementation av lat beräkning, vilket betyder att värdet av funktionsanrop inte beräknas förrän resultatet faktiskt behövs. Detta gör att Haskell tillåter arbete på datastrukturer som är större än vad som kan lagras i datorns arbetsminne.

4.1.2 Starkt typade programmeringsspråk

Vi såg även att Haskells typsystem var väl ämnat både för att utföra de typkontroller som vi planerade att stora delar av vår bevismotor skulle bygga på, men även att det skulle göra det enklare för oss att garantera att bevismotorn gav korrekta resultat. Att ett programmeringsspråk har ett starkt typsystem betyder att programmeringsspråket tillåter få, om några, implicita typkonverteringar, det är till exempel inte tillåtet att utföra matematiska operationer med flyttal och booleska värden, eller naturliga tal och strängar. Att typsystemet är statiskt betyder att varje variabel bara kan vara instansierad av en datatyp. Det är till skillnad från dynamiska typsystem där det skulle vara tillåtet att först skriva $x = 1$ och sedan ändra värdet på $x = \text{True}$ sådant att x nu är ett booleskt värde istället för ett heltal. Båda dessa egenskaper är en övervägning mellan att förhindra svåridentifierade problem mot användarvänlighet.

4.1.3 Monader

Vi kände även att Haskells monader skulle vara väl lämpade till att implementera ett robust system för att hantera resultat, varningar och felmeddelanden. En **monad** [7] är en abstrakt struktur som gör det möjligt för en datatyp att bädda in och hantera beräkningar med extra kontext, till exempel felhantering, bieffekter eller icke-determinism, på ett sätt som gör Haskell applicerbart för utveckling av program och som bevarar dess funktionella renhet.

Monaden tillhandahåller två grundläggande operationer:

- `return`: lyfter ett värde in i monadens kontext och på så sätt skapas ett monadiskt värde

- `(>>=)` (*bind*): kedjar ihop två monadiska beräkningar, där utdata från den första matas in i den andra. Detta kan liknas vid imperativa språk, där instruktionerna körs sekventiellt och varje stegs resultat används av nästa

Eftersom Haskell är ett rent funktionellt språk är både `return` och `(>>=)` rena funktioner utan dolda bieffekter. En datatyp som är en instans av `Monad` har implementerat dessa operationer och uppfyller monadlagarna (vänsteridentitet, högeridentitet och associativitet). Under 5.1 går vi närmare in på hur vi inkorporerade och implementerade monader.

För att sekventiellt komponera monadiska operationer utan att direkt använda *bind*-operatorn `(>>=)` tillhandahåller Haskell **do-notation**, vilket är ett syntaktiskt socker som gör koden mer läsbar och imperativt inspirerad [8], [9].

4.2 Design av bevismotorn

Vi utgick från bevismotorn då vi ansåg att den var den mer tekniskt utmanande delen av projektet. Då problemen som en bevismotor måste lösa i många avseenden liknar de problem som en traditionell kompilator för programmeringsspråk löser valde vi att basera vår design på liknande principer. Det är framförallt två problem som vi delar med kompilatorer: den första är att förstå text som en människa har skrivit och den andra är att validera att dess betydelse är korrekt.

4.2.1 Lexikalanalys

För att en kompilator ska kunna omvandla kod till ett exekverbart program behöver den först översätta koden till ett format som är lättare att förstå för datorer. Detta görs oftast i ett antal steg, varav det första är lexikalanalys där irrelevant information såsom kommentarer och överflödiga blanksteg tas bort och koden delas upp i en sekvens av token. Denna pseudokod

```
print("hello world")
```

kan till exempelvis konverteras till sekvensen

```
["print", "(", """, "hello world", """, ")"].
```

När koden väl är konverterad till relevanta token utför kompilatorn en syntaxanalys på dessa token för att matcha dem mot relevanta satser och uttryck. Det är vanligt att dessa representeras med ett abstrakt syntax träd där varje mening representeras av en nod och där meningens delar är nodens barn. Då bevisen som användaren matar in delvis är i textform behöver vi göra samma process för att få bevisen i en maskinläslig form [10].

För att underlätta vårt arbete valde vi att beskriva syntaxen för logiska uttryck i Backus-Naur Form (BNF) vilket är en formell notation som används för att beskriva syntaxen i ett språk, exempelvis programmeringsspråk och formella grammatiksystem. Vi använde sedan BNFC (Backus-Naur Form Converter) [11] för att generera kod för att utföra lexikal- och syntaxanalys enligt våra regler.

BNF och specifikt en LBNF, Labeled Backus-Naur Form, används av BNFC för att definiera syntaxen genom produktionsregler som specificerar hur giltiga uttryck konstrueras. Exempelvis beskriver följande regel hur parametrar definieras som en sekvens av termer:

```
Params. Params ::= "(" [Term] ")";
```

Denna regel används sedan i andra regler, exempelvis för att definiera syntaxen för predikat:

```
Pred. Pred ::= Ident Params;
```

Regler med högre komplexitet kan sedan definieras rekursivt genom att låta regler referera till sig själva, exempelvis som följande grammatiken för enkla logiska formler:

```
AndForm.  Formula ::= Formula "∧" Formula;
OrForm.   Formula ::= Formula "∨" Formula;
PropForm. Formula :: = Ident
```

4.2.2 Semantisk analys

När kompilatorn har genererat ett abstrakt syntaxträd utförs ofta ett antal semantiska analyser på trädet för att kontrollera att trädet är vettigt utan att behöva exekvera programmet. Den vanligaste typen av semantisk analys är vad som kallas en typkontroll där man letar efter fel i koden genom att analysera nodernas typer för att se huruvida de matchar. Hur effektiv en typkontroll är på att hitta problem i program beror till stor del på hur strikt programmets typsystem är. Vi kan till exempel titta på följande pseudokod:

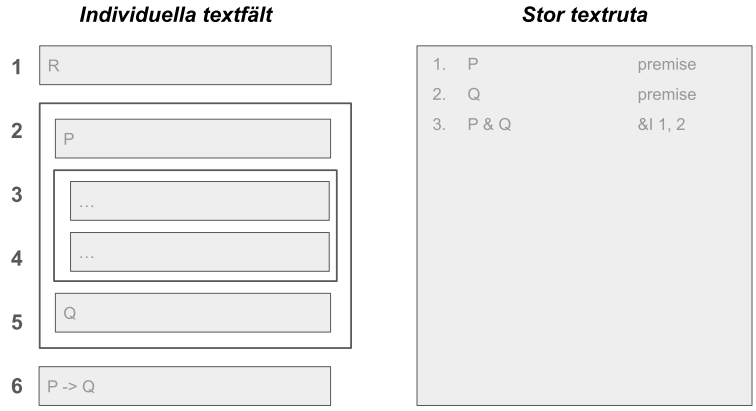
```
x = 1
y = "one"
z = x + y
```

och konstatera att x är ett heltal och y är en sträng, vilket innebär att $x + y$ inte borde vara tillåtet. Eftersom i stort sett alla delar av ett program har typer kan vi på ett liknande sätt resonera kring alla andra delar av ett program [10]. Vi kan använda samma metod för att validera våra bevis genom att behandla de olika formlerna i ett bevis på samma sätt som man behandlar typer i en typkontroll.

Varje steg i beviset har en typ som antingen kan vara en formel eller ett underbevis där antagandet och sista steget motsvarar premissen och slutsatsen. För antaganden, påståenden och regelapplikationer gäller det att deras typ är formeln som användaren har angivit, för regelapplikationer måste resultatet också jämföras med argumenten för att se att deras typer skulle generera det angivna resultatet.

4.3 Design av det grafiska gränssnittet

Som tidigare nämnts informerades vårt val av programmeringsspråk framför allt av bevismotorns behov. Vi valde även att använda Haskell till användargränssnittet för att undvika flera programmeringsspråk i projektet. Vi valde att använda Monomer [12] som ramverk för att konstruera gränssnittet. Monomer är ett Haskell-bibliotek som tillåter skapandet av grafiska användargränssnitt. Det är cross-plattform och stödjer skapandet av gränssnitt för Windows, Linux och Mac OS. Detta görs med hjälp av komponenter som beskriver vad som ska finnas med i gränssnittet samt hur layouten ska se ut. För att beskriva layouten definieras en funktion i Haskell som tar in nuvarande tillstånd av applikationen och returnerar hela programmets utseende. Denna funktion anropas internt av Monomer när applikationen ändras och bara det som har ändrats behöver byggas om och renderas på nytt. Vi ansåg att Monomer var är bra val då det lät oss utveckla för de tre största operativsystemen utan att behöva ändra något i koden. Det fanns även dokumentation som var enkel att följa samt flera exempelprogram att ta inspiration ifrån.

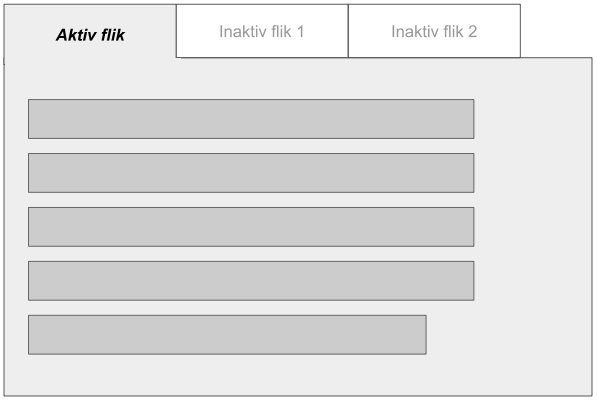


Figur 4.2: Jämförelse mellan individuella textfält och en stor textruta.

När vi väl hade valt Monomer som ramverk behövde vi bestämma hur vi ville designa själva gränssnittet. Denna process informerades av att bevisredigerarens huvudsakliga syfte är att vara ett pedagogiskt hjälpmedel för logikstudenter. Det är därför viktigt att programmet designas så att det är så intuitivt som möjligt att lära sig och använda. Stor del av designen baseras på Nielsens principer för användarvänlig design [13]. Utifrån dessa principer gjorde vi ett antal olika designval för att anpassa användarupplevelsen.

Istället för en stor textruta där användaren matar in hela beviset i form av kod, likt Lean och Agda, valde vi att dela in beviset i rader. I figur 4.2 visas en prototyp av hur raderna strukturerar. Varje rad har flera individuella textfält för formler, regler och dess argument, samt knappar för bland annat ta bort rader, infoga nya rader och skapa delbevis. Nyttan med att implementera bevisskrivandet på detta vis är att det ansågs tillåta för mer intuitiv manipulation av raderna för en mer pedagogisk upplevelse. Formatet är alltså mer vägledande i hur ett bevis generellt brukar struktureras.

Förutom knappar för att hantera raderna kan användaren använda sig av tangentbordsgenvägar för att utföra samma funktioner fast snabbare. Det är även möjligt att infoga specialtecken genom att skriva tecken som är tillgängliga via tangentbordet, till exempel ‘&’ för ‘$\wedge$’, ‘|’ för ‘$\vee$’ och så vidare. Dessa specialtecken går även att välja genom ett klick med muspekaren på respektive symbol i ett uppslagsverk. Det är mycket otympligare att skriva specialtecken med musen än med tangentbordsgenvägar men underlättar för användaren som inte lärt sig alla genvägar utantill än.



Figur 4.3: Fliksystemet tillåter användaren att hantera flera bevis samtidigt.

Samtliga instruktiva hjälpmedel implementerades av den självklara anledningen att så gott som möjligt hjälpa användaren att förstå programmet och komma igång med att skriva bevis. Främst av dessa är välkomstsidan som öppnas första gången programmet startas och ska handleda användaren. Här beskrivs kort hur programmet används, snabblänkar för att skapa ett enkelt bevis samt vilka regler och specialtecken som kan användas. Förutom en välkomstsida finns ett uppslag med symboler och regler som programmet stödjer som alltid visas vid sidan när användaren skriver sitt bevis. Programmet kommer även med flera exempelbevis för att användaren ska kunna se hur bevis skrivs samt hur syntaxen ser ut.

För att användaren inte ska behöva skriva bevis från tomt blad till färdigt bevis, eller ett helt logiskt korrekt bevis i en sittning, gör vi det möjligt att spara inkompletta bevis. Alla bevis som användaren skriver sparas i JSON-format med en skräddarsydd filändelse ".folke" för att kunna skilja på bevis och andra JSON-filer. JSON är ett dataformat som är lätt för både människor och maskiner att läsa och som lämpar sig bra för nästlade datastrukturer [14]. JSON valdes då det är ett av de vanligaste formaten för att representera trädliknande strukturer. Det kan därför användas för att bevara en bevisstruktur som användargränssnittet redan vet om, så som underbevis/lådor, detta medan formler och regler, som användaren skriver in i textfält, sparas som strängar. JSON valdes även för att det finns färdigbyggda parsers för JSON i Haskell. Vi använder oss av *aeson* [15] som är ett bibliotek som smidigt konverterar Haskells datatyper till och från JSON-textsträngar.

Vidare har det gjorts så att användaren snabbt ska kunna hantera flera bevis samtidigt utan att det blir rörigt. Detta genom att implementera ett fliksystem. Fliksystemet gör det möjligt att snabbt byta mellan öppnade bevis utan att behöva ha flera fönster öppnade och utspridda på skrivbordet. I figur 4.3 visas hur flikarna kan se ut. För att användaren snabbt ska kunna öppna och byta mellan flera bevis som ligger i en mapp kan mappen öppnas i bevisredigeraren. Denna mapp blir då programmets *arbetskatalog* och visas hela tiden för att snabbt komma åt bevis.

Vi gör det även möjligt för användaren att exportera sina bevis till LaTeX-dokument. Detta för att användare ska ha möjlighet att författa bevis med redigerarens stöd, men sedan även kunna omvandla det som skrevs till ett mer traditionellt och tillgängligt format vid inlämning av uppgifter eller rapportskrivning.

Användaren erbjuds även förmågan att anpassa sin användarupplevelse genom att ändra inställningar för textstorlek, färgtema, typsnitt och så vidare. Detta är i synnerhet för att underlätta för användare med synsvårigheter. Då detta är ett potentiellt verktyg för högskolestudenter vill vi underlätta för studenter med varierade svårigheter och bakgrunder. Typsnittet *OpenDyslexic* är inkluderat för dyslektiker. Typsnittet är open-source och utvecklat av dyslektiker för att göra text enklare att läsa [16]. Dessa inställningar sparas i bakgrunden och bevaras till nästa gång programmet används. Även öppnade bevis och osparade ändringar sparas internt när programmet stängs för att sedan återappliceras när programmet öppnas igen. Öppnade bevis skrivs alltså inte över när programmet stängs utan koms ihåg av programmet.

5

Tekniska resultat

I detta kapitel av rapporten går vi igenom relevanta delar av projektets tekniska implementationer.

5.1 Bevismotorn

Bevismotorn har en design som efterliknar de första stegen i en traditionell programmeringsspråkstolk. I det första steget läser bevismotorn in en JSON-fil som innehåller ett bevis i ett delvis strukturerat format. I steget som följer används BNFC-genererade parsers för att bearbeta de delarna av formatet som är sparade i text och konverterar det hela till ett abstrakt syntaxträd för beviset.

Eftersom det abstrakta syntaxträdet konstrueras sekventiellt och i samma ordning som beviset är skrivet genomförs samtidigt en typkontroll av det resulterande syntaxträdet. Det innebär att vi granskar de olika delarna i beviset så att de överensstämmer med varandra och är logiskt konsekventa. Ingångspunkten för typkontrollen implementeras på följande sätt i Haskell:

```haskell
checkSequentFE :: Env -> FE.FESequent -> Result Proof
checkSequentFE env sequent = do
  -- Typkontrollera premisser
  premsT  <- checkPremsFE env (_premises sequent)
  -- Typkontrollera slutsatsen
  concT   <- checkFormFE env (_conclusion sequent)
  -- Typkontrollera bevisstegen
  proofT  <- checkProofFE env (sequentSteps sequent) 1
  -- Sätt ihop det förväntade resultatet
  let expected = Proof [] premsT concT

  -- Jämför det förväntade beviset med det som användaren skrev
  if proofT == expected
    then return expected
    else throwMismatchedFormulaError env concT (getConclusion proofT)
```

Funktionen `checkSequentFE` avgör om användarens bevis är korrekt genom att stegvis typkontrollera premisser, slutsats och bevisstegen. Typkontroll i kontexten av bevisredigeraren handlar om att säkerställa både syntaktisk välformning och logisk korrekthet. För premisser och slutsats kontrolleras att formlerna följer korrekt syntax. För bevisstegen verifieras bland annat att varje härledning följer en giltig regelapplikation och att variabelbindningar respekteras.

Kontrollen resulterar i ett typat bevisobjekt som representerar hela sekventen. Därefter görs en slutgiltig jämförelse som säkerställer att det konstruerade beviset är semantiskt ekvivalent med det förväntade bevisobjektet, vilket bekräftar att beviset faktiskt härleder den angivna slutsatsen från premisserna. Med semantisk ekvivalens menas här att det bevis som användaren konstruerat genom bevisstegen faktiskt motsvarar den sekvent som användaren från början skrev ut och avsåg bevisa.

Varje bevis representeras i det abstrakta syntaxträdet av en sekvent-nod bestående av ett antal premisser, en slutsats och stegen som krävs för att gå från premisserna till slutsatsen:

```
data FESequent = FESequent {
  _premises :: [FEFormula],
  _conclusion :: FEFormula,
  _steps :: [FEStep]
}

type FEFormula = Text

data FEStep
  = Line {
    _statement :: Text,
    _rule :: Text,
    _usedArguments :: Int,
    _arguments :: [Text]
  }
  | SubProof [FEStep]
```

Observera att i Haskell-koden ovan gäller det att både premisserna och slutsatsen är i sin tur formler. Vi kan sedan kontrollera att beviset är korrekt genom att kontrollera om alla steg är korrekta.

För att representera de olika stegen i ett bevis som användaren kan tänkas behöva göra har vi ett flertal steg som representeras av olika sorters noder i vårt abstrakta syntaxträd. Varje steg-nod kan antingen evalueras för att få en term, formel eller ett underbevis vilket vi ser som dess typ. Resultatet av steget sparas i miljön med en etikett som motsvar antingen ett radnummer i beviset eller ett intervall av radnummer ifall steget är ett underbevis.

De olika typerna av steg kan delas upp på följande sätt:

- **Deklarationer av premisser och antaganden**: För att premisser och antaganden ska kunna användas i beviset måste de deklareras. Bevismotorn behandlar premisser och antaganden för det mesta som samma sak. Skillnaden är att premisser endast får deklareras i rotbeviset och måste matcha premisserna i sekventen, medan antaganden endast kan deklareras i underbevis. Det är senare formeln som angivits som premiss eller antagande som är nodens typ.

- **Deklarationer av fräscha variabler**: Det är endast tillåtet i underbevis och alla kraven från logik för att en variabel ska vara fräsch appliceras. Nodens typ är en term, specifikt en term med aritet noll.

- **Lådor**: När användaren använder en låda i sitt bevis så representeras den av en underbevis-nod som precis som en sekvent består av en lista av steg och delar nästan all sin implementation med rotbevis. Ett underbevis har en typ som består av dess premisser, fräscha variabler och slutsats.

- **Regelapplikation**: Noden som representerar användningen av en regel består av en identifierare som identifierar vilken regel som appliceras, argumenten som regeln utförs på, och det svar som användare har angivit. Argumenten anges som referenser till tidigare steg via deras radnummer och kan vara formler eller underbevis. Nodens typ är den formel som resulterar av applikationen av regeln om den överensstämmer med det angivna svaret.

Bevismotorn går igenom beviset rekursivt i samma mönster som en djupet-först-sökning och det som behövs lagras mellan noder i trädet sparas i ett miljötillstånd. Varje nod i det abstrakta syntaxträdet motsvaras av en funktion som tar in en nod och returnerar nodens typ. Vi använder datatypen `Env` för miljöhantering, det vill säga bevisets föränderliga kontext som innefattar allt från spårandet av premisser och referenser till omfång av bevisets variabler. Vi definierar följande datatyp i Haskell:

```
data Env = Env {
    prems      :: Formulas,
    depth      :: Integer,
    fresh      :: Terms,
    refs       :: Map Ref ArgTup,
    rules      :: Map String (Env -> [ArgTup] -> Formula -> Result Formula),
    user_rules :: Map String UDefRule,
    pos        :: Refs,
    rule       :: String,
    bound      :: Map String (),
    ids        :: Map String IDType
}
```

Varje formel representeras av ett träd av formler, predikat och termer. Varje logisk operation representeras av en egen typ av nod med ett barn för varje operand. Exempelvis så representeras $\forall x P(x)$ av en för-alla nod med två barn, de första en term med aritet noll och den andra ett predikat med aritet ett.

```
data Formula = Pred Predicate
             | And  Formula Formula
             | Or   Formula Formula
             | Impl Formula Formula
             | Eq   Term    Term
             | All  Term    Formula
             | Some Term    Formula
             | Not  Formula
             | Bot
             | Nil
```

Varje instans av ett predikat i beviset representeras av en predikat-nod som har ett antal termer som dess argument, vi representerar påståenden som predikat med aritet noll.

```
data Predicate = Predicate String [Term]
```

Termer är representerade på samma sätt som predikat och precis som att påståenden är predikat med aritet noll så är variabler funktioner med aritet noll.

```
data Term = Term String [Term]
```

Det är intressant att nämna att vi för predikat inte har en separat representation för påståenden och för termer inte separerar på funktioner och variabler. Detta då de på många ställen i logiken är utbytbara, på de ställen som det finns krav på till exempel att en term måste vara en variabel så som för kvantifikatorer så kontrollerar vi via mönstermatchning att termerna har atriet noll. Det är även värt att nämna att bevismotorn inte har stöd för konstanter.

De olika noderna i det abstrakta syntaxträdet är implementerade som Haskell-datatyper och vi använder oss mycket av Haskells typsystem och rekursion tillsammans med mönstermatchning för att bearbeta bevisen. För varje typ, som `Formula`, `Term`, eller `Predicate`, finns det en eller flera motsvarande funktioner för att typkontrollera noden.

I bevismotorn finns det inbyggd stöd för samtliga regler från tabell 2.2 och 2.3. Utöver de inbyggda reglerna så finns det även möjlighet för användaren att använda redan författade bevis som nya användardefinierad regel. När en användardefinierad regel används går programmet igenom varje premiss nod för nod och jämför med motsvarande argument för att kontrollera att de matchar varandra. Jämförelsen bryr sig inte om vilka namn termerna och predikaten har utan jämför bara att de har samma struktur. I och med att de inbyggda reglerna är implementerade direkt i koden kan dessa regler ha beteenden som inte är möjliga i användardefinierade regler såsom fräscha variabler i lådor och variabelsubstitution.

5.1.1 Felhantering och typen `Result`

För att bygga upp den komplexa felhantering som krävs i en bevisredigerare, designade vi följande datastruktur:

```
data Result a =
    Ok [Warning] a
  | Err [Warning] Env Error
```

Denna typ utgör grunden för vår felhantering och möjliggör både detaljerad felrapportering och ackumulering av varningar. När en beräkning lyckas, returneras värdet tillsammans med eventuella varningar. När en beräkning misslyckas returneras ett strukturerat felmeddelande tillsammans med miljötillståndet vid felpunkten.

Det som gör denna datastruktur särskilt effektiv för bevisverifiering är dess förmåga att:

- Bevara kontextuell information (genom `Env`)
- Tillhandahålla rika felmeddelanden (via `Error`-strukturen)
- Ackumulera varningar även genom lyckade beräkningar
- Möjliggöra komposition av komplexa verifieringssekvenser

Genom att implementera `Monad`-typklassen för denna datatyp kunde vi skapa en elegant lösning för att hantera den komplexa verifieringskedja som krävs vid bevisgranskning.

5.1.2 `Monad`-implementation för `Result`-typen

En av nyckelkomponenterna i bevisredigerarens felhanteringssystem är implementationen av `Monad`-typklassen för `Result`:

```
instance Monad Result where

    return :: a -> Result a
    return x = Ok [] x

    (>>=) :: Result a -> (a -> Result b) -> Result b
    (Ok warns x) >>= f = case f x of
        Ok newWarns y -> Ok (warns ++ newWarns) y
        Err newWarns env err -> Err (warns ++ newWarns) env err
    (Err warns env err) >>= _ = Err warns env err
```

- **return**: I implementationen definierar vi `return`, vilket lyfter ett värde till ett lyckat resultat. Detta innebär att `return x` skapar ett `Ok [] x` - ett lyckat resultat utan några varningar. Denna implementation säkerställer att enkla värden kan integreras i `Result`-baserade beräkningar.

- **(»=) (*bind*)**: Denna operation är hjärtat i vår felhantering och har två huvudsakliga beteenden:

  - När tillämpat på ett lyckat resultat `(Ok warns x)`: Den applicerar funktionen `f` på värdet `x` och kombinerar varningarna från båda beräkningarna, oavsett om `f x` lyckas eller misslyckas. Detta garanterar att inga varningar går förlorade i beräkningskedjan.

  - När tillämpat på ett misslyckande `(Err warns env err)`: Den ignorerar funktionen `f` och propagerar felet med dess tillhörande miljö och varningar. Detta implementerar "kortslutningslogiken" där ett fel avbryter hela beräkningskedjan.

5.2 Användargränssnittet

Användargränssnittet använder, likt bevismotorn, datastrukturen `FESequent` för att representera bevis. När användaren gör ändringar i beviset, till exempel genom att ändra texten i ett textfält, genererar Monomer olika *event* beroende på vad i beviset som har ändrats. Från dessa event genereras en ny `FESequent` med den nya datan vi får från Monomer. Slutligen genereras bevisredigeringsfönstret på nytt, dynamiskt utifrån den nya datan.

För att behålla alla ändringar användaren har gjort till beviset sparar vi hela editorns tillstånd till en intern JSON-fil. Vi skapar först en datastruktur med all data som vi vill spara:

```
data PersistentState = PersistentState {
  _windowMode :: MainWindowState,
  _workingDir :: Maybe FilePath,
  _currentFile :: Maybe FilePath,
  _openFiles :: [FilePath],
  _fileExplorerOpen :: Bool,
  _rulesSidebarOpen :: Bool,
  ...
}
```

Vi kan sedan, med hjälp av *Aeson*, automatiskt koda och avkoda datastrukturen:

```
$(deriveJSON defaultOptions ''PersistentState)

parseProofToJSON :: FESequent -> T.Text
parseProofToJSON = toStrict . toLazyText . encodePrettyToTextBuilder
```

Funktionen `parseProofToJSON` genererar en fint formatterad JSON-sträng. Den ser till att alla specialtecken kodas korrekt. JSON-strängen skrivs sedan till en fil när användaren stänger ner programmet. När användaren öppnar editorn igen läses filen in, avkodas och resultatet används för att återskapa editorn i samma tillstånd som den lämnades.

5.3 Testning

För att säkerställa att projektet uppnår våra mål vad det gäller logisk korrekthet och användarvänlighet har vi spenderat en stor del av projektet på att testa både gränssnittet och bevismotorn. Testandet av gränssnittet gjordes framförallt genom att gruppen internt använde programmet för att befatta olika bevis och noterade eventuella smärtpunkter som stöttes på. Vi genomförde även en del liknade tester med vår handledare för att få ett perspektiv från någon som är mer erfaren i logik.

För att testa bevismotorn byggde vi ett automatiskt testsystem i HUnit [17], vilket är ett bibliotek för att skriva testsystem i Haskell. Testsystemet består delvis av traditionella unit-tester i de fallen när det ansågs lämpligt. Dessa fall inkluderar bland annat funktioner som kräver ytterligare säkerhet kring korrekthet. Utöver detta finns det även ett system som kör en förberedd lista av färdigskrivna bevis genom programmet för att kolla om den ger det förväntade resultatet. Detta system kombinerades senare med testningen av gränssnittet då det var i ett användbart skick. Detta tillät oss att spara bevisen som vi skapade i det grafiska gränssnittet och sedan använda de som en del av testsystemet. Bevisen vi använde kom från ett antal olika källor: både validerade bevis från gamla tentamina i kursen *Logic in Computer Science* på Chalmers och böcker bland annat *Logic in Computer Science* [1] men även sådana vi skrev själva både för att testa specifika saker men också generella bevis.

5.4 Kommunikation mellan gränssnittet och bevismotorn

Kommunikation mellan det grafiska gränssnittet och bevismotorn sker också via JSON. Det är upp till parsern att hantera formlerna och reglerna som skickas från gränssnittet i form av strängar på ett korrekt sätt. Både gränssnittet och parsern använder biblioteket *aeson* för att automatiskt konvertera till och från JSON, vilket gör att vi kan använda Haskell för att definiera och standardisera vårt gemensamma dataformat. Vi använder `FESequent` som ett gemensamt dataformat för kommunikation mellan alla delar av programmet.

Användaren skickar beviset som behöver valideras tillsammans med flaggor från gränssnittet till parsern. Detta görs via kanaler för att tillåta asynkron kommunikation mellan de två olika komponenterna. Detta är viktigt då det inte finns några garantier på hur snabbt bevismotorn kan validera bevisen, vilket skulle kunna leda till att användargränssnittet hänger sig medan det väntar på svar från bevismotorn. Bevismotorn svarar sedan så fort den är färdig med att validera beviset. Svaret som skickas innehåller huruvida beviset var korrekt eller ej, eventuella felmeddelanden, varningar och tipsmeddelanden.

6

Diskussion

I detta kapitel presenteras reflektioner över arbetet gällande både tillvägagångssätt och resultatet. Utöver detta kommer även vidare utveckling beröras, både för potentiella uppdateringar av den nuvarande bevisredigeraren men även framtida projekt i liknande mån.

6.1 Bevismotorn

Bevismotorn stödjer samtliga relevanta delar av första ordningens logik samt ger tydliga och pedagogiska felmeddelanden. Haskell lämpade sig väldigt bra för att kunna hantera logiken i koden. Genom Haskells strikta typstruktur och monader var det lätt att koda programmet samt blev koden välstrukturerad och lättläst.

Programmets interna tillstånd kapslas in i en explicit miljöstruktur, vilket möjliggör rena funktionella uppdateringar och belyser hur funktionell programmering kan användas för att spåra tillstånd utan bieffekter. Vid varje verifieringssteg uppdateras miljön för att reflektera definierade termer, bindningar och referenser. Detta förenklar både hantering av variabelbindningar och kontext, särskilt i närvaro av kvantifikatorer där skillnaden mellan bundna och fria variabler är kritisk. Denna metod underlättar vidare hanteringen av delbevis, där en utökad kontext används för att rekursivt verifiera delbevis utan att kompromissa med det globala tillståndet. Utifrån en teknisk vinkel och i strävan att skapa en bevisredigerare med optimal prestanda finner vi att det finns utrymme för förbättring i datastrukturen för miljötillstånd. I den nuvarande implementationen förlänger eller förkortar vi miljöobjekt (variabler instansierade av `Env`-strukturen) vilket inte är optimalt. Genom att istället utforma datastrukturen så att en helt ny miljö skapas vid varje nytt underbevis är det möjligt att förbättra prestandan avsevärt. Med den nuvarande implementationen av datatypen kopieras hela miljön, vars tidskomplexitet är linjär med hänsyn till storleken av miljötillståndet. En implementation av miljöhantering som är förbättrad och minimerar kopieringar, och som istället beaktar faktumet att vi inte behöver lagra hela miljön vid varje nytt underbevis, skulle potentiellt kunna leda till förbättringar i tidskomplexitet. Detta eftersom man istället hade räknat ut skillnaden mellan tillståndet av föregående underbevis och nuvarande underbevis och skapat en ny miljö baserat på denna skillnaden.

Bevisverifieringsprocessen är indelad i en strukturell valideringsfas (syntaxanalys) och en logisk verifieringsfas (typkontroll), speglande matematikerns arbetsflöde där välformade uttryck först säkerställs innan den semantiska giltigheten prövas. Genom att separera syntax- och typkontroll från regler för härledning uppnås både tydlighet i felrapporteringen och flexibilitet att vidareutveckla systemet med nya deduktiva regler. BNFC användes för parsergenerering, vilket snabbt gav en fungerande grund för syntaxanalys, men visade sig senare begränsande när mer detaljerade felmeddelanden önskades. Detta då BNFC inte är kapabel av att generera specifika felmeddelanden utan bara kan peka på var felet inträffade, vilket visade sig vara svårtolkat av även relativt erfarna användare. För precision i syntaxfelshanteringen krävs därför en mer specialiserad parser i framtida iterationer.

Som tidigare nämnt i 5.3 skrevs enhetstester för olika funktioner inom bevismotorn. Dessa tester täcker dock inte hela bevismotorns implementering utan snarare bedömdes bara funktionernas korrekthet. Vidare finner vi att fler enhetstester hade behövts läggas till för att utöka säkerheten av bevismotorns korrekthet. Trots detta anser vi att bevismotorn implementerar logik på ett korrekt sätt. Detta på grund av den omfattande testningen som gjordes på de hundratals uppgifter vi hämtade från tentamina och diverse böcker.

Sammanfattningsvis har det funktionella programmeringsparadigmet, exemplifierat genom Haskell och monadisk felhantering, visat sig väl lämpat för implementation av en logikbaserad bevismotor. Den uppnådda kodkvaliteten och modulariteten underlättar både underhåll och vidareutveckling, medan erfarenheten av BNFC:s begränsningar pekar på vikten av anpassade verktyg för avancerad felrapportering och förbättrad användarupplevelse. I kommande arbete rekommenderas att fördjupa parserkomponenten för att möjliggöra ännu mer kontextuellt precisa felmeddelanden och därigenom stärka systemets pedagogiska potential. Vidare finner man starka skäl till att utöka testsystemet för att inkludera ett mer omfattande och rigoröst genomförande av enhetstestningen. Detta betonas i allt högre grad särskilt med tanke på det pedagogiska temat av bevisredigeraren. Mjukvaran bör ha som förutsättning att den beter sig korrekt och inte lär ut fel saker till användaren.

6.2 Tekniska förbättringsområden

Med tid som den främsta begränsade resursen underlättar redan existerande programmeringsbibliotek betydligt i utvecklandet av program i allmänhet. Det kan dock uppkomma särskilda problem då ramverket som bibliotek erbjuder kan begränsa vissa utvecklingsmöjligheter för programmet. Det gäller alltså att överväga om vissa beroenden mer förhindrar än hjälper utvecklingen.

6.2.1 Haskell och Monomers prestation

Bevisredigeraren hanterar korta och medelstora bevis väl men vid längre bevis kan prestandan försämras. Även vid snabb inmatning av text kan programmet hacka till. Programmet är fortfarande fullt användbart men användarupplevelsen försämras. Detta kan bero på att Haskell är bristfällig på en fundamental nivå när det kommer till att skildra grafik. En annan anledning kan vara att specifikt Monomer inte är gjord för avancerade gränssnitt med flera hundra textfält och knappar som visas samtidigt. I båda fallen kan detta vara nog underlag att avråda Monomer som gränssnitt för framtida projekt.

Monomer saknade viss funktionalitet som projektet krävde. För att kunna göra ändringar och lägga till funktionalitet skapades en lokal kopia av Monomer. Det krävdes flera ändringar i Monomers källkod för att, bland annat, göra det möjligt för textfält att byta ut specialtecken medans användaren skrev in text, lägga till funktionalitet för att mata in text genom att trycka på knappar med musen, fixa buggar som försämrar användarvänligheten och för att få programmet att fungera även på Windows och Mac OS. Det bör noteras att möjligheten att förändra källkod är en särskild styrka hos open-source och i den meningen gjorde Monomer sig bra tillgänglig.

En annan nackdel med Monomer är att installationen på Windows inte fungerade som avsett. Det krävdes mycket arbete och med hjälp av foruminlägg av andra som haft samma problem lyckades vi till slut få programmet att fungera på Windows.

6.2.2 Bättre parser

Som tidigare nämnt i 6.1 begränsade vårt val av att använda BNFC hur informativa eror meddelanden vi kunde generera för syntax fel. Det skulle därför vara intressant att utforska möjligheten att skriva en skräddarsydd parser. Detta skulle göra det möjligt för bevismotorn att ge mycket mer specifika varningar såsom till exempel att det saknas parenteser i en formel, otillåtna karaktärer eller att en formel är felskriven. Detta skulle dock kräva en hel del arbete och göra det svårare att göra snabba ändringar i syntaxen, särskilt då användandet BNFC gör det mycket lättare att skriva en helt deterministisk parser.

6.3 Stöd för flera operatorer och termer

Vi ser att programmet bättre skulle uppfylla sitt syfte som en pedagogiskt hjälpmedel om den hade stöd för en större andel av bevis som potentiella studenter kan förväntats stöta på. Det skulle vara värdefullt att lägga till extra funktionalitet som gör det enklare för användaren att representera vissa koncept som tekniskt sätt stöds av programmet men där det inte uttryckligen implementerade.

Ett exempel på detta skulle vara ekvivalens, det vill säga formler på typen $A \leftrightarrow B$. I nuläget saknas tecknet $\leftrightarrow$ vilket gör det omständigt att att representera sådana påståenden. Stöd för en $\leftrightarrow$ operator i vår grammatik och tillhörande elimination och introduktions regler skulle underlätta för användaren.

Ett annat område där det finns möjlighet för förbättring skulle vara i vår implementation av termer, som i nuläget bara stödjer variabler och funktioner men inte konstanter. Det skulle också vara fördelaktigt att stödja matematiska operator så som $+$ och $-$. Detta skulle göra det möjligt för användaren att lättare skriva och validera matematiska bevis, vilket är ett av de få områdena där studenter kan förvänta sig stötta på logiska bevis och skulle därför möjliggöra mer praktiska exempel.

6.4 Automatisk bevislösare

Eftersom automatisk bevislösning inte ingick i projektets omfång valde vi att inte implementera ett sådant system. Avsaknaden av detta innebär att programmet inte kan ge förutsägande rekommendationer till användaren. Det skulle därför vara intressant att implementera någon form av automatisk bevislösare, troligtvis en enklare variant baserad på en heuristisk algoritm då denna troligtvis skulle leda till resultat som mer efterliknar bevis skrivna av människor. Detta skulle tillåta programmet att ge mer insiktsfull hjälp till användaren som att föreslå lämpliga strategier för att lösa problem, varna om användare är ute på ett sidospår eller i en återvändsgränd.

6.5 Tillgänglighetsåtgärder

I varje steg måste man ha i åtanke huruvida en förändring eller implementering lämpar sig för sin målgrupp. Bevisredigerarens tillgänglighet för studenter är centralt för projektets syfte. Projektet hade alltid studenter i fokus, men det menar inte att alla möjliga implementeringar och åtgärder som tjänade det ändamålet realiserades. Detta kan bero på till exempel tekniska begränsningar eller att åtgärderna befinner sig utanför projektets ramar.

6.5.1 Användarstudie

Det skulle tjäna projektets intresse att genomföra någon form av användarstudie för att samla in information om huruvida programmet är av nytta för den planerade målgruppen. Detta skulle vara extra intressant då vi inte har genomfört någon sådan studie eftersom det inte var den del av projektets omfång. Detta anses vara ett självklart nästa steg om projektet ska fortsättas i framtiden. Det skulle även vara intressant att se hur inte bara studenter utan även lärare och kursansvariga skulle använda programmet som en del av en logikkurs.

6.5.2 Design av användargränssnitt

Utan att ha följt någon formell designmall har gränssnittets utseende inspirerats av det vi i gruppen förväntar oss att programvara inom detta område ska se ut och presenteras. Detta refererar till mer samtida designad mjukvara, som Sublime Text och Visual Studio Code, med ett fokus på stilrenhet, simplism, och monoton färgläggning, med mer färgglad ikonografi för kontrast. Att designen skulle vara bristfällig på grund av detta är möjligt. Projektet specialiserade sig inte i det rent grafiska, att ha någon tredje part ge professionella åsikter om problemet eller möjligtvis designa om utseenden på gränssnittet kan därför övervägas i hopp om att förbättra produkten.

6.5.3 Onlinefunktionalitet

Bevisredigeraren som en webbaserad tjänst skulle tillåta för ytterligare funktionalitet som underlättar för studenter som vill skriva bevis. Detta skulle beröra implementationer såsom molnlagring och kooperativt arbete. Förutom detta är en webbtjänst man inte behöver ladda ned oftast smidigare då den är tillgänglig på alla smartapparater som har tillgång till en webbläsare och internet. Huvudanledningen till att detta inte byggdes på var att vi såg en viss risk att några eventuella webbtjänster kopplade till projektet inte skulle fortsätta vara tillgängliga i framtiden. Bland annat därför valde vi att designa programmet som en traditionell inbyggd app. Med detta sagt hade en webbaserad tjänst varit eftersträvansvärt i ett framtida arbete med liknande syfte som detta.

6.5.4 Mobilapplikation

För tillfället stödjer bevisredigeraren bara operativsystem för datorer. Detta då det kändes som det primära digitala verktyget som projektets målgrupp använder för sina studier. Men att programmet bara är tillgänglig på datorer är inte en ideal teknologisk begränsning. Det kan mycket väl förekomma att studenter i behov inte har lätt tillgång till datorer för deras studier och det påverkar bevisredigerarens effektiva funktionalitet åt det negativa. En variant av programmet som en mobilapplikation skulle vara ett effektivt sätt att åtgärda detta. Förutom det faktum att mobiltelefoner idag är vanligare än persondatorer är mängden studenter som saknar tillgång till både mobiltelefon och dator ytterst liten. Att bevisredigeraren är ett digitalt verktyg kommer alltid vara en begränsning som kräver tillgång till en smart maskin, men att i framtiden stödja både datorer och mobiler lär vara den mest effektiva och övertäckande åtgärden för bevisredigerarens tillgänglighet.

Förutom en fråga om tillgänglighet skulle en mobilapplikation även kunna handla om bekvämlighet. Att ha en mobilapplikation skulle tillåta för mer bitvisa studiesessioner. Att kunna ta upp telefonen på till exempel bussen till skolan och försöka sig på något bevis är en kvalitativ upplevelse som definitivt skulle underlätta studenter i deras studier. Detta skulle kräva en större omdesign, potentiellt ett helt nytt gränssnitt för syftet, med grafik och funktioner mer anpassade för mobilen.

6.6 Distribution och öppen källkod

Detta projekt är tänkt som ett pedagogiskt tillskott för logikstudenter, och att göra projektet fritt tillgängligt via publicering på exempelvis GitHub stärker just detta syfte på flera sätt.

Studenter ska kunna uppnå transparens och lärande genom att kunna inspektera och förstå hela verifieringsmotorn, allt från typkontroll till felhantering. På detta sätt fås insikter i hur formell logik kan implementeras. Tack vare kodens modulära struktur blir det enkelt att plocka ut och anpassa olika komponenter, till exempel typkontroll eller deduktiva regler, för att skapa egna övningar. Dessutom kan externa aktörer bidra till projektets utveckling genom att rapportera buggar, föreslå nya regler eller förbättra dokumentationen.

6.7 Slutsats

Vi anser att bevisredigeraren tjänar sitt syfte, som är att erbjuda lämpligt stöd för masterstudenter som lär sig skriva bevis inom naturlig deduktion. Som en bevisredigerare är bevismotorn fungerande och kan rätta bevis korrekt. Som ett pedagogiskt verktyg erbjuder programmet felmeddelanden som hjälper användaren att skriva bevisen på rätt sätt. Utöver detta existerar även mer bekvämlighetsinriktade åtgärder som filsystem för sparandet av bevis, som förhoppningsvis ska underlätta ytterligare för användare som lär sig skriva bevis. Det finns dock förbättringsområden i projektets produktutveckling, bland annat i hur det grafiska gränssnittets val av programmeringsspråk och bibliotek påverkade dess utveckling. Vidare finns det framtida utvecklingsområden för bevisredigeraren i sin helhet som skulle kunna förbättra både dess tillgänglighet och användarupplevelse. Bevisredigeraren kan fungera som en utgångspunkt för framtida projekt med liknande syfte som denna. Det återstår att se om bevisredigeraren, Folke, uppnår sitt faktiska mål att komma till användning för masterstudenter.

Referenser

[1] M. Huth och M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004, ISBN: 978-0-521-54310-1.

[2] Open Logic Project, *Natural deduction proof editor and checker*, 2025. URL: https://proofs.openlogicproject.org/.

[3] *Agda*, Accessed: 2025-04-19. URL: https://wiki.portal.chalmers.se/agda.

[4] T. Coquand och G. Huet, *The Rocq Prover*, 2025. URL: https://rocq-prover.org/.

[5] Lean FRO, *Lean: Programming Language and Theorem Prover*, Accessed: 2025-04-19. URL: https://lean-lang.org/.

[6] G. Gentzen, "Untersuchungen über das logische Schließen. I," *Mathematische Zeitschrift*, årg. 39, nr 1, s. 176–210, dec. 1935, ISSN: 1432-1823. DOI: 10.1007/BF01201353. URL: https://doi.org/10.1007/BF01201353.

[7] J. Newbern, *All About Monads*, 2011. URL: https://wiki.haskell.org/All_About_Monads#Maybe_a_monad.

[8] Haskell Wiki contributors, *Syntactic sugar*, https://wiki.haskell.org/Syntactic_sugar, 2006.

[9] Haskell Wiki contributors, *Haskell IO for Imperative Programmers*, https://wiki.haskell.org/index.php?title=Haskell_IO_for_Imperative_Programmers, 2009.

[10] A. Aho, M. Lam, R. Sethi och J. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education Limited, 2014, ISBN: 978-1-292-02434-9.

[11] A. Abel, *BNFC*, Accessed: 2025-05-01. URL: https://hackage.haskell.org/package/BNFC.

[12] F. Vallarino, *Monomer*, 2025. URL: https://github.com/fjvallarino/monomer.

[13] J. Nielsen, "Enhancing the explanatory power of usability heuristics," i *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '94, Boston, Massachusetts, USA: Association for Computing Machinery, 1994, s. 152–158, ISBN: 0897916506. DOI: 10.1145/191666.191729. URL: https://doi.org/10.1145/191666.191729.

[14] D. Crockford, *Introducing JSON*, Accessed: 2025-04-19, 2001. URL: https://www.json.org/json-en.html.

[15] B. O'Sullivan, *aeson: Fast JSON parsing and encoding*, Accessed: 2025-04-19, 2025. URL: https://hackage.haskell.org/package/aeson.

[16] A. Gonzalez, *OpenDyslexic*. URL: https://opendyslexic.org/.
[17] D. Herington, *HUnit: A unit testing framework for Haskell*, Accessed: 2025-05-29. URL: https://hackage.haskell.org/package/HUnit.