UNIVERSITETET I OSLO
Institutt for informatikk

Modallogikk som spørrespråk for en deduktiv database

Masteroppgave

Anders Skolseg
Bruvik

9. mars 2006

Takk

Takk til Arild Waaler for god veiledning, til Johan W. Klüwer for faglig bistand og for bildene på side 20 og 21 og til alle på logikkseminaret INF5170 for godt faglig påfyll.
En spesiell takk til Bente for fantastisk støtte under hele prosjektet.
Takk til Trym, som kom til verden akkurat i tide til å få med seg den siste innspurten, og til mine foreldre, for all støtte underveis.
Takk til alle venner i Realistforeningen og andre studentforeninger.

Innhold

1 Introduksjon **1**
1.1 Innledning 1

2 Bakgrunnsstoff **3**
2.1 Eksempeldatabase 3
2.2 Språk 4
2.3 Førsteordens syntaks 4
2.4 En deduktiv database i førsteordens logikk 6
2.5 Skranker 10

3 Æ som en database **12**
3.1 $\mathcal{L}_F$: Faktaspråket 13
3.2 $\mathcal{L}_C$: Skrankespråket 15
3.3 $\mathcal{L}_{PL}$: Utsagnslogisk ekstensjon av $\mathcal{L}_F$ og $\mathcal{L}_C$ 16
3.4 En deduktiv database i Æ 17
3.5 Tell og Ask 23
3.6 Oppsummering 26

4 Implementasjon i Maude **27**
4.1 L_I - Sekventkalkyle 27
4.2 Algoritme for sekventkalkylen 29
4.3 Introduksjon til Maude 35
4.4 Implementasjon 36

5 Videre arbeid **54**

A Fullstendig kode **59**

Kodeeksempler

4.1 Definisjon av sorter 37
4.2 Definisjon av språket 38
4.3 Definisjon av modale operatorer 38
4.4 Mengder av formler 39
4.5 Regel for $R\neg$ 40
4.6 regelen $R\wedge$ representert i Maude 40
4.7 tynningsregel representert i Maude 40
4.8 Regel for modalitetene representert i Maude 42
4.9 Hjelpefunksjonen removeModal 42
4.10 Stack av Result 44
4.11 Funksjonen performProofSearch 44
4.12 Uttdrag fra proofSearch 45
4.13 Anvendelse av en modal regel i proofSearch 46
4.14 Hjelpefunksjonen applyRules 46
4.15 Regelen applyRule 47
4.16 Funksjonen oneStepModal 48
4.17 Identitetsregelen 48
4.18 Metaregel for aksiom 49
4.19 Regel for conv-aksiomet 49
4.20 Metaregel for conv-aksiomet 49
4.21 Negativ aksiomsjekk 50
A.1 Hele programmet 59

Kapittel 1

Introduksjon

1.1 Innledning

En database er en samling av fakta (dataelementer) lagret i et datasystem på en systematisk måte, slik at det er mulig å hente ut data (svare på spørsmål). Et datasystem som er laget for å lagre og hente ut data fra en database kalles et databasesystem, eller "Database management system" (DBMS) på engelsk[1]. En viktig egenskap med alle databasesystemer er at data lagres i mindre elementer, kalt atomære fakta eller atomer. Når man henter ut data settes disse atomene sammen til mer komplekse enheter igjen. Det er i denne sammenhengen viktig at systemet ikke trekker usunne slutninger, samtidig som det henter mest mulig informasjon ut fra systemet. Operasjonen med å hente data ut fra en database kalles en *spørring*, mens operasjonen som legger til data i en database kalles gjerne å *fastslå* fakta. Språket som brukes til å stille spørsmål og fastslå fakta kalles for et *spørrespråk*.

Tradisjonelt brukes *relasjonsmodellen* som teoretisk rammeverk for studier av databaser. Et alternativ er å modellere databasene i 1. ordens logikk, og i en slik modell kan man se på fakta som *aksiomer* ("antatte sannheter"), mens søk etter fakta i databasen blir et *bevissøk*. Relasjonsmodellen kan representeres som en 1.ordens teori [14], men førsteordens logikk har større uttrykkskraft enn relasjonsmodellen, og man kan derfor gå lenger og tillate mer komplekse formler i databasen. Dermed kan man også lage mer komplekse databaser. Eksempler på dette er databaser som kan inneholde slutningsregler sammen med fakta, databaser som kan inneholde ufullstendig informasjon

[1] I praksis brukes ofte begrepet database både om samlingen av fakta, og om systemet som organiserer databasen (DBMS).

eller databaser som inneholder negativ informasjon. Slike databaser kaller vi for *deduktive databaser*. Man kan skille mellom ulike klasser av deduktive databaser etter hvilke egenskaper de har.

Det vil også være naturlig å ta steget videre, og se på hvordan andre logikker med alternative egenskaper egner seg som spørrespråk for en database. I oppgaven har jeg tatt utgangspunkt i logikken Æ [17], som er en utsagnslogisk modallogikk i "only knowing" - familien introdusert av Levesque, [9]. Æ har mulighet for å uttrykke defaulter direkte i språket. Ved hjelp av en omskriving av en formel som representerer oppfatninger kan man få et direkte uttrykk for modellene. Æ har dessuten en naturlig utvidelse til flere agenter.

Oppgaven består av to deler. I den første delen ser jeg på hvordan et språk for en deduktiv database kan representeres i Æ, disse kaller jeg for Æ-databaser. Jeg ser på enkelte sentrale problemstillinger omkring deduktive databaser, og ser på hvordan en Æ-database kan håndtere disse problemene. I den andre delen av oppgaven har jeg sett hvordan en teorembeviser for en beslektet logikk, kalt L_I, kan implementeres. I motsetning til tradisjonelle deduktive databaser har jeg tatt utgangspunkt i et enkelt språk for representasjon av fakta. Dette gjør at man kan bygge en Æ-database med en relasjons-database i bunn, og et fremtidig mål vil være å bygge en prototype på en slik database. Dette kan gjøres ved å kombinere en relasjonsdatabase med en søkemotor basert på teorembeviseren jeg har implementert. I kapittel 3 ser jeg på en en utgave av Æ med bare en agent. For en slik database vil det være tilstrekkelig med en SAT-teorembeviser, men for en multiagent-utvidelse vil ikke dette være tilstrekkelig. For en multi-agent utgave av en Æ-database vil teorembeviseren jeg beskriver i kapittel 4 kunne brukes som utgangspunkt for en implementasjon.

Sentrale problemstillinger i studier av deduktive databaser er behandling av *negative data*, *nullverdi-problemet* og *ufullstendige data*, med det siste menes tilfellet der vi vet at $P(a) \vee P(b)$ er sann, men vi vet ikke om $P(a)$ er sann, om $P(b)$ er sann, eller om begge er sanne. I tillegg har vi problemstillinger som effektivitet og anvendelsesområder. Ved design av et databasesystem vil det alltid være en avveining mellom uttrykkskraft og effektivitet. Dessuten er det ønskelig med et hensiktsmessig spørrespråk, som gjør det enkelt å finne tilbake til informasjon som er lagret i systemet.

Til slutt har jeg skissert noen anvendelser av systemet, spesielt rettet mot semantisk web.

Kapittel 2

Bakgrunnsstoff

2.1 Eksempeldatabase

I denne delen skal jeg introdusere en database som skal fungere som et eksempel. Eksempeldatabasen skal inneholde informasjon om studenter og hvilke kurs de tar, og den skal bygges ut med mer informasjon gjennom resten av teksten. I utgangspunktet skal databasen inneholde en tabell med oversikt over en rekke kurs, studenter og kurskoder som i figur 2.1.

Student	StudentID
ole	A
kari	B
per	C

STUDENTER

StudentId	KursKode
A	300
B	400
C	100

STUDENTKURS

KursKode	Navn
100	Logikk
200	Matte
300	Fysikk
400	Kinesisk

KURS

Figur 2.1: Eksempeldatabasen

Her har vi en database som består av tre tabeller. En av tabellene inneholder en oversikt over studenter, en over en mengde kurs og den siste tabellen inneholder oversikt over hvilke studenter som tar hvilke

kurs. Selvsagt ville det vært naturlig å legge til mer informasjon, som for eksempel personnummer eller annen unik identifikasjon i Student-tabellen og navn på kurset i Kurs-tabellen, men for enkelhets skyld gjør jeg ikke det her.

Jeg skal legge til noen krav til databasen, og det første er et krav som sier at alle studenter og kurs som er nevnt i *Studentkurs*-tabellen også skal forekomme i henholdsvis *Student-* og *Kurs*-tabellen. Så skal jeg kreve at alle navn er unike, og at alle kurs skal ha unik kurskode. Dette kan vi gjøre i en relasjonsdatabase ved å si at kolonnen for studenter skal være en primærnøkkel for student-tabellen, og at kolonnen for kurs skal være en primærnøkkel for kurs-tabellen. Senere skal jeg legge til noen andre krav og begrensinger til databasen, i tillegg til at den kommer til å bli utvidet noe.

2.2 Språk

Deduktive databaser er representert som teorier i førsteordens logikk. Jeg bruker derfor klassisk førsteordens logikk til å introdusere deduktive databaser, men vil senere introdusere andre språk som gir økt uttrykkskraft. Førsteordens logikk er uavgjørbar, men for anvendelser på databaser er vi avhengige av avgjørbarhet. Dette blir i praksis løst ved å begrense hvilken form vi tillater på klausulene vi bruker til å representere data i databasen, og dermed redusere uttrykkskraften i språket. Det er vanlig å skille mellom ulike klasser av deduktive databaser ut fra hvilke klausuler vi tillater i språket.

2.3 Førsteordens syntaks

Språket består av en uendelig tellbar mengde av variabler, konstantsymboler, funksjonssymboler og relasjonssymboler. Funksjonssymbolene og relasjonssymbolene har tilordnet en *aritet* som er et naturlig tall ≥ 1. I tillegg har vi de logiske konnektivene $\neg, \wedge, \supset, \equiv$, kvantoren $\forall$ samt parenteser. Symbolene $\exists$, $\vee$ og $\subset$ kan defineres som forkortelser på vanlig måte. Termspråket består av konstanter, variabler og funksjonssymboler; Dersom $t_1, \ldots, t_n$ er termer og f er et funksjonssymbol med aritet n, så er $f(t_1, \ldots, t_n)$ en term. Termer uten variabler kalles for *grunntermer*.

Formler bygges opp av relasjonssymboler og termer. Dersom P er et relasjonssymbol med aritet n og $t_1, \ldots, t_n$ er termer, så er $P(t_1, \ldots, t_n)$

en *atomær formel*, og dersom ϕ og ψ er formler så er $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \supset \psi)$ formler. Dersom ϕ er en formel og x er en variabel så er $\forall x \phi$ en formel. Vi kaller variabelen x for *bundet* i formelen $\forall x \varphi$. En variabel som ikke er bundet er *fri*. Dersom en formel ikke inneholder noen frie variabler kaller vi formelen for *lukket*. En atomær formel eller negasjonen av en atomær formel kalles et *literal*, og noen ganger kalles en atomær formel for et *positivt literal*.

I forbindelse med databaser er det ofte naturlig å snakke om funksjonsfrie formler, det vil si at vi tillater kun konstanter og variabler som termer. I en database vil det normalt ikke være noe problem å begrense seg til et endelig antall konstanter, siden domenene som databasen skal representere normalt består av endelig mange elementer. Dersom vi legger inn disse to begrensingene oppnår vi at vi får et endelig antall termer, noe som er en stor fordel for den praktiske anvendelsen av databaser.

Normalformer og klausuler. Ved behandling av databaser er det ofte hensiktsmessig å skrive om formlene til en *normalform*.

En lukket formel har *preneks normalform* dersom alle kvantorer forekommer først i formelen. En preneks formel er på *skolem normalform* dersom alle eksistensielt kvantifiserte variabler er erstattet med vilkårlige funksjoner som avhenger av alle universelt kvantifiserte formler som forekommer før dem i i formelen. Disse funksjonene kalles for *skolemfunksjoner*. En skolemfunksjon med 0 argumenter kalles for en *skolemkonstant*.

En *klausul* er en disjunksjon av literaler, der alle variablene er implisitt universelt kvantifisert. Vi kan dermed eliminere alle kvantorer fra formelen, og formelen (klausulen) kan dermed skrives på formen:

$$\neg A_1 \vee \ldots \vee \neg A_m \vee B_1 \vee \ldots \vee B_n \tag{2.1}$$

eller ekvivalent:

$$A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \vee \ldots \vee B_n \tag{2.2}$$

der A'ene og B'ene er positive literaler. Dersom n er 0 eller 1 kaller vi klausulen for en *hornklausul*. Dersom både m og n er 0 kaller vi klausulen for en *tom* klausul. En klausul der B er større enn 1 kalles en *disjunktiv* klausul. En klausul uten variabler kaller vi en *grunn* klausul. Enhver lukket formel kan skrives om til klausul-form (og bevare oppfyllbarhet).

2.4 En deduktiv database i førsteordens logikk

Det er vanlig å representere deduktive databaser som en teori i førsteordens logikk, og derfor introduserer jeg databaser representert i førsteordens logikk i dette kapittelet.

Det finnes mange måter å lage en teoretisk beskrivelse av en database på, og den vanligste er *relasjonsmodellen* til Codd (se for eksempel [14]). Her er databasen representert som en mengde tabeller slik som det er gjort i eksempelet i 2.1, og det er vanligvis brukt en *relasjonsalgebra* som teoretisk modell. Men det går også an å bruke andre representasjonsmodeller for en database, for eksempel kan det være naturlig å representere tabellene fra relasjonsdatabasen som en mengde grunne førsteordens formler eller klausuler, der hver tabell er representert med ett relasjonssymbol. Spørringer, påstander og skranker uttrykkes som åpne eller lukkede førsteordens klausuler.

Eksempel 1 (Eksempeldatabasen på klausulform)
Student-tabellen fra eksempelet i 2.1 kan da representeres som klausulene:

$$
\begin{aligned}
Student(ole, A) &\leftarrow \\
Student(kari, B) &\leftarrow \\
Student(per, C) &\leftarrow
\end{aligned}
$$

De andre tabellene kan representeres på tilsvarende måte.

Ullman viser i [14] at Datalog med negasjon[1] har større uttrykkskraft enn relasjonsalgebra. Han viser at mengden av funksjoner som kan uttrykkes i relasjonsalgebra er ekvivalent med mengden av funksjoner vi kan uttrykke i datalog med negasjon, men med regler begrenset til “safe rules”, dvs. ingen rekursjon og kun “stratified” negasjon, dermed kan vi representere enhver relasjonsdatabase som en deduktiv database, men ikke nødvendigvis det motsatte, for eksempel kan transitiv tillukning enkelt uttrykkes i en datalog-database, men dette kan ikke uttrykkes i relasjonsalgebra. Det er derfor mulig å bruke

[1]Datalog er et spørrespråk for databaser basert på prinsipper fra logisk programmering, og implementerer *bestemte* deduktive databaser, se [2] for en introduksjon. Grunnen til at vi trenger negasjon er at relasjonsalgebra har én ikkemonoton operator (mengdedifferanse), og datalog uten negasjon er monoton, og har dermed ikke en tilsvarende funksjon.

førsteordens logikk som en alternativ teoretisk modell for databaser, men førsteordens logikk har økt uttrykkskraft i forhold til relasjonsmodellen. Blant annet er det enkelt å uttrykke transitiv tillukning i førsteordens logikk.

Dersom man ser på klausuler som har økt uttrykkskraft i forhold til en relasjonsdatabase ser man at det vil være enkelt å lage klausuler som representere slutningsregler i tillegg til de som representerer data eller skranker. Databaser som inneholder klausuler som representerer slutningsregler kalles *deduktive databaser*. Ulempen med å tillate vilkårlige førsteordens formler er selvsagt at man risikerer at man får uavgjørbarhet, det vil finnes setninger (eller "fakta") som man ikke kan avgjøre om er sanne eller usanne i databasen. I tillegg vil man for praktisk bruk ofte ønske å garantere en viss effektivitet for søk. Derfor må vi legge visse føringer på hvilke formler vi tillater i databasen, og dette avgjøres av forholdet mellom ønsket uttrykkskraft og ønsket ytelse for databasen. Ofte har man lagt en forholdsvis sterk begrensing på klausulene for å oppnå unike minimale modeller for databasene, og denne klassen av databaser er de *bestemte* deduktive databasene. I slike databaser tillater man ikke disjunksjon i hodene på klausulene (disjunktive regler) eller negasjon. Det oppstår et fundamentalt skille når man tillater disjunktive regler eller "klassisk negasjon". I dette tilfellet har vi ikke lenger en unik minimal modell for databasen.

Vi skiller altså mellom såkalte *bestemte databaser*, der alle klausulene er horn-klausuler, og *disjunktive databaser* (også kalt *ubestemte databaser*). I en bestemt database har databasen én unik minimal modell, mens i en disjunktiv database kan vi ha flere ikke ekvivalente minimale modeller. Resultatet er at svaret på én spørring i en bestemt database er alltid unikt og entydig, mens i en disjunktiv database kan vi ha flere "korrekte" svar på en spørring. Dette kalles ufullstendig informasjon.

Eksempel 2 (Database med flere minimale modeller)
Vi ønsker å utvide eksempeldatabasen med informasjon om hvilke studenter som er høyeregrads– og laveregrads studenter. I tillegg er det slik at alle studentene er enten høyere eller laveregradsstudenter. Vi legger derfor til en klausul i databasen:

$$LGrad(x) \vee HGrad(x) \leftarrow Student(x, y) \tag{2.3}$$

som forteller oss at alle studenter er enten på lavere grad eller på høyere-grad. I tillegg legger vi til følgende fakta, her representert ved en tabell , som sier at Kari er en høyeregradsstudent:

$$\frac{\textit{\textbf{HGrad}}}{\textit{kari}}$$

Eller på klausulform:

$$HGrad(kari) \leftarrow$$

Denne databasen vil ha to minimale modeller, én hvor Ole er høyeregradsstudent og én hvor Ole er laveregradsstudent.

Slike databaser kalles Disjunktive deduktive databaser og “extended disjunctive databases” (med negasjon i tillegg til disjunksjon i hodet på regelen). Det er verdt å merke seg at i databaser som lager ufullstendig informasjon kommer førsteordens logikk til kort som spørrespråk, det finnes spørringer som er naturlige å uttrykke som ikke kan uttrykkes i førsteordens logikk. Her kommer for eksempel autoepistemisk logikk inn som spørrespråk, eller Levesques $\mathcal{KL}$ [8]. Dette er språk som har større uttrykkskraft og som kan uttrykke spørringer som omhandler hvilke data som finnes i databasen.

En deduktiv database består altså av en mengde data og en mengde slutningsregler. Databasen (databasesystemet) er i stand til å selv utlede nye data ved hjelp av eksisterende data samt slutningsregler. De gitte dataene kalles ekstensjonen eller den ekstensjonale databasen, mens data som kan utledes kalles den intensjonale databasen. Både data og slutningsregler er representert som klausuler.

Definisjon 1 (Deduktiv database)
En database DB består av et par $\langle W, \Gamma \rangle$, der $W = \{c_1, \ldots, c_n\}$ er en endelig mengde konstanter kalt domenet *eller* universet*, og Γ er en mengde funksjonsfrie 1.ordens klausuler der domenet for konstantene er begrenset til W.*

Klausulene i Γ kan representere både fakta, integritetsskranker og slutningsregler. Ved å begrense hvilken form vi tillater på klausulene kan vi også begrense oss til databaser med unike minimale modeller, og lavere kompleksitet på søkealgoritmene. Det kan deles inn i ulike klasser av databaser ut fra hvilke klausuler vi tillater, se [5] for en oversikt. To viktige grupper av databaser er de *bestemte* databasene, og de *disjunktive* databasene.

Ved å kreve at klausulene ikke inneholder funksjoner oppnår vi at spørringer får endelige og eksplisitte svar, siden dette begrenser oss til endelig termunivers, noe som gir oss endelige modeller.

Definisjon 2 (Spørring)
En spørring mot en database DB *er en formel* $W(x_1, \ldots, x_n)$ *der* $x_1, \ldots, x_n$ *er frie variabler eller konstantsymboler. Svaret på en slik spørring består av mengden av tupler* $W(e_1, \ldots, e_n)$ *slik at* $DB \vdash W(e_1, \ldots, e_n)$*. Svaret kan eventuelt oppgis som* `true` *dersom svaret er ikketomt og* `false` *dersom svaret er en tom mengde.*

En database er tradisjonelt lukket under tre forskjellige antakelser. De to første er *antakelsen om unike navn* – vi antar at individer i en database er forskjellig dersom de har forskjellige navn, og *domenelukkingsantagelsen* (domain closure assumption), som sier at det ikke finnes andre individer enn de som forekommer i databasen. Den siste antagelsen er antagelsen om lukket verden.

Antagelsen om åpen verden. Dersom vi bruker vanlig førsteordens bevissøk for å søke etter fakta i en database, vil det være slik at dersom vi har gitt en database DB og en spørring Q, hvor DB en database som er definert som i definisjon 1, og Q er en førsteordens formel, så vil de eneste svarene vi finner være de vi finner ved bevis av Q gitt DB som aksiomer. Dette kalles antagelsen om *åpen verden*.

Eksempel 3 (Antagelsen om åpen verden)
Spørringen $\neg Studentkurs(ole, 1)$ *vil ikke gi svaret* `true` *anvendt på eksempelet i 2.1, selv om dette vil være en rimelig antagelse ut fra informasjonen som er gitt. Dette skyldes at* $\neg Studentkurs(ole, 1)$ *ikke kan bevises ut fra aksiomene gitt av databasen.*

For en database er antagelsen om åpen verden lite hensiktsmessig, siden mengden av negative data normalt er veldig mye større enn de positive dataene. Dersom negative data skulle vært gitt eksplisitt til systemet ville mengden av disse vært svært mye større enn mengden av positive data. Det ville rett og slett ha blitt for mye data til at det ville vært effektivt å registrere data, og å foreta søk.

Antagelsen om lukket verden. Siden en antagelse om åpen verden ikke egner seg for en database, er det i stedet hensiktsmessig at en database er lukket under *antagelsen om lukket verden* (Closed world assumption, CWA)[2]. Dette er antagelsen om at fakta er usanne dersom man ikke kan bevise at de er sanne, eller med andre ord: Man antar at $\neg R(t_1, \ldots, t_n)$ er sann dersom vi ikke finner tuppelet $t_1, \ldots, t_n$ i R.

[2]Antagelsen om lukket verden kalles også “konvensjonen for negativ informasjon”.

Eksempel 4 (Antagelsen om lukket verden)
Spørringen $\neg Studentkurs(A, 100)$ *vil gi svaret* `true` *anvendt på eksempelet i 2.1. Siden tuppelet* $\{A, 100\}$ *ikke finnes i tabellen Studentkurs så kan man anta at* $\neg Studentkurs(A, 100)$ *er sann, ut fra antagelsen om lukket verden.*

Vi innfører derfor antagelsen om lukket verden, som sier at dersom en grunn atomær formel $P(c_1, \ldots, c_n)$ ikke kan utledes fra ekstensjonen av databasen og slutningsreglene, så kan vi slutte $\neg P(c_1, \ldots, c_n)$

En definitiv deduktiv database har alltid én (unik) minimal modell, og den er alltid konsistent under CWA, men dette er ikke nødvendigvis tilfelle for en disjunktiv database. Det er mulig å utvide CWA til en generell CWA, introdusert av Minker i [12]. Se forøvrig [13] for en grundigere introduksjon til CWA.

Intensjonal og ekstensjonal database. Vi skiller mellom den *intensjonale* og den *ekstensjonale* databasen. Den ekstensjonale databasen (EDB) er mengden av alle fakta som er eksplisitt gitt til systemet, og består av en mengde klausuler på formen

$$a \leftarrow$$

der a kan være en konstant eller en atomær formel.

"A predicate whose relation is stored in the database is called an *extensional database* (EDB) relation, while one defined by logical rules is called an *intensional database* (IDB) relation. We assume that each predicate symbol either denotes an EDB relation or an IDB relation, but not both" ([14]). Den *intensjonale* databasen (IDB) er mengden av alle data som kan utledes ved hjelp av klausulene i databasen, og som ikke er en del av den intensjonale databasen.

En database som kun består av grunne klausuler blir en ren ekstensjonal database. Det er vanlig å legge en føring som sier at et relasjonssymbol enten angir en EDB-relasjon, eller en IDB-relasjon, men ikke begge deler.

2.5 Skranker

Skranker er virkemiddelet som brukes til å begrense lovlige tilstander til en database. Vi kan for eksempel ønske å begrense lovlige verdier i en relasjon, kreve at enkelte verdier skal være unike, eller vi kan ønske at konstanter som er inneholdt i en relasjon også skal være inneholdt i

en annen relasjon. I en relasjonsdatabase hvor man tillater nullverdier[3] kan det være ønskelig å ikke tillate nullverdier i bestemte kolonner i en tabell, for eksempel dersom en kolonne skal kunne refereres til i en annen tabell. Virkemiddelet vi bruker for å sette slike begrensinger kaller vi *skranker*. Vi skiller mellom flere typer skranker. *Primærnøkler* brukes til å gi en unik identifikator til en tabell, vi sier at alle verdiene i en kolonne i tabellen (dersom vi ser for oss relasjonen som en tabell) skal være unik. Dersom en primærnøkkel er satt kreves det også at det ikke er noen nullverdier i denne kolonnen. En annen type skranke er *fremmednøkler*. Når vi introduserer en fremmednøkkel krever vi at én verdi i én relasjon i databasen også skal være inneholdt i en annen relasjon. Vi kan også ønske å innføre andre typer begrensinger på databasen, for eksempel kan vi innføre restriksjoner på hvilke verdier som er tillatte. Skranker kan uttrykkes som vilkårlige førsteordens formler:

Definisjon 3 (Integritetsskranke)
En database DB oppfyller en integritetsskranke w, formulert som en formel i førsteordens logikk dersom $DB \cup w$ er konsistent.

Skranker kan representeres i databasespråket som førsteordens formler. For eksempel kan vi for en gitt relasjon $R(t_1, \ldots, t_k, \ldots, t_m)$ ønske at t_k skal være en primærnøkkel. Dette kan vi uttrykke ved formelen

$$\forall \overrightarrow{t} \forall \overrightarrow{u} (R(t_1, \ldots, t_k, \ldots, t_m) \land R(u_1, \ldots, u_k, \ldots, u_m) \land t_k = u_k) \supset (\overrightarrow{t} = \overrightarrow{u})$$

Fremmednøkler kan uttrykkes ved formler på formen

$$\forall t_k (R(t_1, \ldots, t_k, \ldots, t_n) \supset \exists u_l (S(u_1, \ldots, u_l, \ldots, u_m) \land t_k = u_l))$$

som sier at for én bestemt t_k i relasjonen R så skal denne verdien forekomme i relasjonen S også.

[3]En nullverdi er et element i en database som ikke har noen verdi

Kapittel 3

Æ som en database

Her skal jeg definere et hierarki av språk for en deduktiv database som baserer seg på logikken Æ. Dette språket har en mer fleksibel håndtering av antagelsen om lukket verden enn tradisjonelle språk for deduktive databaser, og det har større uttrykkskraft ved at man kan lagre defaulter direkte i databasespråket. Æ er introdusert i artikkelen [17]. Min introduksjon til Æ er basert på denne artikkelen. Språkhierarkiet er basert på arbeid gjort i [7].

Jeg skal også vise hvordan et spørrespråk for databasen, basert på en funksjonell operator TELL kan representeres. Jeg ser på hvilken uttrykkskraft dette gir, og hvordan problemstillinger som CWA, ufullstendig informasjon og negative data kan håndteres i en slik database. I tillegg skal jeg skissere noen utvidede muligheter en slik database gir.

Språket Æ er et utsagnslogisk språk, men for å få mulighet til en hensiktsmessig representasjon av data i en database er det nødvendig med et språk som inneholder relasjoner. For en database trenger man bare å snakke om endelige databaser, og det er dermed mulig å lage et utsagnslogisk språk som en ekstensjon av datarepresentasjonsspråket, hvor man med en enkel omskriving oppnår enn representasjon av databasen i utsagnslogikk. Dette språket kalles $\mathcal{L}_{PL}$.

Datarepresentasjonsspråket består av to språk, $\mathcal{L}_F$ som er et språk for å representere fakta, og $\mathcal{L}_C$ som er et språk for å representere skranker. Disse språkene korresponderer med henholsvis tabeller og skranker i en relasjonsdatabase. Formlene i $\mathcal{L}_C$ tolkes som den karakteristiske formelen ρ som bestemmer det logiske rommet, og disse begrenser forestillingsrommet.

I tillegg gir Æ mulighet til å uttrykke “defaulter” og mer generelle skranker direkte i databasen. Det vil også være mulig å få en mer

fleksibel behandling av negasjon ved feiling og antagelsen om lukket verden enn det som er mulig i andre databasesystemer.

3.1 $\mathcal{L}_F$: Faktaspråket

Språket $\mathcal{L}_F$, som definert i [7], er et språk for representasjon av data eller fakta i databasen, og språket brukes til å representere den ekstensjonale databasen. En database uttrykt i dette språket tilsvarer også en mengde tabeller i en relasjonsdatabase, og en formel i $\mathcal{L}_F$ vil korrespondere med en enkel rad i en tabell i en relasjonsdatabase.

Definisjon 4 (Faktaspråket – $\mathcal{L}_F$)
$\mathcal{L}_F$ består av en mengde konstanter*, C, og en mengde* sorter*:*

$$S_1, \ldots, S_i \subseteq C$$

Det kreves at alle konstantene har tilordnet en sort, altså at:

$$\{S_1, \ldots, S_i\} \cap C = \emptyset$$

Predikatene *har tilordnet en aritet som er et naturlig tall større enn eller lik en, og de har argumenter med en tilordnet sort, skrevet*

$$R(x_1{:}S_1, \ldots, x_n{:}S_n)$$

der $x{:}S$ betyr at x er en variabel av sort S. Det kreves at sortkravene til predikatene overholdes.

Predikater med aritet lik én kan brukes til å uttrykke at en term tilhører en sort. Disse entallspredikatene kan skrives på formen $x{:}S$. Det tillates at medlemmer legges til eller fjernes fra en sort etterhvert som databasen oppdateres.

Fakta er grunne instanser av predikater, det vil si at argumentene til predikatsymbolene er konstanter. Vi krever at sort-kravene til predikatet er oppfylt. Mengden av tabeller i en relasjonsdatabase kan representeres som en mengde fakta.

I eksempelet nedenfor bruker jeg sortene på tilsvarende måte som datatyper brukes i relasjonsdatabaser. Passende sorter kan for eksempel være *Text* for representasjon av tekststrenger og *Int* for representasjon av heltall. Eksempeldatabasen blir da representert som i eksempel 5:

Eksempel 5 (Representasjon av eksempeldatabasen i $\mathcal{L}_F$)
Eksempeldatabasen kan representeres i $\mathcal{L}_F$ ved:

- *En mengde sorter S:*
$$S = \{Text, Int\}$$
- *En mengde konstanter C:*
$$C = \{ole, kari, per, A, B, C, 200, 300, 400, 100\}$$
- *Tilordning av konstantene til sorter: 2.1:*
$$\begin{aligned} ole&:Text & 100&:Int \\ kari&:Text & 200&:Int \\ per&:Text & 300&:Int \\ A&:Text & 400&:Int \\ B&:Text \\ C&:Text \end{aligned}$$
- *Predikater som representerer tabellene:*
$$\begin{aligned} Studentkurs(x_1{:}Text, x_2{:}Int) \\ Studenter(x_1{:}Text, x_2{:}Text) \\ Kurs(x_2{:}Int, x_1{:}Text) \end{aligned}$$
- *Lukkede formler som representerer de ulike tabellene:*
$$\begin{aligned} Studenter(ole{:}Text, A{:}Text) && Studentkurs(A{:}Text, 300{:}Int) \\ Studenter(kari{:}Text, B{:}Text) && Studentkurs(B{:}Text, 400{:}Int) \\ Studenter(per{:}Text, C{:}Text) && Studentkurs(C{:}Text, 100{:}Int) \\ Kurs(100{:}Int, Logikk{:}Text) \\ Kurs(200{:}Int, Matte{:}Text) \\ Kurs(300{:}Int, Fysikk{:}Text) \\ Kurs(400{:}Int, Kinesisk{:}Text) \end{aligned}$$

3.2 $\mathcal{L}_C$: Skrankespråket

Skrankespråket fra [7] kan brukes til å uttrykke skrankene i en relasjonsdatabase. Disse formlene skal representere *den karakteristiske formelen* i metaspråket, og bestemmer hvilke tilstander som er lovlige for databasen.

Definisjon 5 (Skrankespråket – $\mathcal{L}_C$)
La R være en relasjon, og $S_1, \ldots, S_n$ være sorter, da er

$$R(x_1 : S_1, \ldots, x_n : S_n)$$

en formel i $\mathcal{L}_C$. I tillegg er følgende formler dersom φ og ψ er formler:

$$\begin{array}{cc} \varphi \wedge \psi & \varphi \vee \psi \\ \neg\varphi & \varphi \supset \psi \\ (\forall x{:}S)(\varphi) & (\exists x{:}S)(\varphi) \end{array}$$

En relasjonsdatabase kan beskrives i $\mathcal{L}_F$ og $\mathcal{L}_C$, med fakta representert i $\mathcal{L}_F$ og relasjonsdatabasens skranker representert i $\mathcal{L}_C$.

Eksempel 6 (Skranker for eksempeldatabasen uttrykt i $\mathcal{L}_C$)
Det første kravet til databasen i avsnitt 2.1 var at alle studenter og kurs som var nevnt i Studentkurs-*tabellen også skulle forekomme i* Student- *og* Kurs-*tabellene. Dette kan uttrykkes i $\mathcal{L}_C$ med formlene*

$$(\forall x_1{:}Text)(\forall x_2{:}Text)(Studentkurs(x_1{:}Text, x_2{:}Int) \supset (\exists x_3{:}Text)Student(x_3{:}Text, x_1{:}Text)) \quad (3.1)$$

$$(\forall x_1{:}Text)(\forall x_2{:}Text)(Studentkurs(x_1{:}Text, x_2{:}Int) \supset \exists x_3{:}Text(Kurs(x_2{:}Int, x_3{:}Text))) \quad (3.2)$$

Kravet om at alle navn skal være unike blir oppfylt i det vi oversetter databasen til et utsagnslogisk språk i avsnitt 3.3. Det er derfor ikke nødvendig å spesifisere dette (det er heller ikke mulig å gjøre dette i førsteordens logikk).

Det er verdt å merke seg at konstanter alltid kan uttrykkes uten å referere til konstanter, ved å innføre en ad-hoc tabell og bruke fremmednøkler, som i eksempel 7.

Eksempel 7 (Uttrykke en skranke med en ad-hoc tabell)
I eksempeldatabasen ønsker vi at ingen student skal delta på kurs 2 og 4 samtidig. Dette kan vi oppnå ved å legge til en tabell med uønskede kurskombinasjoner:

Ulovlig kurskombinasjon	
200	400
400	200

Tabell 3.1: Uønskede kurskombinasjoner

Denne tabellen representeres som formlene:

$$Ikkekurs(200{:}Int, 400{:}Int)$$
$$Ikkekurs(400{:}Int, 200{:}Int)$$

i $\mathcal{L}_F$. Nå kan den ønskede skranken uttrykkes ved formelen

$$\forall x_1{:}Text((Studentkurs(x_1{:}Text, x_2{:}Int) \wedge Ikkekurs(x_2{:}Int, x_3{:}Int)) \supset \neg Studentkrus(x_1{:}Text, x_3{:}Int))$$

Ved å om nødvendig innføre slike ad-hoc tabeller er det mulig å uttrykke alle mulige skranker uten å referere direkte til konstantene, og det er derfor greit å definere språket $\mathcal{L}_C$ uten konstantsymboler.

3.3 $\mathcal{L}_{PL}$: Utsagnslogisk ekstensjon av $\mathcal{L}_F$ og $\mathcal{L}_C$

Den utsagnslogiske ekstensjonen av kvantifiserte $\mathcal{L}_C$-formler er gitt ved følgende omskrivingsregler fra [7]:

Definisjon 6 ($\mathcal{L}_{PL}$)
$\mathcal{L}_{PL}$ er definert som tillukningen av $\mathcal{L}_F$ og $\mathcal{L}_C$, der alle kvantifiserte formler er ekspanderte inn i utsagnslogiske formler i følge dette mønsteret:

$$(\forall x{:}S) \rightsquigarrow \bigwedge\{\varphi[a/x] : a \in S\}$$
$$(\exists x{:}S) \rightsquigarrow \bigvee\{\varphi[a/x] : a \in S\}$$

Eksempel 8 (Ekstensjon av formelen 3.1)
Den utsagnslogiske ekstensjonen av formelen 3.1 blir:

$$
\begin{aligned}
&Studentkurs(A, 100) \supset (Student(ole, A) \vee Student(per, A) \vee Student(kari, A)) \wedge \\
&Studentkurs(A, 200) \supset (Student(ole, A) \vee \ldots) \wedge \\
&\vdots \\
&Studentkurs(C, 100) \supset (Student(kari) \vee \ldots) \wedge \\
&\vdots \\
&Studentkurs(C, 400) \supset (Student(kari) \vee \ldots)
\end{aligned}
$$

Tilsvarende kan man utvide de andre formlene fra eksempelet, ved å substituere inn alle mulige kombinasjoner av konstantsymbolene for variablene, og ta konjunksjonen av alle disse kombinasjonene for universelt kvantifiserte formler, og disjunksjonen av disse for eksistensielt kvantifiserte formler.

Resultatet av å utføre denne tillukningen er at vi står igjen med et utsagnslogisk språk der utsagnsvariablene er de ulike atomære formlene.

3.4 En deduktiv database i Æ

På toppen av den utsagnslogiske ekstensjonen av $\mathcal{L}_F$ og $\mathcal{L}_C$ ligger det *refleksive* laget, som er en metateori formalisert i språket Æ [17].

Definisjon 7 (Syntaks for Æ)
Objektspråket består av en mengde utsagnsvariabler, konstantene $\top$ *og* $\bot$*, konnektivene* $\neg, \wedge, \vee, \supset$ *og* $\equiv$*.*

Dersom Γ er en endelig mengde formler skriver vi konjunksjonen av en vilkårlig mengde som $\bigwedge \Gamma$ og disjunksjonen som $\bigvee \Gamma$. De modale operatorene er $\Box$, B_k (oppfatning) og C_k (co-oppfatning) der k er en indeks som spenner over en endelig indeks-mengde I, partielt ordnet av en relasjon $\preceq$.

Definisjon 8 (Modale atomer)
Et modalt atom *er en formel på formen* $B_k\varphi$*,* $C_k\varphi$ *eller* $\Box\varphi$ *og et* modalt literal *er enten et modalt atom eller dets negasjon.*

Definisjon 9 (De modale operatorene $\Diamond\varphi$, $b_k\varphi$ og $c_k\varphi$)
$\Diamond\varphi$ er definert som $\neg\Box\neg\varphi$ (mulighet), $b_k\varphi$ er definert som $\neg B\neg\varphi$ (φ er kompatibel med oppfatning ved k) og analogt for c_k.

Den *utsagnslogiske substitusjonsoperatoren* $\varphi[\psi_1/\psi_2]$ gir oss φ der alle forekomster av ψ_1 er erstattet med ψ_2. En *tautologi* er en substitusjonsinstans av en formel som er gyldig i klassisk utsagnslogikk; dersom φ er en tautologi i utsagnslogikk skriver vi $\vdash_{PL} \varphi$.

Logikken Æ er definert som den minste mengden som inneholder alle (utsagnslogiske) tautologier, og som inneholder alle instanser av de følgende aksiomskjemaer for alle $k \in I$:

$$
\begin{array}{rlrl}
Def\Box: & \Box\varphi \equiv (B_k\varphi \wedge C_k\varphi) & T: & \Box\varphi \supset \varphi \\
K_B: & B_k(\varphi \supset \psi) \supset (B_k\varphi \supset B_k\psi) & K_C: & C_k(\varphi \supset \psi) \supset (C_k\varphi \supset C_k\psi) \\
B_\Box: & B_k\varphi \supset \Box B_k\varphi & C_\Box: & C_k\varphi \supset \Box C_k\varphi \\
\overline{B_\Box}: & \neg B_k\varphi \supset \neg\Box B_k\varphi & \overline{C_\Box}: & \neg C_k\varphi \supset \neg\Box C_k\varphi \\
P_B: & B_i\varphi \supset B_k\varphi \text{ for alle } i \prec k & P_C: & C_i\varphi \supset C_k\varphi \text{ for alle } i \prec k
\end{array}
$$

Definisjon 10 (Språket for det refleksive laget)
Det refleksive *laget er en metateori formalisert i språket Æ, der vi legger til de modale formlene $B\varphi$, $C\varphi$, og $\Box\varphi$, i tillegg til dualene $b\varphi$, $c\varphi$ og $\Diamond\varphi$, til $\mathcal{L}_{PL}$.*

Nå kan en database i Æ defineres ved følgende:

Definisjon 11 (Æ-database)
En Æ-database består av følgende:

- *Grunnleggende fakta, representert som en mengde atomære formler Φ i $\mathcal{L}_F$*
- *Skranker, representert som en mengde formler Γ i $\mathcal{L}_C$.*
- *En mengde Æ-formler Δ.*

Formlene i Γ representeres som en mengde formler Ψ i $\mathcal{L}_{PL}$, ved omskrivingen gitt av definisjon 6.

$\bigwedge\Gamma$ brukes som den karakteristiske formelen, og denne bestemmer et logisk rom *[17].*
En Æ-database er definert som unionen av Φ, Δ og ρ.

Æ - Modeller

Definisjon 12 (Æ modell)
En Æ-modell består av et kvadruppel (U, U^+, U^-, V)*, der* U *er en ikke-tom mengde av punkter,* U^+ *og* U^- *er funksjoner som tilordner en delmengde av* U *til hver indeks i* I*.*

$U^+(k)$ og $U^-(k)$ skrives som henholdsvis U_k^+ og U_k^-. V er en valuasjonsfunksjon som tilegner en delmengde av U til hver utsagnsvariabel i språket. For hver $k \in I$ kreves det at

$$U_k^+ \cup U_k^- = U$$

$$U_k^+ \subseteq U_k^+ \text{ og } U_k^- \subseteq U_k^- \text{ for hver } i \prec k \tag{3.3}$$

Definisjon 13 (todelt modell)
En modell er todelt *dersom det for hver* $k \in I$*:*

$$U_k^+ \cap U_k^- = \emptyset \tag{3.4}$$

En oppfyllbarhetsrelasjon kan defineres for hvert punkt x:

- $M \models_x p$ hviss $x \in V(p)$ for en utsagnsvariabel p.
- $M \models_x$ hviss $M \models_y$ for hver $y \in U$
- $M \models_x B_k\varphi$ hviss $M \models_y \varphi$ for hver $y \in U_k^+$
- $M \models_x C_k\varphi$ hviss $M \models_y \varphi$ for hver $y \in U_k^-$

Relasjonen er som normalt for boolske konnektiver. ($M \models \varphi \wedge \psi$ hviss $M \models \varphi$ og $M \models \psi$ og så videre). Vi skriver $||\varphi||$ for *sannhetsmengden* av φ i M (der M er en modell): $||\varphi|| = \{X \in U | M \models_x \varphi\}$. Intuitivt betyr dette mengden av alle punkter i universet til M der φ er sann.

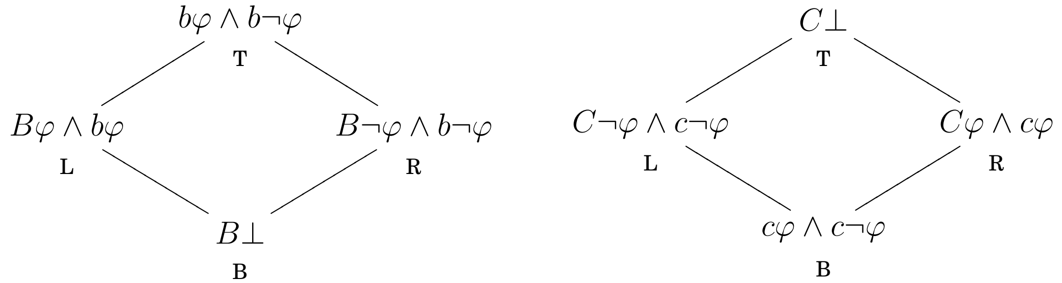
Definisjon 14 (Oppfyllbarhet og sannhet)
En formel er oppfylt *i en modell dersom den er sann i et av modellens punkter. Dersom* $M \models_x \varphi$ *for alle* $x \in U$ *skriver vi* $M \models \varphi$ *og sier at* φ er sann i M*. Dersom* φ *er sann i alle modeller skiver vi* $\models \varphi$*.*

Siden vi får samme sannhetsverdi for alle punktene i en modell for en fullstendig modalisert formel, gir det mening å bruke notasjonen $M \models \varphi$ dersom en fullstendig modalisert formel φ er tilfredsstilt i M. Det kan vises at dersom det i en modell M for hver $k \in I$ finnes en formel φ slik at M tilfredsstiller formelen $O_k\varphi_k$, så er M todelt. Det bør nevnes at todeling ikke er en føring vi legger på modellene, men det er enkelt å tvinge frem todeling ved å legge til et syntaktisk krav: La M være en modell som, for hver $k \in I$, tilfredsstiller en formel $O_k\varphi_k$. Da er M todelt [17].

Æ - tolkning. Jeg introduserer et forslag til tolkning av Æ, basert på teksten i [17]. En Æ-modell er ment å skulle representere et "doksastisk" subjekt. Universet U til en modell representerer alle tilstander som er forestillbare for et "doksastisk" subjekt. U referes noen ganger til som *"Space of conceivability"*, altså "forestillingsrommet". Subjektet har en *oppfatningstilstand* som ved hver grad av tiltro[1] er modellert ved U_k^+, og en *co-oppfatning* tilstand modellert ved U_k^-.

Det er naturlig å se på oppfatnings-modaliteten B som uttrykk for at punkter utelukkes som ikke-plausible: $B\varphi$ uttrykker at alle punkter der $\neg\varphi$ er sann er utelukket. Tilsvarende uttrykker $C\varphi$ at *ingen* punkter hvor $\neg\varphi$ er sann kan utelukkes; gitt en todelt modell uttrykker dette at alle $\neg\varphi$-punkter er plausible. Det er verdt å observere at dersom:

$$M \vDash C\neg\varphi \text{ så er} ||\varphi|| \subseteq U^+$$



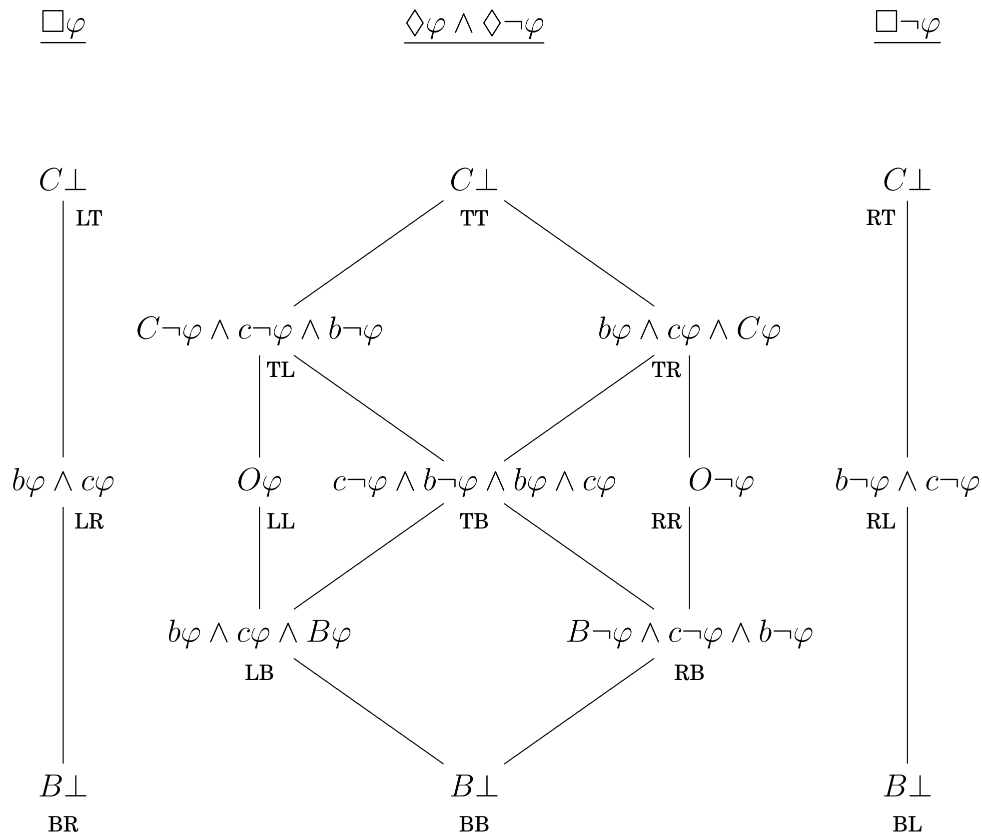
Figur 3.1: Doksastiske posisjoner for B og C

Tolkning av $C\varphi$. Operatorene B og C gir opphav til *doksastiske posisjoner*, som er skissert i figur 3.1, fra [17, side 16].

Ved å kombinere de fire modale operatorene (B, C, b, og c) får vi muligheten til å skille mellom ulike grader av "doxastic attitude toward a proposition". For eksempel kan et uttrykk for oppfatning $B\varphi$ spesifiseres videre ved å kombinere det med $b\varphi$, for å spesifisere at b er "consistently believed", eller med $C\neg\varphi$ for å spesifisere at "precisely φ is believed" [17]. Resultatet er mulighet for en finkornet gradering av utsagn.

Vi har fire konsistente kombinasjoner av B og b, disse er $B\bot$, $B\varphi \wedge b\varphi$, $B\neg\varphi \wedge b\neg\varphi$ og $b\varphi \wedge b\neg\varphi$. Tilsvarende kombinasjoner har vi for

[1] confidence



Figur 3.2: De 15 doksastiske posisjoner for kombinasjoner av B og C

C og c. Se figur 3.1, fra [17]. I artikkelen finnes det en presis oversikt og tolkning av disse. Dersom vi kombinerer én av hver av disse B- og C-kombinasjonene ender vi opp med vi 16 ulike kombinasjoner, hvorav 15 er konsistente. Det er disse 15 konsistente kombinasjonene som senere vil være interessante å se på i forbindelse med et spørrespråk for en deduktiv database, siden disse tilfører økt uttrykkskraft i forhold til for eksempel autoepistemisk logikk eller Levesques $\mathcal{OL}$. Disse 15 konsistente posisjonene er skissert i figur 3.2 (denne figuren er også hentet fra [17, side 18]).

Muligheter i en Æ-database Den store uttrykkskraften i Æ skal først og fremst utnyttes i tre sammenhenger; det skal brukes til å kode *defaulter* inn i databasen. Det vil si at vi lagrer informasjon om den "normale" tilstanden til data i databasen, og bruker denne til å trekke slutninger dersom vi mangler eksplisitt informasjon. For eksempel kan det være slik at en høyeregrades-student vanligvis ikke er laveregradsstudent samtidig.

For å kunne gi svar på spørringer skal Æ brukes til å uttrykke en rekke *doksastiske posisjoner*, som karakteriserer den modale tilstanden til en vilkårlig formel med hensyn på databasen. De doksastiske posisjonene er formet som en graf, og ved å bruke posisjoner i denne som svar på spørringer vil det være mulig å få mer presise svar enn det som er mulig i andre databaser.

En tredje og spesielt interessant mulighet som Æ gir er muligheten til å kunne angi negasjon ved feiling eksplisitt for bestemte tabeller/relasjoner i databasen. Ved å utnytte denne muligheten kan man velge å tolke databasen under antagelsen om åpen verden, slik at svaret på spørringen

$$\neg Studentkurs(A, 100) \tag{3.5}$$

ikke ga svaret `true`, siden det ikke finnes informasjon i databasen om hvorvidt `A` tar kurs `1`. Så kan man anta at på et tidspunkt ut i semesteret er det ikke lenger mulig å melde seg på flere kurs. Nå vil det være rimelig at svaret på 3.5 er `false`, og dette kan løses ved å legge til informasjon som sier at informasjonen i relasjonen $Studentkurs$ skal betraktes som fullstendig, ved å legge til denne som en defaultregel.

Defaulter skrives på mønsteret

$$\frac{\alpha : \beta}{\gamma}$$

der α, β, og γ er formler i $\mathcal{L}_{PL}$, og tolkes som at dersom α er sann og β er konsistent, så kan vi slutte γ, en introduksjon til defaulter finnes i [4].

Eksempel 9
Dersom vi ønsker å uttrykke negasjon ved feiling for Studentkurs-*tabellen kan vi gjøre dette med følgende defaultregel:*

$$\frac{: \neg Studentkurs(x, y)}{\neg Studentkurs(x, y)}$$

der vi setter inn vilkårlige konstanter for de frie variablene.

Vi kan derfor velge å tolke databasen *uten* negasjon ved feiling, og vi kan velge å sette eksplisitt negasjon ved feiling for de enkelte tabellene i databasen. Negasjon ved feiling kan legges til for hver enkelt tabell i databasen, slik det er gjort i eksempel 9. Vi får dermed mye finere kontroll over dataene i databasen.

3.5 TELL og ASK

Levesque introduserte i artikkelen [8] et grensesnitt mot en kunnskapsbase som består av funksjonelle operatorer. Sentralt er funksjonene TELL som legger til ny kunnskap i en eksisterende kunnskapsbase og ASK som er en operasjon som kan stille spørsmål til basen.

Motivasjon for å legge til TELL og ASK er todelt; på den ene siden har vi tanken om et funksjonelt grensesnitt, noe som gir oss en lagdeling av databasen. På den andre siden ønsker vi å tydeliggjøre at språket som vi bruker til å stille spørsmål og legge data til databasen ikke er det samme som vi bruker til å representere data internt i databasen. Selv om representasjonsspråket er et førsteordens språk er ikke dette tilstrekkelig til å snakke om data som ligger til databasen.

Ved å innføre TELL og ASK som grensesnitt mot en database oppnår vi også at databasen blir en "abstrakt datatype" som interagerer med brukeren gjennom et fåtall operasjoner. I det mest grunnleggende tilfellet har vi to operasjoner: en TELL-funksjon som legger til data (en påstand) i databasen og en ASK-funksjon som stiller spørsmål til databasen (spørring). Det er også mulig å forestille seg andre operasjoner, for eksempel FORGET og ASSUME, ikke alle er like interessante å studere her – men med tanke på databaseanvendelser er det selvsagt interessant med som hva, hvem, hvor etc.

TELL og ASK er definert på det Levesque kaller "level of knowledge", uten å gjøre noen antakelser om hvordan kunnskapen i kunnskapsbasen er representert. Det Levesque mener med "level of knowledge" er det grensesnittet brukeren møter i kontakt med kunnskapsbasen. Siden det er et begrenset språk som brukes til å interagere med kunnskapsbasen, er det heller ikke mulig for denne å ha kunnskap utover det som kan uttrykkes i dette språket.

ASK er en funksjon som tar eksisterende kunnskap og en spørring og returnerer et svar. TELL er en funksjon som tar eksisterende kunnskap og en påstand og returnerer ny kunnskap:

$$\text{ASK} : Knowledge \times Query \rightarrow Answer$$
$$\text{TELL} : Knowledge \times Assertion \rightarrow Knowledge$$

I Levesques tekst er mengden mulige svar begrenset til "ja", "nei" og "vet ikke", og han kan dermed la språket for å uttrykke spørringer og påstander være det samme. Det kan være fire ulike resultater av en spørring. Dersom spørringen uttrykkes som en setning i et spørrespråk og man spør om den er sann i databasen kan resultatet være at setningen er sann, eller at setningen er usann. De to andre mulighetene er at databasen er inkonsistent, og dermed svarer at *alle* spørringer er sanne, eller at databasen ikke er i stand til å gi et svar, det vil si at setningen kan være både sann og usann.

Dersom vi ønsker å se på den underliggende databasen som en abstrakt datatype trenger vi en ekstra funksjon som returnerer en tom database; $INITIAL[] = e_0$. Man kan da tenke seg "livsløpet" til en database som en sekvens av tilstander $e_0, e_1, e_2, \ldots$ hvor e_0 er resultatet av en $INITIAL$-operasjon og for alle $i > 0$ er det setninger α_i slik at $e_i = \text{TELL}[\alpha_i, e_{i-1}]$.

ASK i Æ Ask er en funksjonell operator som ut fra en spørring (formulert som en formel i Æ) og en eksisterende database skal returnere et svar som forteller oss noe om hvorvidt formelen er sann i databasen. Svaret vi ønsker oss er en posisjon i figuren 3.2:

$$\text{ASK} : \alpha \times DB \rightarrow LP$$

Der DB er databasen, α er en setning i Æ, og LP er en posisjon i figuren 3.2. Det er med andre ord en funksjon som tar en formel og en database,

og returnerer en formel, angitt ved en posisjon i en graf, som svar[2]. α kan være en formel i $\mathcal{L}_{PL}$, eller det kan være en formel i Æ.

Svaret vi får vil avhenge av hva slags form α har, spesielt vil vi få et ulikt svar om vi lar α være en modal formel eller om α er en formel i $\mathcal{L}_{PL}$. Dersom vi lar spørringen være en formel i $\mathcal{L}_{PL}$ vil vi få et modalt svar, angitt av en doksastisk-posisjon. Dersom vi i stedet stiller spørsmål om hvorvidt en modal formel er sann, vil svaret være et ja/nei-svar.

Tolkning av svar på ASK. Grafen i figur 3.2 har 15 ulike posisjoner, og ASK er en funksjon som ut fra en formel peker ut en av disse. Artikkelen [17] skisserer hva en del av disse posisjonene betyr i et svar, og jeg skal se på eksempler på noen spørringer og hva svaret på disse spørringene bør være.

Algoritme for ASK. Algoritmen for å avgjøre hvilken posisjon i grafen kan være en konsistensjekk der man tester formelen det spørres om mot hver enkel posisjon i grafen. En mulighet for å lage en slik algoritme er å bruke ekvivalensene som inngår i algoritmen for reduksjonsteoremet i [17].

TELL i Æ. Den grunnleggende TELL-operatoren i en Æ-database kan være en funksjon som legger en formel til formelmengden som databasen består av:

$$\text{TELL} : DB \times \alpha \rightarrow DB$$

der α er en lukket formel og DB er en database. TELL er altså en funksjon som tar en lukket formel og en database, og returnerer en database, eller med andre ord: oppdaterer databasen.

Eksempel 10 (Spørring med en modal formel)
Vi ønsker å stille spørsmålet "er ole nødvendigvis høyeregradsstudent". Spørringen kan defineres ved formelen:

$$\Box Hgrad(ole)$$

Her spør vi med en modal formel, og vi får derfor et `ja/nei`*-svar. Svaret på denne spørringen er* `no`*.*

[2] Dette er i motsetning til Levesque som har definert ASK som en funksjon som returnerer $\{yes, no\}$.

Dersom vi i stedet stiller spørsmålet "er ole høyeregradsstudent" vil en Æ-database kunne gi mer informasjon enn bare `ja` *og* `nei`*, siden vi ikke har noe informasjon om hvorvidt Ole er høyeregrads-student eller ikke, men det er heller ikke inkonsistent med hva vi vet. Vi bør derfor få svaret* TB *på denne spørringen.*

Eksempel 11 (En spørring i en Æ-database.)
Vi ønsker å spørre databasen spørsmålet "Er student A på kurs 300". En tradisjonell database ville svart "nei" på dette spørsmålet, siden kombinasjonen $(A, 300)$ *ikke finnes i* studentkurs-*tabellen. Ut fra closed world-assumption vil den da konkludere med at* `A` ikke *er på kurs 300.*

I vår database har vi et mer uttrykkskraftig spørrespråk, og vi ønsker å utnytte dette til å gi et mer utfyllende svar; nemlig noe i retning av "vi vet ikke dette med sikkerhet nå, men det er en mulighet" (dersom vi legger til $(A, 3)$ *vil ikke dette gi oss en inkonsistent database). Spørringen vår kan formuleres som*

$$\text{ASK}(Studentkurs(A, 3), DB)$$

Og det ønskede svaret er det som i figur 3.2 tilsvarer posisjonen TL, *nemlig*

$$C\neg\varphi \wedge c \wedge b\neg\varphi$$

3.6 Oppsummering

I dette kapittelet har jeg sett på hvordan et språk for en deduktiv database i Æ kan se ut, og jeg har sett på noen eksempler på hva man kan uttrykke i dette språket. ASK-funksjonen tilfører økt uttrykkraft til spørrespråket i forhold til tradisjonelle databaser, ved at man får mulighet til å gradere og nyansere svaret på en spørring. TELL-funksjonen tilfører også nye muligheter, spesielt det at man får mulighet til å ha en *selektiv* antagelse om lukket verden, og at man får muligheten til å legge til generelle defaulter i databasen.

Kapittel 4

Implementasjon i Maude

4.1 L_I - Sekventkalkyle

I artikkelen av Waaler [16] er det introdusert en sekventkalkyle for en variant av Æ, kalt L_i. L_i er en multiagent-variasjon av Æ, men den mangler tillitsnivåene.

I motsetning til Æ slik det er presentert ovenfor, representerer indeksmengden her en mengde agenter. Denne logikken inneholder derfor ikke aksiomskjemaene P_B og P_C, men inneholder et aksiomskjema kalt *conv*-aksiomet. Dette har formen

$$\Diamond : \Diamond_k\varphi \text{ forutsatt } \varphi \nvdash \bot, \text{ der } \varphi \text{ er fri for modalitet } k$$

Conv-aksiomet sikrer at $\Box_k\varphi$ er bevisbar bare hvis φ er bevisbar. Dessuten gjør det at B_k og C_k er komplementære modaliteter [16].

Språket og sekventkalkylen. Som for Æ består språket av en mengde utsagnsvariabler, konstantene $\top$ og $\bot$, de boolske konnektivene $\neg$, $\vee$, $\wedge$, $\supset$ og $\equiv$. De modale operatorene B_k og C_k er definert for hver $k \in I$, der I er en ikketom indeksmengde. I motsetning til for Æ representerer indeksmengden her en mengde agenter. B_k er en *oppfatnings* operator og C_k er en (komplementær) *co-oppfatnings* operator for agenten k. Symbolet O_k er definert ved $O_k\varphi = B_k\varphi \wedge C_k\neg\varphi$. Man kan si at $O_k\varphi$ betyr at φ er en eksakt representasjon av agenten ks oppfatninger. Videre forkortelser: $b_k = \neg B_k\neg$, $c_k = \neg C_k\neg$ og $\Box_k\varphi = B_k\varphi \wedge C_k\varphi$. Dualen $\Diamond_k\varphi$ kan dermed defineres som $b_k\varphi \vee c_k\varphi$.

En formel φ er *fullstendig modalisert* dersom alle forekomster av utsagnsvariabler er innenfor skopet til en modal operator. Den er *fullstendig boolsk* dersom det ikke forekommer noen modale operatorer.

En formel φ er en *første-ordens formel* dersom for alle $k \in I$ og for hver subformel $B_k\psi$ eller $C_k\psi$ i φ så er ψ fri for modalitet k. Den *modale dybden* $m(\varphi)$ uttrykker nøstingen av alternerende modaliteter i φ.

Definisjon 15 (Modal dybde)

- *Den modale dybden til en fullstendig Boolsk formel φ er lik 0.*
- *Dersom φ er på formen $B_k\psi$ eller $C_k\psi$ er den modale dybden definert som:*
  - *La Ψ være mengden av modale atomer som forekommer som delformler til ψ.*
  - *Den modale dybden $m(\varphi)$ er det største tallet i $\{m(\chi)+1 \mid \chi \in \Psi$ og χ er ikke k-modalisert $\} \cup \{m(\chi) \mid \chi \in \Psi$ og χ er k-modalisert$\}$*
- *Ellers er den modale dybden til φ den maksimale $m(\varphi)$ for en delformel ψ av φ.*

Vi sier at en formel med modal dybde lik 1 er *uten nøstede modaliteter*, eller på *normalform*.

L_I er definert som den minste mengden som inneholder alle tautologier, og som er lukket under alle instanser av slutningsreglene

$$\frac{\Box_k\varphi}{\varphi}\ \text{RN} \qquad\qquad \frac{\varphi \quad \varphi \supset \psi}{\psi}\ \text{MP}$$

og aksiomskjemaene:

K_B:	$B_k(\varphi \supset \psi) \supset (B_k\varphi \supset B_k\psi)$
K_C:	$C_k(\varphi \supset \psi) \supset (C_k\varphi \supset C_k\psi)$
$B_\Box$:	$B_k\varphi \supset \Box_k B_k\varphi$
$C_\Box$:	$C_k\varphi \supset \Box_k C_k\varphi$
$\overline{B_\Box}$:	$\neg B_k\varphi \supset \Box_k\neg B_k\varphi$
$\overline{C_\Box}$:	$\neg C_k\varphi \supset \Box_k\neg C_k\varphi$
$\Diamond$:	$\Diamond_k\varphi$ forutsatt $\varphi \nvdash \bot$, der φ fri for modalitet k

Disse er de samme aksiomene som for Æ, men persistensaksiomene er fjernet siden det ikke er tillitsnivåer i språket. Aksiomskjemaet $Def\Box$ er erstattet med aksiomet $\Diamond$, kalt conv-aksiomet.

Definisjon 16 (Sekvent)

En sekvent *er et utsagn på formen $\Gamma \Rightarrow \Delta$. Formelmengden Γ kalles* antecedenten *og formelmengden Δ kalles* succedenten*. Både Γ og Δ er endelige multimengder av formler.*

$$\frac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ LT \qquad \frac{\Gamma \Rightarrow \varphi, \Delta}{\Gamma \Rightarrow \Delta}\ RT$$

$$\frac{\Gamma, \varphi, \varphi \Rightarrow \Delta}{\Gamma, \varphi \Rightarrow \Delta}\ LC \qquad \frac{\Gamma \Rightarrow \varphi, \varphi, \Delta}{\Gamma \Rightarrow \varphi, \Delta}\ RC$$

Figur 4.1: Strukturelle regler

$$\frac{}{\varphi \Rightarrow \varphi}\ ID \qquad \frac{\Gamma_1 \Rightarrow \varphi, \Delta_1 \quad \Gamma_2, \varphi \Rightarrow \Delta_2}{\Gamma_1, \Gamma_2 \Rightarrow \Delta_1, \Delta_2}\ CUT$$

Figur 4.2: Identitetsregler

Jeg skriver Γ, φ for $\Gamma \cup \{\varphi\}$, og jeg følger konvensjonen fra Waalers artikkel [16], og skriver Γ^{B_k} for mengden $\{B_k\varphi | \varphi \in \Gamma\}$ og tilsvarende for Γ^{C_k}. Γ^k betyr at Γ kun inneholder modale atomer av modalitet k.

Definisjon 17 (Bevis)
Et bevis *i sekventkalkylen er et tre som er bygd opp ved å bruke reglene for sekventkalkylen, og hvor alle løvnodene er aksiomer.*

Reglene er delt i fire grupper, de strukturelle reglene, identitetsreglene, de logiske reglene og reglene for modalitetene. De strukturelle reglene for sekventkalkylen er gjengitt i figur 4.1, identitetsreglene er gjengitt i figur 4.2, de logiske reglene er i figur 4.3 og reglene for modalitetene finnes i figur 4.4.

4.2 Algoritme for sekventkalkylen

I dette avsnittet skal jeg skissere en algoritme å søke etter bevis med sekventkalkylen. Jeg skal se på hvilke problemer og utfordringer som oppstår ved spesifikasjonen av en slik algoritme, og vise hvordan noen av disse problemene kan løses. Spesielt skal jeg se på backtracking og terminering. I avsnitt 4.4 skal jeg også se på hvordan man kan kontrollere rekkefølgen for anvendelse av reglene, og jeg har løst en del av termineringsproblemene i implementasjonen.

Et bevis er et tre som bygges opp ved bruk av reglene i figurene 4.1, 4.2, 4.3 og 4.4, som er slik at alle løvnodene er aksiomer. Noen av reglene splitter treet, og reglene kan deles inn i tre kategorier etter hvordan de splitter opp bevistreet. Den ene kategorien er de reglene som har én

$$\frac{\Gamma \Rightarrow \varphi, \Delta}{\Gamma, \neg\varphi \Rightarrow \Delta} L\neg \qquad \frac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \neg\varphi\Delta} R\neg$$

$$\frac{\Gamma, \varphi_1, \varphi_2 \Rightarrow \Delta}{\Gamma\varphi_1 \wedge \varphi_2 \Rightarrow \Delta} L\wedge \qquad \frac{\Gamma_1 \Rightarrow \varphi\Delta_1 \quad \Gamma_2 \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi\Delta} R\wedge$$

$$\frac{\Gamma, \varphi \Rightarrow \Delta_1 \quad \Gamma, \psi \Rightarrow \Delta_2}{\Gamma, \varphi \vee \psi \Rightarrow \Delta} L\vee \qquad \frac{\Gamma \Rightarrow \varphi, \psi, \Delta}{\Gamma \Rightarrow \varphi \vee \psi, \Delta} R\vee$$

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma, \varphi \supset \psi \Rightarrow \Delta} L \supset \qquad \frac{\Gamma, \varphi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \supset \psi, \Delta} R \supset$$

$$\frac{\Gamma, \varphi \Rightarrow \psi, \Delta \quad \Gamma, \psi \Rightarrow \varphi, \Delta}{\Gamma, \varphi \equiv \psi \Rightarrow \Delta} L \equiv$$

Figur 4.3: Regler for de logiske konnektivene

$$\frac{\Theta^k, \Gamma \Rightarrow \Delta^k, \varphi}{\Theta^k, \Gamma^{B_k} \Rightarrow \Delta^k, B_k\varphi} LRB_k \; ^a \qquad \frac{\Theta^k, \Gamma \Rightarrow \Delta^k, \varphi}{\Theta^k, \Gamma^{C_k} \Rightarrow \Delta^k, C_k\varphi} LRC_k \; ^b$$

$$\frac{}{B_k\varphi, C_k\psi \Rightarrow} CONV \; ^c$$

[a] forutsatt at Δ^k, φ er ikke-tom
[b] forutsatt at Δ^k, φ er ikke-tom
[c] dersom $\{\neg\varphi, \neg\psi\}$ er konsistent

Figur 4.4: Regler for modalitetene

premiss og én konklusjon, de splitter altså ikke opp treet. Dette er reglene i figur 4.1, samt reglene med bare én premiss i figur 4.3. Videre er det reglene med to premisser og én konklusjon, disse er resten av reglene i 4.3. Her må begge premissene være oppfylt dersom konklusjonen skal være det. Til slutt det de reglene i figur 4.4 som har én premiss og én konklusjon, men der det er mulig å velge hvilken formelmengde man inkluderer i mengden Γ i premissene. For disse reglene holder det at man kan finne et bevis for *en* av de mulige premissene for at konklusjonen skal være oppfylt. Disse reglene gir derfor opphav til *backtracking*. Dette skjer når man man gjør feil valg her, og må gå tilbake og velge en av de andre mulige premissene for denne regelen.

Underveis har man både valg mellom hvilke regler som skal brukes, og for reglene for modalitetene har man også mulighet til å velge hvordan reglene kan brukes. En algoritme som benytter en vilkårlig lovlig regel blir dermed ikkedeterministisk.

Det siste problemet er at vi en ikketerminerende kalkyle, siden de strukturelle reglene gir opphav til en uendelig løkke der tynning og kontraksjon brukes om hverandre og legger til og fjerner den samme formelen gjentatte ganger. Det er også mulig å få til ikke-terminering ved å bruke en logisk modale reglene sammen med kontraksjonsregelen. Dersom man bruker kontraksjon først, og så bruker en annen regel på en av formlene, så kan man bruke kontraksjon igjen, og man får en løkke.

Algoritmen må derfor ta hensyn til disse forholdene, og det er backtrackingen og ikketermineringen som skaper problemene. Ikkedeterminismen som er innført av muligheten til å bruke ulike regler er bare av betydning for effektiviteten til algoritmen, da uhensiktsmessig bruk av reglene kan føre til en eksponensiell vekst av søkerommet.

En annen vanskelighet, også kompleksitetsmessig, er conv-aksiomet. I praksis krever dette en tilgang til en teorembeviser, og det er derfor ønskelig å ikke sjekke om vi har et conv-aksiom oftere enn strengt nødvendig.

Vi ønsker heller ikke at tynningsregelen gjentatte ganger fjerner formler fra formelmengden slik at vi til slutt står igjen med bare en formel på hver side (der vi ikke har et aksiom).

For å unngå problemene med de strukturelle reglene, vil jeg legge en sterk begrensing på bruken av dem. Tynningsregelen skal kun fjerne duplikater av formler, og kontraskjonsregelen skal kun brukes sammen med de modale reglene. For at det skal gå bra å begrense tynningsregelen, vil jeg endre identitetsregelen slik at denne blir:

$$\frac{}{\Gamma, \varphi \Rightarrow \varphi, \Delta} \; ID$$

Med denne identitetsregelen blir det ikke nødvendig å bruke tynningsregelen for å sjekke om man har et aksiom. I tillegg har jeg endret de modale reglene fra [16], slik at den opprinnelige formelen kopieres med på høyresiden. Dermed får man en implisitt kontraskjon ved anvendelse av denne regelen, og det er ikke nødvendig å ha med en egen kontraskjonsregel. De nye modale reglene blir da:

$$\frac{\Theta^k, \Gamma^{B_k}, \Gamma \Rightarrow \Delta^k, \varphi}{\Phi_1^{\setminus k}, \Theta^k, \Gamma^{B_k} \Rightarrow \Delta^k, B_k\varphi, \Phi_2^{\setminus k}} \; LRB_k \quad \text{forutsatt at } \Delta^k, \varphi \text{ er ikke-tom}$$

Der $\Theta^{\setminus k}$ betyr at alle elementene i Θ er fri for modalitet k. Regelen for LRC_k blir tilsvarende. Dette løser problemet med ikketerminering som følge av gjentatt bruk av kontraksjons- og tynningsreglene.

Snittreglen er heller ikke mulig å implementere, siden den introduserer en ny formel φ, og vi ikke vet hvilken formel dette skal være. Men i [16] er det et snitteliminasjonsteorem for *første-ordens* formler (se avsnitt 4.3). Det vil i praksis si at dersom vi ikke har nøstede modaliteter trenger vi heller ikke snitt-regelen. Det er også slik at alle formler kan skrives om til en formel uten nøstede modaliteter [17], og vi trenger dermed ikke å implementere snitt-regelen.

Ved å fjerne de strukturelle reglene, og ved å endre på de modale reglene er den delen av termineringsproblemet som skyldes gjentatt bruk av identitetsreglene løst. Det samme gjelder problemet med snitt-regelen, siden den kan fjernes. Det forutsettes at alle formler først skrives om til en ekvivalent formel uten nøstede modaliteter, dette har jeg ikke implementert, men en algoritme for dette er beskrevet i [10]. Imidlertid introduserer den implisitte kontraksjonsregelen i de modale formlene en ny mulighet for ikke-terminering. Dette problemet har jeg ikke løst, men jeg har skissert noen mulige løsninger på dette problemet mot slutten av kapittelet.

Ved å bruke teorembeviseren selv som en teorembeviser kan man også håndtere conv-aksiomet:

$$\frac{}{B_k\varphi, C_k\psi \Rightarrow} \; CONV \quad \text{dersom } \{\neg\varphi, \neg\psi\} \text{ er konsistent}$$

Dette tilsvarer å gjøre et bevissøk etter sekventen:

$$\Rightarrow \varphi, \psi \tag{4.1}$$

Det vil være slik at conv-aksiomet er oppfylt dersom vi ikke lykkes i å finne et bevis for sekventen i 4.1. Av hensyn til effektivitet ønsker vi å utføre denne testen så sjelden som mulig.

Det siste som må tas hensyn til er dermed reglene for modalitetene, siden disse innfører backtracking. Måten jeg skal løse dette problemet på er at jeg skal bruke en stack av sekventer, der jeg skal legge alle sekventer som ennå ikke er behandlet. Sammen med sekventene skal jeg lagre informasjon om *hvorfor* de er lagt på stacken, det vil si hvilken regel (eller type regel) som ble brukt for å få frem den enkelte sekventen. Algoritmen skal jobbe med bare én gren av gangen, det blir altså et dybde-først-søk, noe som ikke vil være et problem så lenge man har en algoritme som terminerer. Reglene som har mer enn én betingelse deles inn i to typer, den ene typen er regler som har to betingelser og én konklusjon, og som altså splitter treet i to, det vil si at vi må bevise begge betingelsene for å ha et bevis for konklusjonen. Disse reglene kaller jeg *AND-regler*. Den andre typen regler er reglene for modalitetene som kan ha flere ulike betingelser, men hvor kun den ene trenger å bevises. Disse reglene kaller jeg *OR-regler*. Når treet bygges opp nedenfra og oppover, og man bruker en regel som splitter treet, skal algoritmen fortsette å jobbe med én av sekventene fra reglene som splitter, mens de andre sekventene legges til stacken. Dersom vi finner et aksiom, sier vi at grenen er lukket, og i så fall ser vi på det første elementet på stacken. Dersom dette kommer fra anvendelse av en AND-regel går vi videre og jobber med denne. Dersom den derimot kommer fra en OR-regel kan denne sekventen kastes, og vi går videre til neste element på stacken.

Dersom man kommer til en gren man ikke greier å lukke, det vil si at man bare har atomære formler og ingen av formlene på høyre side finnes på venstre side, eller at høyresiden er tom, men man ikke har et conv-aksiom, går man til første element på stacken. Dersom dette kommer fra en OR-regel går man videre og jobber med denne sekventen i stedet. Dersom det i stedet kommer fra en AND-regel kan man kaste sekventen og se på neste element i stacken.

Algoritmen er ferdig når det ikke er flere elementer igjen på stacken. Dersom man sitter igjen med en lukket gren og en tom stack, så har man et bevis. Dersom man sitter igjen med en åpen gren og en tom stack, så har man ikke et bevis.

Beskrivelse av algoritmen. Datatypen for algoritmen er et trippel som består av en boolsk verdi, *closed* som skal angi om den siste grenen

i søketreet som ble behandlet var en åpen eller lukket gren, en sekvent kalt den *aktive sekventen*, og en stack av ubehandlede sekventer. Jeg har kalt datatypen for dette trippelet for `WorkTriple` i implementasjonen:

$$\big(\texttt{Bool}, \texttt{Sequent}, \texttt{Stack}\big) \rightarrow \texttt{WorkTriple}$$

Algoritmen velger hele tiden hva den skal gjøre kun ut fra hvordan trippelet ser ut, uten at den trenger å ta hensyn til hva forrige steg var. Det er tre forskjellige muligheter: Enten så er den aktive sekventen tom, og det finnes sekventer på stacken, den aktive sekventen er tom og det er ingen elementer på stacken, eller den den aktive sekventen er ikke tom. Algoritmen starter med en sekvent, som er den aktive sekventen. Alle formlene i sekventen må være uten nøstede modaliteter. Stacken er tom i det algoritmen starter.

- Dersom den aktive sekventen ikke er tom anvender algoritmen en vilkårlig regel på denne.
  - Dersom regelen har bare én premiss så blir denne den nye aktive sekventen.
  - Dersom regelen er en *AND-regel*, så blir én av premissene den nye aktive sekventen. Den andre legges på stacken sammen med informasjon om at det var en AND-regel som ble brukt.
  - Dersom regelen er en *OR-regel*, så kan det være mange forskjellige premisser. Alle mulige premisser beregnes, én av disse blir ny aktiv sekvent, og de resterende legges én og én på stacken sammen med informasjon om at det var en OR-regel som ble brukt.
  - Dersom det viser seg at den aktive sekventen er et aksiom, så er denne grenen lukket. Vi fjerner sekventen fra den aktive sekventen og setter *closed* til `TRUE`. Deretter går vi videre og ser på første element på stacken.
  - Dersom den aktive sekventen ikke er et aksiom, og det er helt klart at det ikke er mulig å komme frem til et aksiom, så settes *closed* til `FALSE`. Vi går videre og ser på neste element på stacken.
- Dersom den aktive sekventen er tom, og det finnes elementer på stacken, så forsøker algoritmen å finne en ny aktiv sekvent fra

stacken. Den behandler elementene på stacken ulikt etter hva verdien av *closed* er:

- `TRUE`: Den siste grenen som ble behandlet er lukket, og dersom det ligger elementer på toppen av stacken som kommer fra en OR-regel, kan disse kastes. Første element som kommer fra en AND-regel blir ny aktiv sekvent.
- `FALSE`: Den siste grenen som ble behandlet er ikke lukket, og vi må finne en sekvent som kommer fra en OR-regel som vi kan fortsette med. Sekventer fra AND-regler kastes fra stacken, og vi fortsetter med neste sekvent som kommer fra en OR-regel.

- Dersom både stacken og den aktive sekventen er tom avsluttes søket. Dersom *closed* er `FALSE`, er sekventen vi startet med ikke oppfyllbar. Dersom *closed* er `TRUE`, er startsekventen oppfyllbar.

4.3 Introduksjon til Maude

Maude er et funksjonelt og deklarativt programmeringsspråk med en presist definert semantikk basert på omskrivingslogikk. Det er egnet til "rapid prototyping", og er godt egnet til simulering av distribuerte og kommuniserende systemer. Siden det har en presist definert semantikk egner det seg til formell modellering. Språket har et delspråk som er et *funksjonelt* språk, der funksjoner har unike verdier som resultat. I tillegg har det et utvidet *regelbasert* språk som gjør det enkelt å modellere ikkedeterminisme og samtidige operasjoner.

Den underliggende semantikken som er basert på omskrivingslogikk har refleksive egenskaper, det betyr at man kan uttrykke eksekvering av Maude-programmer i Maude selv. Det at den har refleksive egenskaper betyr at det finnes en endelig representert omskrivingsteori $\mathcal{U}$ (kalt den universelle teorien), som kan representere enhver omskrivingsteori $\mathcal{R}$ som en term $\overline{\mathcal{R}}$, og alle termer t, t' i $\mathcal{R}$ som termer $\overline{t}, \overline{t'}$ slik at man har følgende ekvivalens:

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash (\overline{\mathcal{R}}, \overline{t}) \rightarrow (\overline{\mathcal{R}}, \overline{t'})$$

Denne egenskapen gjør at man kan lage egne søkestrategier til Maude-programmer, ved å representere den universelle teorien i Maude, og så representere programmet i denne meta-teorien. Deretter kan man bruke meta-teorien til å gi en detaljert angivelse av hvordan programmet

vårt skal eksekveres. Det er dette meta-nivået som gjør Maude egnet til å skrive en teorem-beviser. Maude kommer med et bibliotek av funksjoner, kalt "meta-level" som er en implementasjon av denne universelle teorien, og dette gjør det mulig å styre eksekveringen av et Maude-program. Meta-level inneholder egne funksjoner som blant annet simulerer kjøringen av programmer. Dessuten finnes det funksjoner som oversetter til og fra et metarepresentert program, som kan brukes til å skrive egne "metaprogrammer". Dette gjør det mulig å skrive en egen eksekveringsstrategi for et program, noe som er av stor nytteverdi ved modellering av språk, testing av (eksekverbare) semantikker o.l. Ved hjelp av denne modulen kan man lage søkestrategiene *internt* i systemet, det vil si at strategiene kan defineres ved hjelp av utsagn i en normal Maude-modul, og denne modulen kan det også resonneres omkring på samme måte som enhver annen modul i Maude.

Maude har dessuten en effektiv innebygd søkemotor, som baserer seg på et bredde-først-søk. Søkemotoren bruker blant annet en hashtabell for å lagre alle tidligere tilstander slik at den ikke besøker samme tilstand flere ganger i løpet av et søk.

Maude har også mulighet for å definere en egen syntaks for hver operator, og denne kan være enten prefiks, postfiks, infiks eller alle kombinasjoner av disse. Dette gir fleksibilitet, og mulighet til å legge seg tett opptil det opprinnelige språket dersom vi bruker Maude til å modellere et annet språk.

Et Maudeprogram består av en rekke moduler, som er av to typer; Den ene typen er *funksjonelle moduler* som er funksjonelle programmer med likhetslogikk som underliggende semantikk. Disse består av en mengde funksjoner. Den andre typen moduler er *systemmoduler* som består av funksjoner og omskrivingsregler. Disse har omskrivingslogikk som semantikk[1].

4.4 Implementasjon

Jeg har valgt å dele min implementasjon i flere ulike moduler:

- `FORMULA` Denne modulen inneholder definisjonen av språket; av formler, sekventer og mengder av sekventer, i tillegg til noen hjelpefunksjoner for å behandle disse.

[1] Det finnes også en utvidelse av Maude kalt *Full Maude*, som gir mulighet for objekt-orienterte moduler.

Kodeeksempel 4.1: Definisjon av sorter

```
1 fmod FORMULA is
    protecting NAT .
3   sorts Atom Literal Fml FmlSet Sequent Configuration
       ModOp .
    subsorts Atom < Literal < Fml < FmlSet .
5   subsort Sequent < Configuration .
```

- `STRUTURAL_RULES` Denne modulen inneholder de strukturelle reglene i sekventkalkylen og identitetsreglene. Jeg bruker ikke alle disse reglene i søkestrategien, men jeg har implementert dem likevel, fordi jeg ønsker at implementasjonen av kalkylen skal være så tett opp til spesifikasjonen som mulig. Snittregelen er likevel ikke implementert, siden denne ikke lar seg spesifisere i Maude uten videre.
- `SEQUENT_RULES` Dette er alle reglene for konnektivene som bare har én premiss, altså de reglene som ikke fører til at bevistreet splittes.
- `AND_RULES` Dette er reglene for konnektivene som har to premisser, altså reglene som fører til at treet splittes.
- `MODAL_RULES` Denne modulen inneholder de to modale reglene, LRB_k og LRC_k. Dette er OR-reglene i algoritmen. Conv-aksiomet håndteres i modulen som utfører søket.
- `PROOF` Inneholder en rekke definisjoner av sorter og hjelpefunksjoner for bevissøket.
- `PROOF-SEARCH` Dette er modulen som inneholder selve bevissøkalgoritmen, samt noen hjelpefunksjoner for denne. Bevissøket er implementert på Maudes meta-nivå.

I gjennomgangen av programmet viser jeg små utdrag i teksten, mens hele programmet er gjengitt i appendiks A. I koden har jeg stort sett holdt meg til korte variabelnavn, da koden har en tendens til å bli lite lesbar dersom reglene blir så lange at de brekker linjene. Så langt det er mulig har jeg forsøkt å holde meg til fornuftige og konsekvente forkortelser, slik som `N` for naturlige tall, `G` for Γ, `f` for φ, `FS` for `FormulaSet` og så videre. De fullstendige variabeldeklarasjonene finnes i koden i appendiks A.

Kodeeksempel 4.2: Definisjon av språket

```
**** Definisjon av språket:
ops top bot : -> Atom [ctor format(! o)] .
*** Utsagnsvariable:
op p : Nat -> Atom [ctor] .
op not_ : Fml -> Fml [ctor format(! o d) prec 10] .
ops _/\_ _\/_ _==_ : Fml Fml -> Fml [ctor assoc
   format(d ! o d) prec 20] .
op _imp_ : Fml Fml -> Fml [ctor format(d ! o d) prec
   30] .
*** Literaler er negerte atomer
mb not A:Atom : Literal .
```

Kodeeksempel 4.3: Definisjon av modale operatorer

```
*** Modale operatorer
op B : Nat -> ModOp [ctor] .
op C : Nat -> ModOp [ctor] .
op __ : ModOp Fml -> Fml [ctor prec 15] .
op __ : ModOp Fml -> Fml [ctor prec 15] .
```

Starten på `FORMULA`-modulen finnes i eksempel 4.1. Sortdefinisjonene i starten definerer `Atom`, `Literal` og `Fml` som korresponderer med de tilsvarende begrepene i utsagnslogikk. `FmlSet` er multisett[2] av formler. `Sequent` er sorten for sekventer, mens `Configuration` er multimengder av sekventer. Sorten `ModOp` er de modale operatorene (B_k og C_k). Subsort-deklarasjonen korresponderer med delmengdeinklusjon av elementene i de enkelte sortene, det vil si at alle atomer er literaler, alle literaler er formler o.s.v.

I eksempel 4.2 finnes konstruktører for atomære formler (top, bot, og utsagnsvariable), samt induktiv definisjon av formler. Jeg har valgt å definere utsagnsvariable som en funksjon som tar et naturlig tall som parameter, dermed får jeg et uendelig antall utsagnsvariable som kan skrives på formen `p(1)`, `p(2)` etc. Den siste linja bruker et *membership axiom* til å si at negerte atomer er literaler. Siden atomer allerede er en subsort av literaler trenger vi ikke å si dette eksplisitt for positive

[2]Maude har en *idem*-attributt, for å deklarere idempotens, men slik den er implementert gir den ikketerminering når den er brukt sammen med *assoc*-attributtet for assosiativitet. Det er derfor mer hensiktsmessig å implementere multisett dersom det ikke er strengt nødvendig med mengder.

Kodeeksempel 4.4: Mengder av formler

```
     *** bagger av formler
18   op emptyFmlSet : -> FmlSet [ctor] .
     op _,_ : FmlSet FmlSet -> FmlSet
20      [ctor assoc comm id: emptyFmlSet format(d b! o
           d) prec 50] .
```

literaler. `/\`, `\/` og == er deklarert som assosiative operatorer, ved *assoc*-attributtet. Format-attributtene sier hvordan termene skal skrives ut, og prec angir presedens for operatorene slik at termer kan skrives uten parenteser. `ctor`-attributtet er en forkortelse for *constructor*, denne forandrer ikke semantikken, men den forteller at operatoren er en konstruktør, noe som øker lesbarheten og er nyttig for feilsøking.

I eksempel 4.4 finnes definisjonen av multimengder av formler, sekventer, og multimengder av sekventer. De modale operatorerene er definert i eksempel 4.3, og på samme måte som med utsagnsvariabler er disse definert som funksjoner som tar et naturlig tall som parameter. Dermed får man en uendelig mengde modale operatorer på formen `B(1)`, `B(2)` etc., og tilsvarende for C.

Resten av modulen er hjelpefunksjoner for å avgjøre om en formelmengde er tom (`empty?`), om en formel eller en mengde formler er modale atomer (`kmodalAtom?` og `kmodalAtoms?`), om en formel er fullstendig modalisert (`kModalized?`), og om en formel er fri for en modalitet (`kFree?`).

Reglene for sekventkalkylen finnes i modulene `STRUCTURAL_RULES`, `SEQUENT_RULES` og `AND_RULES`. Alle sekventkalkyle-reglene er implementert som regler i Maude. Reglene er deklarert som

```
rl [label] : term1 => term2 .
```

Meningen er at termen `term1` kan skrives om til `term2` ved anvendelse av regelen `label`. Siden kommentarer i Maude-koden kan innledes med `---` får vi en elegant implementasjon av reglene da disse kan skrives på tilnærmet samme form som de er spesifisert i avsnitt 4.1. For eksempel kan $R\neg$ reglen:

$$\frac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \neg\varphi, \Delta} \; R\neg$$

representeres med koden i eksempel 4.5.

De andre utsagnslogiske formlene er implementert på tilsvarende måte. Reglene som ikke fører til splitting av bevistreet er samlet i mo-

Kodeeksempel 4.5: Regel for $R\neg$

```
    rl [Lnot] : G , (not f) ==> D
133             => --------------
                  G ==> f , D .
```

Kodeeksempel 4.6: regelen $R\wedge$ representert i Maude

```
    rl [Rand] : G ==> ( f /\ p) , D
157            => --------------------------------
                  (G ==> f , D) | (G ==> p , D) .
```

dulen SEQUENT_RULES, mens de som fører til at treet splittes er samlet i modulen AND_RULES. Grunnen til at jeg har delt disse i to forskjellige moduler er at dette gjør det enklere å implementere søkestrategien. For de utsagnslogiske reglene som fører til splitting av treet (AND-regler) bruker jeg sorten Configuration som er definert i FORMULA. For eksempel blir $R\wedge$;

$$\frac{\Gamma \Rightarrow \varphi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi, \Delta} R\wedge$$

representert som i eksempel 4.6.

De strukturelle reglene finnes i modulen STRUCTURAL_RULES. Jeg har implementert alle reglene bortsett fra snitt-regelen. Snitt-regelen er vanskelig å implementere siden denne introduserer en ny variabel i konklusjonen og det ikke finnes noen enkel måte å automatisk finne en instansiering av denne på. I tilegg har jeg modifisert tynningsreglene slik at de bare fjerner like formler (duplikater), som i eksempel 4.7. Kontraksjonsreglene er implementert for fullstendighetens skyld, men de brukes ikke i selve bevissøket. I stedet har jeg kombinert reglene for modalitetene med en kontraksjonsregel slik jeg har beskrevet i avsnitt 4.1.

Kodeeksempel 4.7: tynningsregel representert i Maude

```
    rl [Lt] :  G , p , p ==> D
96             => -------------
                  G , p ==> D .
```

Reglene for modalitetene, LRB_k og LRC_k, er implementert i modulen `MODAL_RULES`. Conv-aksiomet er implementert sammen med resten av søkestrategien i `PROOF-SEARCH`-modulen, siden dette aksiomet er en meta-regel og følgelig må implementeres på meta-nivå. Jeg har som nevnt i avsnitt 4.1 modifisert regelen noe i forhold til formuleringen i [17]. Den nye LRB_k - regelen er:

$$\frac{\Theta^k, \Gamma^{B_k}, \Gamma \Rightarrow \Delta^k, \varphi}{\Phi_1^{\setminus k}, \Theta^k, \Gamma^{B_k} \Rightarrow \Delta^k, B_k\varphi, \Phi_2^{\setminus k}} \; LRB_k \quad \text{forutsatt at } \Delta^k, \varphi \text{ er ikke-tom}$$

Det er en rekke betingelser som må oppfylles for at denne regelen skal kunne brukes, disse er gitt av kravene til innhodet i mengdene Γ^{B_k}, Δ^k og så videre. I tillegg ønsker jeg å begrense bruken av denne regelen så mye som mulig, og vil derfor helst ikke bruke den dersom det er en annen regel som heller kunne vært brukt. Kontroll av disse betingelsene kunne i teorien vært gjort av *membership axioms* i Maude, disse ville gjort det mulig å deklarere sorter for ulike mengder av formler. Man kunne hatt én sort for mengder av modale atomer, én sort for utsagnslogiske formler, og så videre. Membership axioms kunne da blitt brukt til å gi betingelse for når en formelmengde har en bestemt sort. Problemet er at memberships ikke fungerer sammen med operatorer som er deklarert med `assoc` og `iter` attributtene, slik som `FmlSet`. Dette vil føre til at dagens versjon av Maude-interpreteren ikke terminerer når den forsøker å bestemme sorten til en term. Derfor har jeg isteden brukt en rekke hjelpefunksjoner, og implementasjonen av LRB_k finnes i kodeeksempel 4.8. Her innfører jeg en ny variabel `G` på venstresiden av `=>` i konklusjonen, denne instansieres i den siste betingelsen: `G := removeModal(Gbk)` der jeg bruker hjelpefunksjonen `removeModal` i eksempel 4.9 for å fjerne modalitetene fra formlene i `BFS`. I den første betingelsen forsøker jeg å begrense bruken av denne regelen så mye som mulig, ved at jeg sjekker at ingen av formlene består av en boolsk kombinasjon av andre formler.

Søk etter bevis. Modulen `PROOF_SEARCH` inneholder en bevissøkstrategi. Selv om Maude har en innebygd søkefunksjon har jeg valgt å implementere et eget søk. Dette gjør det mulig å skrive en dybde-først strategi, mens den innebygde strategien benytter seg av et breddeførst søk. For et stort søkerom vil et bredde-først søk bli ineffektivt. Dessuten er algoritmen jeg har beskrevet i 4.2 basert på en dybde-først strategi. I tillegg vil man få en enkel representasjon av conv-aksiomet. Sjekk av betingelsen i conv-aksiomet:

Kodeeksempel 4.8: Regel for modalitetene representert i Maude

```
    *** Reglene for modalitetene
187 crl [LRBk] : P1, Tk, Gbk ==> Dk, B(N) f, P2
             => ------------------------------
                     Tk, Gbk, G ==> Dk, f
189     if notBoolComb(P1,Tk,Gbk,Dk,P2)  *** ingen andre
          regler kan brukes
191       /\ kmodalAtoms?(N, Gbk)     *** Gbk inneholder
            kun modale
                                      *** atomer av
                                         modalitet N
193       /\ onlyB(Gbk)               *** kun B-modal
          /\ kmodalAtoms?(N, Dk)
195       /\ onlyB(Dk)                *** Samme for FS
          /\ G := removeModal(Gbk) .  *** Fjern
            modalitetene fra BFS
```

Kodeeksempel 4.9: Hjelpefunksjonen removeModal

```
209 *** Fjern modalitetene fra en formelmengde,
    *** forutsetter at alle fml er modale
211 op removeModal : FmlSet -> FmlSet .
    eq removeModal(B(N) f) = f .
213 eq removeModal(emptyFmlSet) = emptyFmlSet .
    eq removeModal(B(N)f , FS) = f , removeModal(FS) .
```

$$\frac{}{B_k\varphi, C_k\psi \Rightarrow} \; CONV \text{ dersom } \{\neg\varphi, \neg\psi\} \text{ er konsistent}$$

krever tilgang til en teorembeviser, og spesifikasjonen av conv-aksiomet blir enkel dersom man allerede har implementert en funksjon for bevissøk. Den siste fordelen med å implementere en egen søkestrategi er at man har full kontroll over søket, og det vil derfor være enklere å optimalisere strategien på et senere tidspunkt.

For å implementere de enkelte delene av søkestrategien har jeg brukt modulen `META-LEVEL`, som er en del av Maude-systemet. `META-LEVEL` er en implementasjon av den universelle teorien $\mathcal{U}$. Denne modulen gir tilgang til alle de innebygde søkefunksjonene. Disse har jeg brukt til å søke etter alle tilstander som kan nås i nøyaktig ett steg ved bruk av reglene for de modale operatorene. I tillegg inneholder modulen blant annet funksjoner for å anvende en bestemt regel på en term, for å sjekke hvilken sort en term har, og for å redusere en term til normalform (kanonisk form). Tilsammen gir disse mulighet for å gi detaljert kontroll over eksekveringen av en modul. `META-LEVEL` inneholder dessuten nødvendige definisjoner for å metarepresentere termer, og i tillegg inneholder den funksjoner for å oversette en term fra objektnivå til metarepresentasjon og vice versa.

Algoritmen jobber på en term av sorten `WorkTriple`, og denne består av en boolsk variabel som skal fortelle om den siste grenen i treet endte med et aksiom eller ikke, en `Result` som inneholder den sekventen vi jobber på i øyeblikket, og en stack som inneholder de sekventene som ikke er behandlet ennå. Sorten `Result` er deklarert i modulen `PROOF`. Den består av et par av en term og en `RuleSort`, som er en konstant som angir hvilken type regel som ble brukt:

```
op {_|_} : Term RuleSort -> Result [ctor] .
```

`WorkTriple` er definert i modulen `PROOF_SEARCH` ved:

```
op {_|_|_} : Bool Result Stack -> WorkTriple [ctor] .
```

Jeg jobber hele tiden på Maudes meta-nivå. Sekventen jeg forsøker å finne bevis for skrives om til en metarepresentert term, og jeg bruker funksjoner fra Maudes meta-level for å anvende reglene for sekventkalkylen på den metarepresenterte termen, dermed har jeg hele tiden kontroll over hvilke regler som skal brukes.

Stacken som brukes til å lagre ubehandlede sekventer er implementert i modulen `PROOF`, se eksempel 4.10, og i denne modulen finnes også en del sorter og hjelpefunksjoner som brukes i søket. Av disse er det blant annet sorten `ResultList` med tilhørende hjelpefunksjoner

Kodeeksempel 4.10: Stack av Result

```
    *** definisjon av Stack av Result
308 op emptyStack : -> Stack [ctor] .
    ops push : Stack Result -> Stack [ctor] .
310 op top : Stack -> Result .
    op pop : Stack -> Stack .
312 eq top(push(S,R)) = R .
    eq pop(push(S, R)) = S .
314 *** Fjerne tomme elementer fra en stack.
    eq push(S, emptyResult) = S .
```

Kodeeksempel 4.11: Funksjonen performProofSearch

```
349 op performProofSearch : Sequent -> Bool .
    eq performProofSearch(S:Sequent) =
351     proofSearch({ false | {upTerm(S:Sequent)| rnone} |
           emptyStack }) .
```

og funksjonene `getRuleSort` og `getTerm` som henter ut henholdsvis `RuleSort` og `Term` fra en `Result`.

Funksjonen `performProofSearch` i eksempel 4.11 er den funksjonen som kalles for å starte bevissøket. Denne tar en sekvent som input, og konverterer denne til en `WorkTriple` ved å først bruke funksjonen `upTerm` fra `META-LEVEL` for å konvertere termen til en metarepresentert term, før den kaller på funksjonen `ProofSearch` som er den funksjonen som utfører bevissøket.

Funksjonen `ProofSearch` gjennomfører selve bevissøket ut fra reglene som er gitt av algoritmen. Jeg bruker en rekke likninger med betingelser som samsvarer med de ulike alternativene for algoritmen i avsnitt 4.2. Et utdrag av funksjonen finnes i eksempel 4.12. I eksempel 4.12 er den aktive sekventen tom, og likningene sjekker hvilken type regel som ligger på toppen av stacken ved å bruke hjelpefunksjonen `getRuleSort`. Ut fra hvilken sort denne har, og hvorvidt søket i den forrige grenen endte med et aksiom eller ikke, velger den å enten kaste det øverste elementet eller å fortsette å jobbe med dette. Et annet eksempel er eksempel 4.13, der det anvendes en modal regel på den aktive sekventen dersom dette er mulig. Her er det hjelpefunksjonen `applyModal` som forsøker å anvende en modal regel på en `Result`, og den returnerer en term av sorten `ResultList` dersom det-

Kodeeksempel 4.12: Uttdrag fra proofSearch

```
    *** kaste OR-rules fra stacken:
367 ceq proofSearch({true | noresult | S}) =
        proofSearch({true | noresult | pop(S)})
369     if (getRuleSort(top(S)) == or) /\ not empty?(S) .
    *** Motsatt dersom forrige gren endte med et
       ikke-aksiom:
371 ceq proofSearch({false | noresult | S}) =
        proofSearch({false | top(S) | pop(S)})
373     if (getRuleSort(top(S)) == or) /\ not empty?(S) .
    *** kaste AND-rules fra stacken:
375 ceq proofSearch({false | noresult | S}) =
        proofSearch({false | noresult | pop(S)})
377     if (getRuleSort(top(S)) == and) /\ not empty?(S) .
```

te går bra, og en term av sorten `ResultList?` dersom det ikke går bra. `proofSearch` bestemmer altså hvilken regel som skal brukes ut fra hvordan `WorkTriple` ser ut. Jeg har ikke lagt noen føringer på i hvilken rekkefølge reglene skal brukes. Det vil være enkelt å endre på denne funksjonen slik at den velger ut regler på en mer eller mindre intelligent måte ut fra hvordan den aktive sekventen ser ut, slik det er gjort i [6]. Slik det er nå vil reglene bli valgt ikkedeterministisk.[3] Dersom man i stedet ville hatt en deterministisk seleksjon av regler, kunne man erstattet de ulike `proofSearch`-funksjonene med en nøstet if-setning. Det at jeg har implementert søket på meta-nivået i Maude gjør at jeg har mulighet for en detaljert styring på hvilke regler som til enhver tid brukes, hvis jeg ønsker det. En slik styring av reglene på metanivå kommer i tillegg til muligheten man har til å begrense bruken av reglene på objektnivå, slik jeg har gjort i implementasjonen av de modale reglene i eksempel 4.3. Styringen på objektnivå kan bare begrense muligheten til å bruke en regel, mens på metanivået har man i tillegg muligheten til å spesifisere hvilken rekkefølge reglene skal benyttes.

Resten av modulen inneholder de ulike hjelpefunksjonene for hovedfunksjonen `ProofSearch`, og internt i disse har jeg heller ikke lagt føring på hvordan likningene skal anvendes. For eksempel består funk-

[3]Semantisk er en Maude-modul en mengde likninger eller regler, men i praksis vil Maude-interpreten velge den første regelen eller formelen som kan anvendes, og eksekveringen blir derfor deterministisk, selv om man ikke bør basere programmet på at eksekveringen skjer i en bestemt rekkefølge.

Kodeeksempel 4.13: Anvendelse av en modal regel i proofSearch

```
389    *** Vi forsøker å bruke en modal regel:
       ceq proofSearch({B | R | S}) =
391        proofSearch({B | getFirst(RL) |
                pushAll(S, getRest(RL))})
393       if not empty?(R) /\
          RL := applyModal(R) /\ RL =/= emptyResultList .
```

Kodeeksempel 4.14: Hjelpefunksjonen applyRules

```
       *** applyRules: Anvend en (tilfeldig) regel på en term
416    op applyRules : Term -> Result .
       ceq applyRules(T) = R?
418      if R? := applyRule((T), 'Rnot) .
       ceq applyRules(T) = R?
420      if R? := applyRule((T), 'Land) .
       ceq applyRules(T) = R?
422      if R? := applyRule((T), 'Lnot) .
       ceq applyRules(T) = R?
424      if R? := applyRule((T), 'Ror) .
       ceq applyRules(T) = R?
426      if R? := applyRule((T), 'Rimp) .
       eq applyRules(T) = emptyResult [owise] . *** error
```

sjonen `applyRules` i eksempel 4.14 av en mengde likninger som sjekker hvilke regler fra `SEQUENT_RULES` som kan anvendes på en term, og anvender den første som passer. Dersom man ønsker en mer finkornet styring på hvilke regler som brukes kan man erstatte mengden likninger i `applyRules` med en if-setning.

For å implementere hjelpefunksjonene for søket har jeg særlig benyttet funksjonen `metaXapply` i Maudes `META-LEVEL`:

```
op metaXapply : Module Term Qid Substitution  Nat Bound Nat
    -> Result4Tuple .
```

Denne funksjonen brukes til å anvende en regel på en term. Argumentene til funksjonen er en meta-representert modul, en meta-representert term og et metarepresentert navn på en regel (representert med sorten `Qid`). `Nat` og `Bound` kan brukes til å angi på hvilken posisjon i termen omskrivingen skal skje, og det siste argumentet angir hvilken løsning man vil ha returnert. Regelnavnet repre-

Kodeeksempel 4.15: Regelen applyRule

```
431    op applyRule : GroundTerm Qid -> Result [memo] .
       eq applyRule(T,Q:Qid)
433      = {getTerm(metaXapply(['SEQUENT_RULES], T,
                Q:Qid , none, 0, unbounded, 0)) | rnone } .
```

senterer en omskrivingsregel vi ønsker å anvende på termen `Term`. Funksjonen `metaXapply` vil returnere et svar av sorten `Result4Tuple` dersom det er mulig å anvende den ønskede regelen på `Term`. Kvadruppelet `Result4Tuple` inneholder termen som er resultat av regelanvendelsen. Det er denne jeg er interessert i, og for å få ut denne fra `Result4Tuple` brukes funksjonen `getTerm`. I tillegg innehoder `Result4Tuple` informasjon om hvilken substitusjon som ble brukt, hvor i termen omskrivingen skjedde, og hvilken sort resultat-termen har. Et eksempel på anvendelse av denne funksjonen finnes i eksempel 4.15, der funksjonen `applyRule` anvender en regel på en term.

Ved anvendelse av en av de modale reglene, *OR*-reglene, trenger jeg en funksjon som gir alle mulige termer som kan nås i nøyaktig ett omskrivingssteg i modulen `MODAL-RULES`. Her bruker jeg funksjonen `metaSearch`:

```
op metaSearch : Module Term Term Condition Qid Bound Nat ~>
    ResultTriple? .
```

`metaSearch` er funksjon for å bruke Maudes innebygde bredde-først søk på en metarepresentert term. Her er argumentene en meta-representert modul, en meta-representert term som representerer starttermen, og en meta-representert term som representerer mønsteret det søkes etter. `Bound` representerer maksimal dybde for søket, og jeg begrenser søket til nøyaktig ett steg ved å sette denne til 1. `Nat` angir hvilken løsning vi er ute etter, og i funksjonen `oneStepModal` i eksempel 4.16 rekurserer jeg over denne for å finne alle termer som kan nås i eksakt ett steg ved bruk av funksjonene i modulen `MODAL_RULES`.

Avslutte søk. For at algoritmen skal fungere må det være en måte å avgjøre om en sekvent er et aksiom, og det må også være en *negativ* aksiomsjekk, slik at vi kan avslutte søket dersom vi er sikre på at den aktive sekventen ikke kan utledes til et aksiom. Den positive aksiomsjekken benytter seg av `ID`-regelen i `STRUCTURAL-RULES`-modulen:

$$\frac{}{\Gamma, \varphi \Rightarrow \varphi, \Delta}$$

Kodeeksempel 4.16: Funksjonen oneStepModal

```
    **** kalles med oneStepModal(R,0) .
486 op oneStepModal : Term Nat -> ResultList .
    ceq oneStepModal(T, N) = RL : oneStepModal(T,s N)
488     if RL := {
           getTerm(metaSearch(upModule('MODAL_RULES,false),
               T, 'C:Configuration, nil, '+, 1, N)) | or }
                   .
490 eq oneStepModal(T, N) = emptyResultList [owise] .
```

Kodeeksempel 4.17: Identitetsregelen

```
    *** Identitetsregelen
113 rl [ID] : G , p ==> p , D
          => ---------------
115          emptyConf .
```

Denne er implementert med metaregelen i eksempel 4.17 i Maude. I tillegg er det en metaregel i `PROOF_SEARCH` modulen, der `ID`-regelen anvendes for å sjekke etter et aksiom. Denne har jeg implementert som i eksempel 4.18, som bare sjekker om en anvendelse av regelen gir en tom konfigurasjon. I tillegg har jeg implementert et søk etter conv-aksiomet på metanivå, som selv benytter seg av bevissøkfunksjonen. Denne funksjonen er gjengitt i eksempel 4.19. Denne er kombinert med en metaregel i 4.20, som anvender conv-regelen på en grunnterm. Ulempen med å gjøre det på denne måten er at det går utover effektiviteten, siden resultatet av å anvende conv-regelen er at modulen `PROOF-SEARCH` blir metarepresentert. Dersom vi har nøstede modaliteter ender man dermed opp med et “tårn” av metarepresenterte `PROOF_SEARCH` moduler.

Den negative aksiomsjekken i eksempel 4.21 sjekker om én side er tom og den andre siden kun inneholder atomer, eller om begge sider inneholder et (utsagnslogisk) atom, og at disse ikke er like. I begge tilfeller er det umulig å bruke noen regler slik at man kan komme frem til et aksiom.

Denne implementasjonen av den negative aksiomsjekken vil ikke være i stand å avgjøre hvorvidt man ikke har et aksiom i alle tilfeller. For eksempel feiler programmet på sekventen: `B(1) (p(1) imp p(3)) ==> p(2)`, siden det ikke er mulig å bruke noen

Kodeeksempel 4.18: Metaregel for aksiom

```
492    **** aksiomsjekker:
       op axiom? : Result -> Bool .
494    *** sekventen kan være på formen G p ==> p D :
       *** Bruker id-regelen fra STRUCTURAL_RULES modulen
496    ceq axiom?({T | R:RuleSort}) = true if
          getTerm(metaXapply(
                           upModule('STRUCTURAL_RULES,false),
498                        T, 'ID, none, 0, unbounded, 0)) ==
                           'emptyConf.Configuration .
500    ceq axiom?({T | RS}) = true if convAxiom?(T) .
       eq axiom?(R) = false [owise] .
```

Kodeeksempel 4.19: Regel for conv-aksiomet

```
       crl [conv] : FS, B(N) f, C(N) f' ==> FS'
515             => -------------------------
                   emptyConf
517        if  kFree?(N, f) /\ kFree?(N, f') /\
             not performProofSearch(emptyFmlSet ==> not f, not
                 f') .
```

Kodeeksempel 4.20: Metaregel for conv-aksiomet

```
505    op convAxiom? : GroundTerm -> Bool .
       ceq convAxiom?(GT) = true if
507        getTerm(metaXapply(upModule('PROOF-SEARCH,false),GT,
           'conv, none, 0, unbounded, 0)) ==
509         'emptyConf.Configuration .
```

Kodeeksempel 4.21: Negativ aksiomsjekk

```
520    *** Ikke aksiom: en tom side og kun atomer på den andre
         siden:
       op nonaxiom? : Result -> Bool .
522    ceq nonaxiom?({'_==>_[G,'emptyFmlSet.FmlSet] |
          R:RuleSort})
           = true if atomic?(G) .
524    ceq nonaxiom?({'_==>_['emptyFmlSet.FmlSet,G] |
          R:RuleSort})
           = true if atomic?(G) .
526    ceq nonaxiom?({'_==>_['p[G],'p[GT]] | R:RuleSort})
           = true if  not G == GT .
528    eq nonaxiom?(R) = false [owise] .
```

regler her, samtidig som vi ikke har en venstreside som kun består av (utsagnslogiske) atomer. En mulighet for å løse dette er ved å implementere en aksiomsjekk på objektnivå som takler disse tilfellene, og så lage en ny metafunksjon som anvender denne regelen. Aksiomsjekken på metanivå må kunne sammenlikne alle elementene i to mengder for å sjekke at de ikke har noen felles elementer, i tillegg til at de må sjekke at ingen andre regler kan anvendes.

Eksempel 12 (Anvendelse av teorembeviseren)
Bevissøk etter sekventen:

$$B_1(p_1 \supset p_2) \Rightarrow B_1 p_1 \supset B_1 p_2$$

Kan utføres slik i teorembeviseren:

```
  Maude> red performProofSearch(B(1)(p(1) imp p(2)) ==> B(1)
     p(1) imp B(1) p(2)) .
2 reduce in PROOF_SEARCH : performProofSearch(B(1) (p(1) imp
     p(2)) ==> B(1) p(1) imp B(1) p(2)) .
  rewrites: 482 in 60ms cpu (155ms real) (8033
     rewrites/second)
4 result Bool: true
```

Her brukte teorembevisere 60ms cpu-tid på å regne seg frem til svaret, som er `true`.

Terminering. Ved å kreve at input skrives på normalform vil man unngå problemet med trivielle ikketerminerende input, som for eksempel:

$$B\neg Bp \Rightarrow Bp$$

Som gir opphav til bevistreet:

$$\cfrac{\cfrac{B\neg Bp \Rightarrow Bp, p}{B\neg Bp, \neg Bp \Rightarrow p}}{B\neg Bp \Rightarrow Bp} \quad (4.2)$$

I dette beviset er den første sekventen inneholdt i den siste sekventen, og det oppstår en løkke.

Det er likevel en mulighet for at det finnes et bevis som gir en løkke, siden de modale reglene kopierer med hele høyresiden. For å håndtere slike tilfeller må det legges inn ytterligere struktur i programmet, siden det ikke er mulighet til å sjekke om vi har hatt den første sekventen tidligere. En mulighet kan være å utvide sorten for formler slik at man legger til en merkelapp på hver formel. Denne kan angi om en formel har blitt kopiert i en tidligere anvendelse av en LRB_k regel. Dermed kan man sikre at denne kontraksjonen bare skjer én gang i hver gren i bevistreet, og man unngår derfor å få løkker. Det må i så fall vises at dette er tilstrekkelig med hensyn på kompletthet av kalkylen. En annen mulighet er å bruke teknikker fra andre teorembevisere for tablå-kalkyler, og legge til mekanismer for å sjekke etter løkker i bevistreet, slik at vi avslutter søket i en gren dersom vi får en sekvent i treet som vi har hatt før. Begge disse løsningene vil kreve en utvidelse av programmet, siden man får behov for å lagre mer informasjon sammen med sekventene.

En fordel med å implementere sekventkalkylen i Maude har vært i forhold til spesifikasjonen av syntaksen og reglene for sekventkalkylen. Siden det er enkelt å spesifisere en egen syntaks med mulighet for både prefiks-, postfiks- og "miksfiks"-operatorer har det også vært enkelt å spesifisere språket for logikken. Formuleringen av reglene blir naturlig, og koden blir både enkel å forstå og det blir lett å søke etter feil, siden implementasjonen av reglene ligger såpass tett opptil spesifikasjonen i sekventkalkylen. Modulene `FORMLA`, `STRUCTURAL_RULES`, `SEQUENT_RULES` og `AND_RULES` er lette å lese og de er pålitelige. Dersom reglene stemmer overens med spesifikasjonen, er det også lett å oppnå sunnhet. Søkestrategien bruker reglene slik de er spesifisert, og ingen andre regler enn disse. Kompletthet er det litt verre å garantere, siden

det kan være vanskelig å sjekke at søkestrategien bruker de riktige reglene, og at den sjekker alle muligheter før den konkluderer med at det ikke finnes et bevis.

En slik implementasjon gir også muligheter for detaljert kontroll over søkestrategien. Ved å endre på algoritmen kan den gjøres mer effektiv, for eksempel kan man se på hvilken rekkefølge av anvendelsen av reglene som er mest hensiktsmessig, etter samme modell som det er gjort i [6]. Det vil også være mulig å teste ut flere ulike strategier for å finne den mest effektive, slik at man kan lage et godt grunnlag for en effektiv implementasjon i et imperativt programmeringsspråk.

En annen fordel med en implementasjon i Maude er at man kan skape et skille mellom spesifikasjonen av kalkylen og implementasjonen av søkestrategien. Implementasjonen av kalkylen, som består av modulene `FORMULA`, `SEQUENT_RULES`, `AND_RULES` og `MODAL_RULES` ligger tett opptil spesifikasjonen, og de forutsetter ingen ting om søkestrategien. Min søkestrategi forutsetter derimot at man har kjennskap til kalkylen og implementasjonen av denne. I [15] er det gjort et liknende arbeid med å implementere en søkestrategi for CCS i Maude, og her er søkestrategien mer uavhengig av implementasjonen av kalkylen, ved at den selv (nesten) er i stand til å oppdatere seg selv dersom man endrer på kalkylen. Jeg antar at det vil være mulig å gjøre endringer på min søkestrategi slik at skillet mot kalkylen blir ennå klarere, for eksempel ved at den kunne hente ut navnene på de ulike reglene på egenhånd, og at den som implementerer kalkylen bare trenger å angi hvorvidt en regel er en *AND*-regel eller en *OR*-regel, for eksempel ved å plassere reglene i riktig modul. Men man vil ikke kunne oppnå fullstendig skille, siden CONV-aksiomet er et meta-aksiom og dermed *må* implementeres på meta-nivå.

En mulig optimalisering kan kanskje være å endre modalitetsreglene til:

$$\frac{\Theta^{C_k}, \Gamma^{B_k}, \Gamma \Rightarrow \Delta^k, \varphi}{\Phi_1^{\setminus k}, \Theta^{C_k}, \Gamma^{B_k} \Rightarrow \Delta^k, B_k\varphi, \Phi_2^{\setminus k}}\ LRB_k \quad \text{forutsatt at } \Delta^k, \varphi \text{ er ikke-tom}$$

Dette er en grådig versjon av den regelen jeg har implementert; den fjerner mest mulig på venstresiden.

Algoritmen gir oss ikke det fullstendig bevistreet, og i enkelte sammenhenger kan man tenke seg at dette kunne vært nyttig. For eksempel ville dette muliggjort en (manuell) kontroll av at resultatet var korrekt. I en tidligere versjon av programmet forsøkte jeg å lagre alle

mulige bevistrær underveis, for så å hente ut *en* versjon av dette idet algoritmen terminerte med et bevis. Dette viste seg å være vanskelig å implementere i Maude, siden dette ga svært store og uoversiktlige termer. Men det er mulig at det finnes andre måter å gjøre dette på som kunne gitt ut et bevistre på en hensiktsmessig måte. Kanskje man kunne lagt til en ekstra variabel i `WorkTriple` som inneholdt et tre, og hvor sekventer ble lagt til bare etter at de ble brukt i bevissøket. Dermed ville et tre bli bygd opp samtidig med at man utførte bevissøket.

Kapittel 5

Videre arbeid

Jeg vil i dette kapittelet skissere noen muligheter for videre arbeid som kan gjøres med i utgangspunkt i dette arbeidet.

Bevissøket. Her må man gjøre ferdig søkemotoren, løse termineringsproblemet og kombinere søkemotoren med en "normaliseringsalgoritme" slik at man ikke trenger å mate denne med formler på normalform. Normaliserings-algoritmen er også implementert i Maude[11]. Dersom denne modifiseres slik at output er på samme format som input til mitt program, vil man kunne kombinere disse slik at input kan være på vilkårlig form.

For en videre anvendelse mot en database ville det vært interessant å se på hvorvidt det er hensiktsmessig å representere dataene i faktaspråket definert i 2.2 i en relasjonsdatabase. Dermed kan man dra nytte av effektivitet i eksisterende DBMSer med uttrykkskraften i språkene definert i kapittel 3. For en praktisk anvendelse av dette ville det også være interessant å implementere teorembeviseren i et imperativt språk med innebygd støtte for relasjonsdatabaser, for eksempel Java.

Utvide databasen. For å få en komplett database må man se videre på hvordan TELL-funksjonen(e) kan se ut. Spesielt interessant vil det være å se på hvilken uttrykkskraft man har i TELL-språket, for eksempel flere muligheter til å lagre defaulter direkte i databasen, muligheter til å uttrykke CWA ved en TELL-setning.

Det er også nødvendig å se på hvordan Æ-databaser med flere modeller kan behandles. Enten ved å la svarene på ASK-operasjoner ta

høyde for dette, eller ved at man utformer TELL slik at man ikke kan komme i en situasjon der man har flere modeller.

Det ville også vært interessant med en TELL-operator som garanterte en konsistent database.

Multiagent-utvidelse av Æ. Utvide Æ og implementasjonen slik at man kombinerer multi-agent systemet med konfidensnivåer. Det jobbes med en versjon av Æ som både har konfidensnivåer og flere agenter. Det ville vært interessant å se på hvordan dette kan brukes i en database-sammenheng.

Se på en mulig anvendelse innen semantisk web. Semantisk web er et rammeverk som er en utvidelse av dagens www. Det er et problem i dagens web at informasjon ikke er lesbar for maskiner, og man må derfor ty til teknikker som språkanalyse for å søke i websider. Resultatet er at det er vanskelig å finne informasjon på nettet, og det er vanskelig å bruke informasjon maskinelt. For å løse dette problemet har man begynt å lage et rammeverk som skal gjøre det mulig for maskiner å behandle informasjon som ligger på nettet. Semantisk web tar utgangspunkt i XML-formatet, og tar også utgangspunkt i forskning blant annet fra kunstig intelligens og kunnskapsrepresentasjon. "Semantisk" i semantisk web må tolkes som "maskinlesbart", på en eller annen måte skal "betydning" lagres samen med tekst slik at tvetydigheter i språket (som mennesker ikke har problemer med å tolke) ikke skaper problemer. Målet er at data skal kunne deles av flere applikasjoner og grupper.

"The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation." – Tim Berners-Lee, [1].

Semantisk web bygger på en lagdeling som i figur 5.1.
De to nederste lagene er XML og XML-schema lagene, som gir mulighet for strukturert representasjon av tekst. De neste lagene er RDF (resource descripition framework) og RDF-schema, som er uttrykkskraftige data-modelleringsspråk. RDF-laget er en datamodell for å referere til objekter ("ressurser") og for å beskrive hvordan de er relatert. RDF schema er et språk for å beskrive egenskaper ved RDF-ressurser, og for å beskrive klasser av disse.

På toppen av RDF-laget har man også laget en *description logic*, kalt OWL. Dette er systemer som er utviklet spesielt for å konstruere

XML for strukturerte dokumenter
Basic assertion model
schema language
Ontology layer
conversion language
evolution rules language
logical layer

Tabell 5.1: Lagdelingen i semantisk web

ontologier, en formell representasjon av et begrepsapparat innenfor et område. Tanken er at slike ontologier skal gjøre det mulig å utveksle data mellom datamaskiner eller web-områder, ved at man kan finne én felles begrepsforståelse innenfor en slik ontologi. Innenfor dette området har man kommet langt, og det finnes både teorembevisere for å gjøre formell resonnering omkring ontologier, og det finnes verktøy for å designe ontologier (for eksempel Protégé). Man har også tatt dette i bruk i praksis, blant annet finnes det en ontologi over medisinuttrykk som inneholder 300000 oppføringer.

Tilsammen gjør de nederste lagene det mulig å presentere informasjon i en form som er lesbar både for mennesker og for maskiner. Ved hjelp av eksisterende teknologier slik som XHTML og CSS er det mulig å lage strukturerte dokumenter og å presentere disse i et pent og lesbart format. Samtidig kan man koble informasjonen i dokumentet opp mot en ontologi, slik at dataprogrammer som skal hente ut informasjon kan slå opp i ontologien for å finne betydningen til teksten. Dermed slipper man å bruke løsninger som automatisk tekstgjenkjenning og språkanalyse, som kan være alt for upresise.

På toppen av disse lagene er det at meningen at man kan lage søkeverktøyer eller andre programmer som kan hente inn og ta i bruk informasjon som ligger i lagene under. Dette kan være alt fra bedre søkemotorer, til "intelligente" autonome agenter som selv henter inn og prosesserer informasjon og som kan kommunisere med andre slike agenter for å utføre en oppgave.

Man tenker seg også systemer der det lagres data og slutningsregler sammen, for eksempel ved at logiske regler representeres i et XML-format og lagres sammen med dataene. Ved hjelp av dette kan man forestille seg at en slik agent kan "lære seg" å resonnere omkring helt nye data.

Her vil det også være interessant å studere hvilken nytte vi kan ha av spørrespråk med stor uttrykkskraft, spesielt med tanke på anven-

delse i intelligente agenter. Her har vi mange ulike datakilder, kompliserte datastrukturer, og ofte ustrukturerte data, og tradisjonelle søketeknologier kommer derfor til kort, i alle fall dersom man ikke ønsker å konvertere data til en fast struktur før man gjennomfører søk.

Siden Æ har en naturlig multi-agent utvidelse kan det være naturlig å se på hvordan vi kan bruke dette databasespråket til å spørre mot en distribuert database, og det vil være en naturlig utvidelse å se om det kan være mulig å bruke databasespråket som et spørrespråk mot semantisk web.

Det som da må gjøres er å definere en utvidelse av språkhierarkiet som inkluderer description-logic språket OWL-DL, og så definere TELL og ASK mot dette språket.

Et eksempel på kombinasjon av ikkemonotone formalismer og ontologier er [3]. Her er det kombinert answer set programming og semantisk web.

Tillegg A

Fullstendig kode

Her følger fullstendig listing av koden:

Kodeeksempel A.1: Hele programmet

```
fmod FORMULA is
  protecting NAT .
  sorts Atom Literal Fml FmlSet Sequent Configuration
     ModOp .
  subsorts Atom < Literal < Fml < FmlSet .
  subsort Sequent < Configuration .

  **** Definisjon av språket:
  ops top bot : -> Atom [ctor format(! o)] .
  *** Utsagnsvariable:
  op p : Nat -> Atom [ctor] .
  op not_ : Fml -> Fml [ctor format(! o d) prec 10] .
  ops _/\_ _\/_ _==_ : Fml Fml -> Fml [ctor assoc
     format(d ! o d) prec 20] .
  op _imp_ : Fml Fml -> Fml [ctor format(d ! o d) prec
     30] .
  *** Literaler er negerte atomer
  mb not A:Atom : Literal .

  *** bagger av formler
  op emptyFmlSet : -> FmlSet [ctor] .
  op _,_ : FmlSet FmlSet -> FmlSet
        [ctor assoc comm id: emptyFmlSet format(d b! o
           d) prec 50] .

  *** Sequents and multisets (configurations) of sequents
```

```
    op _==>_ : FmlSet FmlSet -> Sequent [ctor format(d m! o
        d) prec 55] .
24  op emptyConf : -> Configuration [ctor] .
    op _|_ : Configuration Configuration ->
26      Configuration [ctor assoc comm id: emptyConf prec
           60] .

28  *** Modale operatorer
    op B : Nat -> ModOp [ctor] .
30  op C : Nat -> ModOp [ctor] .
    op __ : ModOp Fml -> Fml [ctor prec 15] .
32  op __ : ModOp Fml -> Fml [ctor prec 15] .

34  vars FS FS' : FmlSet .
    vars F F' : Fml .
36  vars N N' : Nat .

38  op empty? : FmlSet -> Bool .
    eq empty?(emptyFmlSet) = true .
40  eq empty?(F , FS) = false .

42  *** består en formelmengde kun av atomer?
    op atomic? : FmlSet -> Bool .
44  eq atomic?(p(N)) = true .
    eq atomic?(p(N), FS) = atomic?(FS) .
46  eq atomic?(FS) = false [owise] .

48  *** Operasjoner på modale formler:
    *** er Fml a et modalt atom av modalitet k?
50  op kmodalAtom? : Nat Fml -> Bool .
    ceq kmodalAtom?(N:Nat, B(N') F:Fml) = true if N:Nat =
       N':Nat .
52  ceq kmodalAtom?(N:Nat, C(N') F:Fml) = true if N:Nat =
       N':Nat .
    eq kmodalAtom?(N:Nat, F:Fml) = false [owise] .
54
    *** inneholder FmlSet kun modale atomer av modalitet
       Nat?
56  op kmodalAtoms? : Nat FmlSet -> Bool .
    eq kmodalAtoms?(N, emptyFmlSet) = true .
58  eq kmodalAtoms?(N, F) = kmodalAtom?(N, F) .
    eq kmodalAtoms?(N, F , FS) = kmodalAtom?(N, F) and
       kmodalAtoms?(N, FS) .
```

```
60
    *** er Fml en fullstendig modalisert formel av
       modalitet k?
62  op kModalized? : Nat Fml -> Bool .
    eq kModalized?(N , F /\ FS) = kmodalAtom?(N, F) and
       kModalized?(N, FS) .
64  eq kModalized?(N , F \/ FS) = kmodalAtom?(N, F) and
       kModalized?(N, FS) .
    eq kModalized?(N , F == FS) = kmodalAtom?(N, F) and
       kModalized?(N, FS) .
66  eq kModalized?(N , F imp FS) = kmodalAtom?(N, F) and
       kModalized?(N, FS) .
    eq kModalized?(N, B(N') F) = N == N' .
68  eq kModalized?(N, C(N') F) = N == N' .
    eq kModalized?(N, F) = false [owise] .
70
    *** er Fml fri for modalitet k?
72  op kFree? : Nat Fml -> Bool .
    eq kFree?(N, p(N')) = true .
74  eq kFree?(N, p(N') /\ F) = kFree?(N, F) .
    eq kFree?(N, not F) = kFree?(N, F) .
76  eq kFree?(N, B(N')F) = not kModalized?(N, B(N') F) .
    eq kFree?(N, B(N')F /\ F') = not kModalized?(N,
       B(N')F) and kFree?(N, F') .
78  eq kFree?(N, B(N')F \/ F') = not kModalized?(N,
       B(N')F) and kFree?(N, F') .
    eq kFree?(N, B(N')F == F') = not kModalized?(N,
       B(N')F) and kFree?(N, F') .
80  eq kFree?(N, B(N')F imp F') = not kModalized?(N,
       B(N')F) and kFree?(N, F') .
    eq kFree?(N, C(N')F) = not kModalized?(N, C(N')F) .
82  eq kFree?(N, C(N')F /\ F') = not kModalized?(N,
       C(N')F) and kFree?(N, F') .
    eq kFree?(N, C(N')F \/ F') = not kModalized?(N,
       C(N')F) and kFree?(N, F') .
84  eq kFree?(N, C(N')F == F') = not kModalized?(N,
       C(N')F) and kFree?(N, F') .
    eq kFree?(N, C(N')F imp F') = not kModalized?(N,
       C(N')F) and kFree?(N, F') .
86 endfm

88 **** Rules for the sequent calculus
   mod STRUCTURAL_RULES is
```

```
90      protecting FORMULA .
        vars G  D : FmlSet .
92      vars p : Fml .

94      *** Tynningsregler, modifisert for å kun fjerne
           duplikater:
        rl [Lt] :  G , p , p ==> D
96              => --------------
                   G , p ==> D .
98
        rl [Rt] :  G ==> p , p , D
100            => --------------
                  G ==> p , D .
102
        *** Kontraksjon
104     rl [Lc] :  G , p ==> D
               => ----------------
106               G , p , p ==> D .

108     rl [Rc] :  G ==> p , D
               => ----------------
110               G ==> p , p , D .

112     *** Identitetsregelen
        rl [ID] : G , p ==> p , D
114           => ---------------
                 emptyConf .
116
        *** snitt-regelen er ikke implementert
118 endm

120 *** Alle ikke-splittende sekvent-regler.
    mod SEQUENT_RULES is
122     protecting FORMULA .

124     vars G D : FmlSet .
        vars f p : Fml .
126
        **** Regler for de logiske konnektivene
128     rl [Rnot] : G ==> (not f) , D
                => ---------------
130                G , f ==> D .
```

```
132     rl [Lnot] : G , (not f) ==> D
               => ---------------
134                G ==> f , D .

136     rl [Land] : G , (f /\ p) ==> D
              => ------------------
138                G , f , p ==> D .

140     rl [Ror] : G ==> (f \/ p) , D
             => ------------------
142               G ==> f , p , D .

144     rl [Rimp] : G ==> (f imp p) , D
             => -----------------------
146                G , f ==> p , D .
    endm
148
    *** regler som inneholder splittende regler (AND-regler)
150 mod AND_RULES is
        protecting FORMULA .
152
        vars G D : FmlSet .
154     vars f p : Fml .

156     rl [Rand] : G ==> ( f /\ p) , D
              => ---------------------------------
158                (G ==> f , D) | (G ==> p , D) .

160     rl [Lor] : G , (f \/ p) ==> D
             => ---------------------------------
162               (G  , f ==> D) | (G , p ==> D) .

164     rl [Limp] : G , (f imp p) ==> D
              => ---------------------------------
166                (G ==> f , D ) | (G , p ==> D) .

168     rl [Leq] : G , (f == p) ==> D
             => ------------------------------------------
170               (G , f , p ==> D) | (G ==> f , p , D) .

172     rl [Req] : G ==> (f == p) , D
             => ------------------------------------------
174               (G , f ==> p , D) | (G , p ==> f , D) .
```

```
endm
176
*** Dette er reglene for de modale operatorene (OR-reglene
    i
178 *** beskrivelsen av algoritmen).
mod MODAL_RULES is
180     protecting FORMULA .
        vars P1 P2 G Tk Gbk Gck Dk FS : FmlSet .
182     vars F f  : Fml .
        vars N N' : Nat .
184     vars S : Sequent .

186     *** Reglene for modalitetene
        crl [LRBk] : P1, Tk, Gbk ==> Dk, B(N) f, P2
188              => -----------------------------
                          Tk, Gbk, G ==> Dk, f
190         if notBoolComb(P1,Tk,Gbk,Dk,P2)  *** ingen andre
               regler kan brukes
              /\ kmodalAtoms?(N, Gbk)      *** Gbk inneholder
                 kun modale
192                                        *** atomer av
                                              modalitet N
              /\ onlyB(Gbk)                *** kun B-modal
194           /\ kmodalAtoms?(N, Dk)
              /\ onlyB(Dk)                 *** Samme for FS
196           /\ G := removeModal(Gbk) .   *** Fjern
                 modalitetene fra BFS

198     *** LRCk er identisk med LRBK:
        crl [LRCk] : P1, Tk, Gck ==> Dk, C(N) f, P2
200              => -----------------------------
                          Tk, Gck, G ==> Dk, f
202         if notBoolComb(P1,Tk,Gck,Dk,P2)
              /\ kmodalAtoms?(N, Gck)
204           /\ onlyC(Gck)
              /\ kmodalAtoms?(N, Dk)
206           /\ onlyC(Dk)
              /\ G := removeModal(Gck) .
208
        *** Fjern modalitetene fra en formelmengde,
210     *** forutsetter at alle fml er modale
        op removeModal : FmlSet -> FmlSet .
212     eq removeModal(B(N) f) = f .
```

```
    eq removeModal(emptyFmlSet) = emptyFmlSet .
214 eq removeModal(B(N)f , FS) = f , removeModal(FS) .

216 *** Er en formle en modal formel av modalitet B?
    op bFormula : Fml  -> Bool .
218 eq bFormula(p(N)) = false .
    eq bFormula(C(N)f) = false .
220 eq bFormula(B(N)f) = true .
    *** består FmlSet kun av fml med modalitet B?
222 op onlyB : FmlSet -> Bool .
    eq onlyB(emptyFmlSet) = true .
224 eq onlyB(F , FS) = bFormula(F) and onlyB(FS) .

226 *** Er en formle en modal formel av modalitet C?
    op cFormula : Fml  -> Bool .
228 eq cFormula(p(N)) = false .
    eq cFormula(C(N)f) = false .
230 eq cFormula(C(N)f) = true .
    *** består FmlSet kun av fml med modalitet C?
232 op onlyC : FmlSet -> Bool .
    eq onlyC(emptyFmlSet) = true .
234 eq onlyC(F , FS) = cFormula(F) and onlyC(FS) .

236 *** Sann dersom FmlSet ikke er en boolsk kombinasjon
    *** av andre formler.
238 op notBoolComb : FmlSet -> Bool .
    eq notBoolComb(emptyFmlSet) = true .
240 eq notBoolComb(p(N), FS) = notBoolComb(FS) .
    eq notBoolComb(B(N) F, FS) = notBoolComb(FS) .
242 eq notBoolComb(F /\ f, FS) = false .
    eq notBoolComb(F \/ f, FS) = false .
244 eq notBoolComb(F == f, FS) = false .
    eq notBoolComb(F imp f, FS) = false .
246 eq notBoolComb(not f, FS) = false .
endm
248
*** Modul med alle sorter for bevis, aksiome etc.
250 mod PROOF is
    protecting META-LEVEL .
252 protecting FORMULA .

254 *** Sorter for å representere resultat fra anvendelse
       av regler
```

```
    *** samt lister og stacker av disse.
256 sorts Result Result? ResultList RuleSort Axiom Stack .
    subsort Axiom < Sequent .
258 subsort Result < Result? . *** feil-sort for result
    subsort Result < ResultList .
260
    *** deklarasjon av sorten Result:
262 op noresult : -> Result [ctor] .
    op {_|_} : Term RuleSort -> Result [ctor] .
264
    op emptyResult : -> Result? [ctor] . *** denne brukes
       til
266                                           *** feilhåndtering
    *** Variabeldeklarasjoner
268 vars T : Term .
    vars RS : RuleSort .
270 vars R : Result .
    vars RL : ResultList .
272 vars S : Stack .

274 *** Operasjoner på result:
    *** Hente ut en term:
276 op getTerm : Result -> Term .
    eq getTerm({ T | RS }) = T .
278 *** Hente ut rulesort
    op getRuleSort : Result -> RuleSort .
280 eq getRuleSort({T | RS}) = RS .
    *** sjekke om et resultat er "tomt"
282 op empty? : Result -> Bool .
    eq empty?(noresult) = true .
284 eq empty?({T | RS}) = false .

286 *** liste av Result:
    op emptyResultList : -> ResultList [ctor] .
288 op _:_ : ResultList ResultList ->
        ResultList [ctor assoc gather (e E) id:
           emptyResultList] .
290
    *** Operasjoner på ResultList:
292 *** Hente ut det første elementet
    op getFirst : ResultList -> Result .
294 eq getFirst(emptyResultList) = noresult .
    eq getFirst(R : RL) = R .
```

```
296     *** Lista minus det første elementet
        op getRest : ResultList -> ResultList .
298     eq getRest(emptyResultList) = emptyResultList .
        eq getRest(R) = emptyResultList .
300     eq getRest(R : RL) = RL .

302     *** Konstruktør for RuleSort:
        op and : -> RuleSort [ctor] .   *** For AND-regler
304     op or : -> RuleSort [ctor] .    *** For OR-regler
        op rnone : -> RuleSort [ctor] . *** For ikke-splittende
            regler
306
        *** definisjon av Stack av Result
308     op emptyStack : -> Stack [ctor] .
        ops push : Stack Result -> Stack [ctor] .
310     op top : Stack -> Result .
        op pop : Stack -> Stack .
312     eq top(push(S,R)) = R .
        eq pop(push(S, R)) = S .
314     *** Fjerne tomme elementer fra en stack.
        eq push(S, emptyResult) = S .
316
        *** er Stack tom?
318     op empty? : Stack -> Bool .
        eq empty?(emptyStack) = true .
320     eq empty?(push(S,R)) = false .

322     *** pushe alle elementene i en liste over på en stack.
        op pushAll :  Stack ResultList -> Stack .
324     eq pushAll(S, emptyResultList) = S .
        eq pushAll(S, R : RL) =
326         pushAll(push(S, R), RL) .
        eq pushAll(S, R) = push(S, R) .
328 endm

330 mod PROOF_SEARCH is
        protecting PROOF .
332     including META-LEVEL .

334     *** datatype for algoritmen:
        sorts WorkTriple .
336     op {_|_|_} : Bool Result Stack -> WorkTriple [ctor] .
```

```
338     var N : Nat .
        var B : Bool .
340     var T : Term .
        vars R R? : Result .
342     vars G D GT : GroundTerm .
        vars RS : RuleSort .
344     vars RL : ResultList .
        vars S : Stack .
346
        *** algoritmen starter med en sekvent som input. Denne
           må
348     *** konverteres til meta-nivå
        op performProofSearch : Sequent -> Bool .
350     eq performProofSearch(S:Sequent) =
            proofSearch({ false | {upTerm(S:Sequent)| rnone} |
               emptyStack }) .
352
        *** proofSearch gjennomfører bevissøket ut fra reglene
           gitt av
354     *** algoritmen:
        op proofSearch : WorkTriple -> Bool .
356     *** dersom både aktiv sekvent og stacken er tom så er
           resultatet
        *** gitt av B
358     eq proofSearch({B | noresult | emptyStack}) = B .
        **** dersom aktiv sekvent er tom må vi søke i stacken:
360     *** dersom aktiv sekvent er tom og søket i forrige gren
            endte med et
        *** aksiom, og kaster vi alle elementer fra stacken som
            kommer fra
362     *** en or rule .
        ceq proofSearch({true | noresult | S}) =
364         proofSearch({true | top(S) | pop(S)})
            if (getRuleSort(top(S)) == and) /\ not empty?(S) .
366     *** kaste OR-rules fra stacken:
        ceq proofSearch({true | noresult | S}) =
368         proofSearch({true | noresult | pop(S)})
            if (getRuleSort(top(S)) == or) /\ not empty?(S) .
370     *** Motsatt dersom forrige gren endte med et
           ikke-aksiom:
        ceq proofSearch({false | noresult | S}) =
372         proofSearch({false | top(S) | pop(S)})
            if (getRuleSort(top(S)) == or) /\ not empty?(S) .
```

```
*** kaste AND-rules fra stacken:
ceq proofSearch({false | noresult | S}) =
    proofSearch({false | noresult | pop(S)})
    if (getRuleSort(top(S)) == and) /\ not empty?(S) .
**** Dersom aktiv sekvent ikke er tom må vi jobbe med
   den:
*** vi forsøker å bruke er regel som ikke splitter:
ceq proofSearch({B | R | S}) =
            proofSearch({B | R':Result | S})
if not empty?(R) /\ R':Result := applyOneRule(R) .
*** vi forsøker å bruke en regel som splitter:
ceq proofSearch({B | R | S}) =
            proofSearch({B | getFirst(RL) |
            pushAll(S, getRest(RL))})
            if not empty?(R) /\
            RL := applyAnd(R) .
*** Vi forsøker å bruke en modal regel:
ceq proofSearch({B | R | S}) =
     proofSearch({B | getFirst(RL) |
          pushAll(S, getRest(RL))})
    if not empty?(R) /\
    RL := applyModal(R) /\ RL =/= emptyResultList .
*** Dersom result inneholder et aksiom avslutter vi
   søket og
*** setter b til true
ceq proofSearch({B | R | S}) =
    proofSearch({true | noresult | S})
    if axiom?(R) .
*** Dersom result inneholder et ikke-aksiom avslutter
   vi søket og
*** setter b til false
ceq proofSearch({B | R | S}) =
    proofSearch({false | noresult | S})
    if nonaxiom?(R) .
*** Bruke tynningsregelen til å fjerne kopier av
   formler:
ceq proofSearch({B | R | S}) =
            proofSearch({B | R':Result | S})
if not empty?(R) /\ R':Result := applyStructural(R) .

**** Hjelpefunksjer for søk:
*** anvende en tilfeldig (ikkesplittende) regel på en
   result
```

```
412    op applyOneRule : Result -> Result .
       eq applyOneRule(R) = applyRules(getTerm(R)) .
414
       *** applyRules: Anvend en (tilfeldig) regel på en term
416    op applyRules : Term -> Result .
       ceq applyRules(T) = R?
418      if R? := applyRule((T), 'Rnot) .
       ceq applyRules(T) = R?
420      if R? := applyRule((T), 'Land) .
       ceq applyRules(T) = R?
422      if R? := applyRule((T), 'Lnot) .
       ceq applyRules(T) = R?
424      if R? := applyRule((T), 'Ror) .
       ceq applyRules(T) = R?
426      if R? := applyRule((T), 'Rimp) .
       eq applyRules(T) = emptyResult [owise] . *** error
428
       *** applyRule er funksjon for å anvende en regel
          ,angitt av Qid,
430    *** på en term, i modulen SEQUENT_RULES
       op applyRule : GroundTerm Qid -> Result [memo] .
432    eq applyRule(T,Q:Qid)
         = {getTerm(metaXapply(['SEQUENT_RULES], T,
434              Q:Qid , none, 0, unbounded, 0)) | rnone } .

436    *** tynningsreglene må være med for at den negative
          aksiomsjekken
       *** skal fungere
438    op applyStructural : Result -> Result .
       ceq applyStructural({T | RS}) = R?
440                            if R? := applyOneStructural((T),
                                  'Lt) .
       ceq applyStructural({T | RS}) = R?
442                            if R? := applyOneStructural((T),
                                  'Rt) .
       eq applyStructural({T | RS}) = emptyResult [owise] .
444
       op applyOneStructural : GroundTerm Qid -> Result [memo]
          .
446    eq applyOneStructural(T, Q:Qid)
          = {getTerm(metaXapply(['STRUCTURAL_RULES], T,
448              Q:Qid, none, 0, unbounded, 0)) | rnone } .
```

```
450     *** anvende en tilfeldig (splittende, and) regel på en
        *** result. Denne returnerer metarepresenterte
           configurations, må
452     *** ha resultlist!
        op applyAnd : Result -> ResultList .
454
        *** anvend regelen qid på term og returner resultlist.
           kalles med
456     *** nat=0
        op applyOneAnd : GroundTerm Qid -> Result [memo] .
458     eq applyOneAnd(T, Q:Qid)
            = {getTerm(metaXapply(['AND_RULES], T,
460                   Q:Qid, none, 0, unbounded, 0)) | and } .
        ceq applyAnd({T | RS}) = split(R?)
462                                if R? := applyOneAnd((T),
                                      'Rand) .
        ceq applyAnd({T | RS}) = split(R?)
464                                if R? := applyOneAnd((T), 'Lor)
                                      .
        ceq applyAnd({T | RS}) = split(R?)
466                                if R? := applyOneAnd((T),
                                      'Limp) .
        ceq applyAnd({T | RS}) = split(R?)
468                                if R? := applyOneAnd((T), 'Leq)
                                      .
        ceq applyAnd({T | RS}) = split(R?)
470                                if R? := applyOneAnd((T), 'Req)
                                      .
        eq applyAnd(R) = emptyResult [owise] .
472
        *** splitte en metarepresentert configuration til en
           resultlist:
474     op split : Result -> ResultList .
        eq split({'_|_[G,GT] | R:RuleSort }) =
476          {G | R:RuleSort} : {GT | R:RuleSort } .
        *** generere alle en-stegs omskrivinger med
           modal-regler fra en
478     *** term
        op applyModal : Result -> ResultList .
480     eq applyModal(R) = oneStepModal(getTerm(R),0) .

482     *** her bruker jeg metaSearch til å finne alle
           omskrivinger som
```

```
    *** kan nås i nøyaktig ett steg i modulen MODAL_RULES
       modul.  Nat:
484 *** løsningsnummer.
    **** kalles med oneStepModal(R,0) .
486 op oneStepModal : Term Nat -> ResultList .
    ceq oneStepModal(T, N) = RL : oneStepModal(T,s N)
488     if RL := {
           getTerm(metaSearch(upModule('MODAL_RULES,false),
                T, 'C:Configuration, nil, '+, 1, N)) | or }
                    .
490 eq oneStepModal(T, N) = emptyResultList [owise] .

492 **** aksiomsjekker:
    op axiom? : Result -> Bool .
494 *** sekventen kan være på formen G p ==> p D :
    *** Bruker id-regelen fra STRUCTURAL_RULES modulen
496 ceq axiom?({T | R:RuleSort}) = true if
       getTerm(metaXapply(
                         upModule('STRUCTURAL_RULES,false),
498                      T, 'ID, none, 0, unbounded, 0)) ==
                         'emptyConf.Configuration .
500 ceq axiom?({T | RS}) = true if convAxiom?(T) .
    eq axiom?(R) = false [owise] .
502
    *** egen regel for conv-aksiomet:
504 vars NAT NAT' : GroundTerm .
    op convAxiom? : GroundTerm -> Bool .
506 ceq convAxiom?(GT) = true if
        getTerm(metaXapply(upModule('PROOF-SEARCH,false),GT,
508     'conv, none, 0, unbounded, 0)) ==
         'emptyConf.Configuration .
510
    *** egen regel for conv-aksiomet:
512 vars FS FS' : FmlSet .
    vars f f' : Fml .
514 crl [conv] : FS, B(N) f, C(N) f' ==> FS'
              => ------------------------
516              emptyConf
        if  kFree?(N, f) /\ kFree?(N, f') /\
518       not performProofSearch(emptyFmlSet ==> not f, not
              f') .
```

```
520     *** Ikke aksiom: en tom side og kun atomer på den andre
            siden:
        op nonaxiom? : Result -> Bool .
522     ceq nonaxiom?({'_==>_[G,'emptyFmlSet.FmlSet] |
           R:RuleSort})
            = true if atomic?(G) .
524     ceq nonaxiom?({'_==>_['emptyFmlSet.FmlSet,G] |
           R:RuleSort})
            = true if atomic?(G) .
526     ceq nonaxiom?({'_==>_['p[G],'p[GT]] | R:RuleSort})
            = true if  not G == GT .
528     eq nonaxiom?(R) = false [owise] .

530     *** Sjekk om en term har sorten fmlSet:
        op fmlSet? : GroundTerm -> Bool .
532     eq fmlSet?(GT) = leastSort(upModule('FORMULA, false),
           GT)
          == 'FmlSet .
534
        *** inneholder metarepresentert term kun atomer?
536     op atomic? : GroundTerm -> Bool .
        eq atomic?(G) = getTerm(metaReduce(['FORMULA],
538             'atomic?[G])) == 'true.Bool .

540     *** er metarepresentert term T fri for modalitet K?
        op metakFree? : GroundTerm GroundTerm -> Bool .
542     eq metakFree?(K:GroundTerm, T:GroundTerm) =
            getTerm(metaReduce(['FORMULA],
544             'kFree?[K:GroundTerm, T:GroundTerm])) ==
                   'true.Bool .
    endm
```

Bibliografi

[1] Tim Berners-Lee, James Hendler og Ora Lassila. The semantic web. *Scientific American*, Mai 2001.

[2] Stefano Ceri, Georg Gottlob og Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, mars 1989.

[3] Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer og Hans Tompits. Combining answer set programming with description logics for the semantic web. I Didier Dubois, Christopher A. Welty og Mary-Anne Williams, redaktører, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*, side 141–151. AAAI Press, 2004.

[4] Iselin Engan, Tore Langholm, Espen H. Lian og Arild Waaler. Default reasoning with preference within only knowing logic. I Chitta Baral, Gianluigi Greco, Nicola Leone og Giorgio Terracina, redaktører, *LPNMR*, bind 3662 av *Lecture Notes in Computer Science*, side 304–316. Springer, 2005.

[5] H. Gallaire, J. Minker og J.Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2), 1984.

[6] Bjarne Holen, Einar Broch Johnsen og Arild Waaler. Representing strategies for the connection calculus in rewriting logic. I *Proceedings of International Workshop First Order Theorem Proving*, Koblenz, Germany, 2005.

[7] Johan W. Klüwer. Doxastic positions for database query. Upublisert, Oktober 2005.

[8] Hector J. Levesque. Foundations of a functional approach to knowledge representation. *Artificial Intelligence*, 23(2):155–212, 1984.

[9] Hector J. Levesque og Gerhard Lakemeyer. *The logic of knowledge bases*. Cambridge, Mass, 2000.

[10] Espen Lian. Only knowing with confidence levels: Reductions and complexity. Hovedfagsoppgave, Universitetet i Oslo, 2004.

[11] Espen Lian. Implementasjon av en algoritme for normalisering av æ-formler. Ikke publisert, 2005.

[12] Jack Minker. Perspectives in deductive databases. *Journal of Logic Programming*, 5:33–60, 1988.

[13] Raymond Reiter. On closed world data bases. I *Logic and Databases*, side 55–76. Plenum Press, New York, 1978.

[14] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, bind 1. Computer Science Press, 1989.

[15] Alberto Verdejo og Narciso Martí-Oliet. Implementing CCS in Maude 2. I Fabio Gadducci og Ugo Montanari, redaktører, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, bind 71 av *Electronic Notes in Theoretical Computer Science*, side 239–257. Elsevier, 2002.

[16] Arild Waaler. Consistency proofs for systems of multi-agent only knowing. I Renate Schmidt, Ian Pratt-Hartmann, Mark Reynolds og Heinrich Wansing, redaktører, *Advances in Modal Logic*, bind 5, side 347–366. King's College Publications, 2005.

[17] Arild Waaler, Johan W. Klüver, Tore Langholm og Espen H. Lian. Only knowing with degrees of confidence. To appear in: Journal of Applied Logic, 2006.