Avskjærings- og søkestrategier i system for automatisk bevisføring.

Hovedoppgave i informatikk.

Dag Diesen
Institutt for informatikk
Universitetet i Oslo

25. januar, 1988

Innhold

Tabell-liste

Kapittel 1

Forord med introduksjon

Denne rapporten er en hovedoppgave til cand. scient. graden i informatikk ved Universitetet i Oslo. Min veileder har vært forsker Rolf Nossum ved Norsk Regnesentral. Intern kontaktperson (veileder) ved Institutt for informatikk har vært førsteamanuensis Herman Ruge Jervell.

Oppgaven har bestått i å studere avskjærings- og søkestrategier i et system for automatisk bevisføring. Systemet er basert på en metode utviklet av Wolfgang Bibel, kalt forbindelsesmetoden ("connection method") i denne rapporten. Arbeidet beskrevet i denne rapporten er hovedsakelig et teoretisk arbeide.

1.1 Målsetting og hovedtrekk ved utforming av oppgaven

Målsettingen med oppgaven har dels vært å studere avskjæringsstrategier og søkestrategier for forbindelsesmetoden, med sikte på om mulig å finne forbedringer.

Videre har jeg også sett det som viktig i noen grad å gi en oversikt over deler av teorien for automatisk bevisføring for utsagnslogikk og predikatlogikk.

Med hensyn på avskjæringsstrategier har det vært mulig å gi noen grad av oversikt over hva som er gjort og også å finne visse forbedringer. For søkestrategier er det ikke gjort så veldig mye arbeid. Også i denne rapporten blir søkestrategier behandlet svært kort. Dette har sin årsak i at arbeidet med avskjæringsstrategier, og det å gi en mere generell oversikt over deler av teorien for automatisk bevisføring tok det meste av tida.

Når det gjelder å gi en oversikt over deler av teorien for automatisk bevisføring, behandler jeg i rapporten unifikasjon, normalisering, og noe mere inngående forbindelsesmetoden.

Resolusjon kunne vært behandlet i denne rapporten som en kontrast til forbindelsesmetoden, men tida strakk ikke til for å få metoden beskrevet i rapporten.

Generelt har jeg i min framstilling lagt til grunn at vi søker å validere et utsagn. Dette er i samsvar med den måten forbindelsesmetoden er beskrevet i litteraturen. Dette i motsetning til den vanlige beskrivelsen av resolusjon.

Der er utgansgspunktet at vi forsøker å finne et utsagn som er utilfredstillbart.

De to måter å beskrive automatisk bevisføring er likeverdige. Gitt et predikatlogisk utsagn på lukket og rettet form som er gyldig, da vil det negerte utsagnet være utilfredstillbart.

Normaliseringsformer finnes enten man ønsker å validere et utsagn eller man ønsker å vise at utsagnet er utilfredstillbart (konjunktiv eller disjunktiv normalform). Se kapittel 3 for en utdypning.

1.2 Oppsummering

Her vil jeg kort gi en oversikt over hva de enkelte kapitlene inneholder.

- Kapittel 2 om unifikasjon gir en beskrivelse av unifikasjon og substitusjon. Robinson's unifikasjonsalgoritme blir presentert, og et par effektive unifikasjonsalgoritmer (med ikke eksponensiell kompleksitet) blir beskrevet. Eksperimentelle resultater som sammenligner effektiviteten av disse tre algoritmene blir gitt.

  Noe teori om idempotente unifikatorer og om unifikasjon av utsagn, som ikke har insatt Skolem funksjoner, blir også behandlet. Videre blir svarekstraksjon tatt opp.

- Kapittel 3 om normalisering er ment å gi en oversikt over muligheter og problemer ved normalisering av utsagn. Normalisering av utsagn uten å sette inn Skolem funksjoner og en normaliseringsprosedyre med lineær kompleksitet blir beskrevet.

- Kapittel 4 om forbindelsesmetoden gir en generell beskrivelse av forbindelsesmetoden for utsagnslogikk og predikatlogikk.

  Nossum's k-kompakthetsmetode blir kort presentert.

- Kapittel 5 om skillemerkestrategien gir en grundig beskrivelse av denne avskjæringsstrategien for utsagnslogikk.

  Skillemerkestrategien blir løftet til predikatlogikk, og teoremer med bevis for at konsistens og kompletthet bevares, blir gitt.

- Kapittel 6 tar for seg andre avskjæringer. Der beskrives reduksjoner, og det diskuteres kort mulighetene for å få en effektivitetsgevinst ved *ikke* å normalisere et utsagn.

  En liten avskjæringsmulighet blir presentert med bevis for at kompletthet bevares.

- Kapittel 7 behandler svært kort noen mulige søkestrategier.

- Kapittel 8 gir forslag til videre arbeid, både med hensyn på teori, og for praktisk implementasjon.

- Litteraturlista gir en oversikt over den litteraturer jeg referer til og som har gitt grunnlaget for arbeidet med denne rapporten.

Her passer det å forklare kort hvordan jeg refererer. I en del tilfeller har det vært riktig å referere til kapitler og avsnitt eller direkte til teoremer, definisjoner og lignende. For eksempel vil referansen [Bibel 87, IV.3] vise til kapittel IV, avsnitt 3 i [Bibel 87]. I andre tilfelle har jeg tilsvarende vist til en bestemt side, som i for eksempel [Nossum 84, s 15].

1.3 Selve hovedoppgaven

Denne hovedoppgaven er en blanding av en oversiktsoppgave og en litt mere forskningspreget del. Derfor vil jeg kort oppsummere hvilke deler av rapporten som (kanskje) representerer noe nytt.

I kapittel 2, avsnitt 2.4 gis eksperimentelle resultater fra implementasjon og kjøring av 3 ulike unifikasjonsalgoritmer. I avsnitt 2.7 kommer jeg med noen tanker om unifikasjon av terrmer som ikke har innsatt skolemfunksjoner.

I kapittel 5, avsnitt 5.3 løftes skillemerkestrategien fra utsagnslogikk til predikatlogikk. En ny prosedyre CP_2^1 for predikatlogikk presenteres og bevis for kompletthet og konsistens gis. I avsnitt 5.2.1 påviser jeg en feil i prosedyren CP_2^0, slik den er gitt i [Bibel 87], for utsagnslogikk.

I kapittel 6, avsnitt 6.3 gis en avskjæring med bevis for at kompletthet bevares.

I kapittel 8, avsnitt 8.7.2 og 8.7.3 definerer jeg k-konfluens og k-begrensethet. Men noen måter å beregne k-konfluens og k-begrensethet har jeg ikke funnet.

Her vil jeg også gjøre oppmerksom på at det i kapittel 2 om unifikasjon blir det vist til en implementasjon av Robinson's unifikasjonsalgoritme gjengitt i vedlegg B. Dette programmet er laget av Marit Holden og inngår selvsagt ikke som noe arbeid med min hovedoppgave.

Til slutt i denne introduksjonen, vil jeg rette en takk til alle som har hjulpet og oppmuntret meg mens jeg arbeidet med denne hovedoppgaven.

Marit Holden har gitt meg tillatelse til å gjengi sin implementasjon av Robinson's unifikasjonsalgoritme.

Min far, Knut E. Diesen har hjulpet meg ved å lese korrektur på denne rapporten.

Herman Ruge Jervell har gitt verdifull faglig hjelp, og i tillegg oppmuntret og støttet meg.

Sverre Spurkland ga meg det rådet å søke veiledning hos Rolf Nossum og han gjorde meg oppmerksom på arbeidet til [Corbin, Bidoit 83].

En spesiell takk til dere alle.

Sist men ikke minst, vil jeg rette en meget varm takk til min veileder Rolf Nossum. Han har gitt meg den beste faglige veiledning og i tillegg støttet og oppmuntret meg på alle måter. Og han har latt meg arbeide så selvstendig med oppgaven som jeg ønsket. Videre har han hjulpet meg slik at jeg kunne delta på Advanced Course in Artificial Intelligence som ble arrangert i Oslo sommeren l987.

Uten hans tålmodige veiledning og støtte, ville dette hovedfagsarbeidet vært et vesentlig dårligere arbeid.

Kapittel 2

Unifikasjon

2.1 Innledning

2.1.1 Formål

Hensikten med dette kapitlet er å presentere noen forskjellige unifikasjonsalgoritmer og gi en vurdering av disse algoritmenes effektivitet. Noen eksperimentelle resultater vil bli presentert. Videre vil noe teori om unifikasjon bli gjennomgått.

Bruk av effektive unifikasjonsalgoritmer vil ikke alene kunne gi svært store bidrag til effektivisering av bevisprosedyrer. Men det bidraget som effektiv unifikasjon kan gi, vil komme som tillegg til den effektivisering vi kan oppnå ved bruk av søke- og avskjæringsstrategier.

En grunn til å skrive om unifikasjon er at det i seg selv er et interessant felt å studere.

2.1.2 Definisjon av termer og atomer

Jeg gir her en rekursiv definisjon av atomer og termer.

Definisjon 2.1 (Term) *1. La f være et funksjonssymbol som betegner en funksjon med lengde n ($n \geq 0$).*

Da er

$$f(t_1, \ldots, t_n)$$

en term dersom $t_1 \ldots t_n$ er termer.

Hvis f har lengde 0 sier vi ofte at

$$f()$$

er en konstant og angir det ved å bruke et konstantsymbol.

2. La v være et variabelsymbol. Da er variabelen

$$v$$

en term.

Ingen andre konstruksjoner er termer.

Definisjon 2.2 (Atom) *La P være et predikatsymbol som betegner et predikat med lengde m ($m \geq 0$). Da er*

$$P(t_1, \ldots, t_m)$$

et atom om $t_1 \ldots t_m$ er termer.

Ingen andre konstruksjoner er atomer.

Videre i dette kapitlet vil jeg bruke *utrykk* som felles betegnelse på termer og atomer. Dette fordi substitusjon og unifikasjon vil skje på samme vis for termer og atomer.

2.1.3 Generelt om substitusjon og unifikasjon

I dette avsnittet kommer de grunnleggende defininisjonene av substitusjon og unifikasjon. Definisjonene er ikke så veldig forskjellige fra de som er brukt i boka [Chang, Lee 73].

Definisjon 2.3 (Substitusjons–komponent) *En substitusjons–komponent er et par på formen*

$$t \rightarrow v$$

der t betegner en term og v betegner en variabel. Videre må t og v være forskjellige.

Definisjon 2.4 (Substitusjon) *En substitusjon er et endelig sett av substitusjonskomponenter. I en substitusjon på formen*

$$\{t_1 \rightarrow v_1, \ldots, t_n \rightarrow v_n\}$$

skal alle variable $v_1 \ldots v_n$ være distinkte. Kall $t_1 \ldots t_n$ term–delen *av en substitusjon og kall $v_1 \ldots v_n$* variabel–delen.

Greske bokstaver brukes til å representere en substitusjon.

Definisjon 2.5 (Den tomme substitusjon) *La ϵ stå for den tomme substitusjon.*

Definisjon 2.6 (Instans) *La U være et utrykk (term eller atom) og la θ være en substitusjon. Da kan vi lage en instans av utrykket U ved å erstatte variabelen v_i i U med termen t_i for hver substitusjonskomponent $t_i \rightarrow v_i$ i θ. Erstatningene skjer simultant. θU betegner U instansiert med θ.*

Eksempel 2.1 *La*

$$\sigma = \{a \rightarrow x, h(z) \rightarrow y\}$$

være en substitusjon og la

$$A = P(x, f(y))$$

være et utrykk. Da får vi instansen

$$\sigma A = P(a, f(h(z)))$$

Definisjon 2.7 (Komposisjon) *La θ og λ være to substitusjoner. Komposisjonen $\lambda \circ \theta$ fåes ved å lage en ny substitusjon på følgende måte:*

1. *Instansier alle termene i term–delen til substitusjonen θ med substitusjonen λ (For hver substitusjonskomponent i θ får vi $\lambda t_i \rightarrow v_i$).*
2. *Fjern de substisjonskomponentene i θ som er slik at $\lambda t_i = v_i$.*
3. *Legg til substiusjonskomponentene i λ til komposisjonen.*
4. *Fjern fra komposisjonen alle substitusjonskomponentene $t_j \rightarrow v_j$ i λ som er slik at v_j forekommer i variabeldelen til θ.*

Eksempel 2.2 *La*

$$\theta = \{h(z) \rightarrow x, y_1 \rightarrow y\}$$

og la

$$\lambda = \{y \rightarrow y_1, a \rightarrow x, b \rightarrow z\}$$

Da vil komposisjonen

$$\lambda \circ \theta$$

være substitusjonen

$$\{h(b) \rightarrow x, y \rightarrow y_1, b \rightarrow z\}$$

Legg merke til i eksemplet ovenfor at subsitusjonskomponentene $y_1 \rightarrow y$ i θ og $a \rightarrow x$ i λ ikke er med i komposisjonen.

Instansiering av et utrykk U med komposisjonen $\lambda \circ \theta$ er det samme som først å instansiere U med θ, og deretter instansiere θU med λ.

Definisjon 2.8 (Unifikator) *En substitusjon θ er en unifikator for et sett av utrykk hvis og bare hvis alle utrykk i settet instansiert med θ blir like (dvs. $\theta U_1 = \theta U_2 = \cdots = \theta U_l$ for settet $\{U_1, U_2, \ldots, U_l\}$).*

Definisjon 2.9 (En mest generell unifikator (mgu)) *La σ være en unifikator for et sett av utrykk. Da er σ en mest generell unifikator hvis og bare hvis det for hver unifikator θ for settet av utrykk finnes en substitusjon λ slik at $\theta = \lambda \circ \sigma$.*

Hvis et sett av utrykk har en unifikator, vil det også finnes en mest generell unifikator for settet.

2.2 Representasjon av termene som en trestruktur

Robinson's unifikasjonsalgoritme bygger på at termene i de atomene som skal unifiseres er representert som en trestruktur. Representasjon av termer som et tre følger naturlig av den rekursive definisjonen for termer. Men bruk av en trestruktur leder til en unifikasjonsalgoritme med eksponensiell kompleksitet.

2.2.1 Robinson's unifikasjonsalgoritme

Robinson's unifikasjonsalgoritme finnes i mange versjoner. Jeg gjengir algoritmen i en rekursiv versjon [Corbin, Bidoit 83].

Algoritme 2.1 (Unifiser)
Inn: *To utrykk (term1,term2).*
Retur: *Mest generelle unfikator og en boolsk variabel (mgu,b).*

```
begin
    if et av utrykkene er en variabel
    then
        x := den termen som er en variabel;
        t := den andre termen;
        if x = t
        then
            b := true;
            mgu := ϵ
        else
            if Forekommer(x,t)
            then
                b := false
            else
                b := true;
                mgu := {t → x}
            fi
        fi
    else
        f := (predikat-) funksjonsnavnet til term1;
        g := (predikat-) funksjonsnavnet til term2
        if f ≠ g
        then
            b := false
        else
            k := 0;
            b := true;
            mgu := ϵ ;
            m := antall subtermer i term1 og term2;
            while k < m and b = true
            do
                k := k+1;
                subt1 := k'te subtermen til (predikatet) funksjonen f_m ;
                subt2 := k'te subtermen til (predikatet) funksjonen g_m ;
                (mgu1,b) := Unifiser(subt1,subt2)
                if b = true
                then
                    mgu := mgu1 ∘ mgu;
                fi
            od
```

fi

fi

end

Algoritmen *Forekommer(x,t)* sjekker om variabelen x forekommer i t.

Inn: En variabel og en term (x,t)
Retur: En boolsk verdi som er sann hvis x forekommer i t og falsk ellers.

Selve algoritmen Forekommer(x,t) gjengis ikke her. Implementasjon i LISP av Unifikasjon (og Forekommer) finnes i vedlegg B.

2.3 Representasjon av termene som en endelig rettet asyklisk graf

I dette avsnittet vil jeg presentere to unifikasjonsalgoritmer som begge tar utgangspunkt i at termene er representert som en endelig rettet asyklisk graf (ERA–graf).

I en trestruktur vil en og samme variabel kunne forekomme mange steder. Substitusjon av en term for en variabel vil derfor måtte gjøres mange steder i trestrukturen.

I en rettet asyklisk graf vil derimot alle variable være distinkte. Men hver variabel vil kunne ha flere fedre. Det er heller ikke noe i veien for at en funksjonsterm kan ha flere fedre. Eksempel på det er den grafen vi får etter å har kjørt unifikasjonsalgoritmen til [Corbin, Bidoit 83]. Men i den initielle grafen vil kun variable ha flere fedre.

De to algoritmene som gjennomgås er Robinson's unifikasjonsalgoritme modifisert til å brukes på en ERA–graf [Corbin, Bidoit 83] og en algoritme som lager ekvivalensklasser av unifiserbare termer [Paterson, Wegman 78]. Begge algoritmene har en kompleksitet som er sub–eksponensiell.

2.3.1 Den modifiserte Robinson's algoritme

Denne algoritmen er Robinson's algoritme modifisert slik at den opererer på en endelig rettet asyklisk graf der alle variable er distinkte [Corbin, Bidoit 83].

Algoritme 2.2 (Den modifiserte Robinson's algoritme)

Inn:	*Et par distinkte noder* $(v1, v2)$ *som representerer de to termene som skal unifiseres. Hver term må være representert som en endelig rettet asyklisk graf*
Ut:	*Mest generelle unifikator og en boolsk variabel* (σ, b). *Hvis* $b =$ ***true*** *så er* σ *en mest generell unifikator for de to termene.* σ *er ikke nødvendigvis gitt eksplisitt.*
Sidevirkning:	*Ved en vellykket unifikasjon vil de to termene*

bli instansiert med den mest generelle unifikator. Alle like termer i ERA-grafen vil være representert med en node.

```
Unifiser(v1,v2)
begin
    if en av nodene representerer en variabel
    then v := den noden som representerer en variabel;
         w := den andre noden
         if Forekommer(v,w)
         then
              b := false;
         else
              b := true;
              σ := {Term(v) → Term(w)} ;
              Erstatt(v,w)
         fi
    else
         f := Navn(v1);
         g := Navn(v2);
         if f ≠ g
         then
              b := false
         else
              k := 0;
              b := true;
              m := antall subtermer i Term(v1) og Term(v2);
              σ := ε;
              while k < m and b = true
              do
                   k := k + 1;
                   w1 := Subnode(v1,k);
                   w2 := Subnode(v2,k);
                   if w1 ≠ w2
                   then
                        (σ1,b) := Unifiser(w1,w2);
                        if b = true
                        then
                             σ := σ1 ∘ σ
                        fi
                   fi
              od;
              if b = true
              then
                   Erstatt (v1,v2)
              fi
```

fi

fi

end

Her følger en beskrivelse av de prosedyrer som blir kalt opp av Den modifiserte Robinson's algoritme.

Forekommer(v,w)
Sjekker om noden v forekommer som subnode til w.

Inn:	To noder (v,w)
Retur:	En boolsk verdi som er sann om v forekommer som subnode til w.

Term(t)

Inn:	En node t
Retur:	Termen som noden t representerer på passende form.

Erstatt(v,w)
Erstatter noden v med w i grafen. Oppdaterer pekerene.

Inn:	Nodene v og w
Retur:	—
Sidevirkning	Node v forsvinner fra grafen.

Navn(n)
Gir navnet til en funksjon eller et predikat.

Inn:	Node n
Retur:	Funksjon- eller predikatnavnet til node n

Subnode(v1,k)

Inn:	Node v1 og et naturlig tall k ($1 \leq k \leq f_n$)
Retur	k'te subnode til noden v1

Den modifiserte Robinson's unifikasjonsalgoritme vil kunne returnere en unifikator der termene er på listeform (som en trestruktur). Dette kan oppnåes ved å ordne substitusjonskomponentene på samme måte som i Paterson's og Wegman's unifikasjonsalgoritme. En slik ordning er ikke implementert i forbindelse med denne oppgaven. Se ellers vedlegg C for en implementasjon i LISP av Den modifiserte Robinson's unifikasjonsalgoritme. Unifikatorene er i dette programmet representert som variable med pekere til de substituerende termene i ERA-grafen.

2.3.2 Paterson's og Wegman's unifikasjonsalgoritme

De to termene som skal unifiseres er representert som en endelig rettet asyklisk graf der alle variable er forskjellige. Nodene (som representerer termer og subtermer) deles opp i ekvivalensklasser med utgangspunkt i følgende ekvivalensrelasjon: [Paterson, Wegman 78, side 160]

Definisjon 2.10 (Ekvivalente noder)

1. *To funksjonsnoder er ekvivalente dersom deres korresponderende sønner er parvis ekvivalente.*

2. *Ingen ekvivalensklasser kan inneholde to noder med forskjellige funksjonssymboler.*

3. *Ekvivalensklassene må kunne ordnes etter samme partielle ordning som nodene i grafen.*

$u \equiv v$ står for at node u er ekvivalent med node v.

De to første betingelsene i definisjonen 2.10 skal sikre at hver *ekvivalensklasse* kan *reduseres* til *en* node, men slik at alle funksjoner fra den opprinnelige grafen kan finnes i den reduserte grafen (med samme funksjonssymbol og antall argumenter).

Den tredje betingelsen skal sikre at den reduserte grafen er asyklisk.

Sammenhengen mellom unifikasjon og ekvivalensrelasjonen i definisjon 2.10 gies ved følgende hjelpesetning [Paterson, Wegman 78, side 160]:

Hjelpesetning 2.1 (Unifikasjon og ekvivalente noder) *Termene representert ved node u og v kan unifiseres hvis og bare hvis har $u \equiv v$ (som definert i 2.10). Hvis termene kan unifiseres har vi også en unik minimal ekvivalensrelasjon (som definert i 2.10).*

Et av problemene er å teste at siste betingelsen i ekvivalensrelasjonen 2.10 holder. Dette løses ved å definere en rotklasse som en ekvivalensklasse der nodene ikke har fedre.

Hjelpesetning 2.2 *En ikke-tom ekvivalensrelasjon, som oppfyller samme partielle ordning som nodene i grafen har en rotklasse.*

Med denne siste hjelpesetningen [Paterson, Wegman 78, side 161] er det teoretiske grunnlaget gitt for Paterson's og Wegman's unifikasjonsalgoritme.

Følgende algoritme vil i prinsippet avgjøre om to termer kan unifiseres:

Algoritme 2.3 (Ekvivalens(u,v))

```
begin
    sett u ≡ v;
    while det finnes en rotklasse R
    do
        if R inneholder noder med forskjellige funksjonssymboler
        then
            u og v kan ikke unifiseres. Stopp
        else
            if R inneholder en funksjonsnode
            then
                r1 := en funksjonsnode fra R
            else
                r1 := en node fra R
```

```
            fi;
            k := antall noder i R;
            for i := 2 step 1 until k
            do
                if r_i er en variabelnode
                then
                        lag substitusjonskomponenten r_1 → r_i
                fi
                if r_i er en funksjonsnode
                then
                        l := antall sønner til r_1;
                        for j := 1 step 1 until l
                        do
                            Ekvivalens(Sønn(r_i),Sønn(r_1))
                        od
                fi
            od
            fjern nodene i R og deres utgående kanter
        fi
    od;
    if noen noder ikke er fjernet
    then
          u og v kan ikke unifiseres. Stopp
    fi
end
```

Legg merke til at nodene i en ferdigbehandlet rotklasse fjernes.

Algoritme 2.4 (Ekvivalens-kant(u,v))

```
begin
    Lag-kant(u,v)
    while eksisterer en funksjonsnode r
    do
        Rotklasse(r)
    od;
    while eksisterer en variabelnode r
    do
        Rotklasse(r)
    od
    returner med en mest generell unifikator
end
```

Rotklasse(r) behandler en ekvivalensklasse. For hver node den behandler undersøker den om noden har fedre. I så fall vil *Rotklasse(r)* kalle seg selv for rekursivt å behandle ekvivalensklassene til farsnodene. Når en ekvivalensklasse er ferdigbehandlet, blir nodene i den fjernet. Dermed vil ingen ekvivalensklasser bli behandlet før de er blitt rotklasser.

Algoritme 2.5 (Rotklasse(r))

```
begin
    if Peker(r) er definert
    then
        Avslutt("Grafen er syklisk")
    else
        Peker(r) := r
    fi;
    lag en ny stakk med operasjonene Push(node) og Pop;
    Push(r);
    while stakken er ikke-tom
    do
        s := Pop;
        if Navn(r) ≠ Navn(s)
        then
            Avslutt("Forskjellige funkjsoner i en ekvivalensklasse")
        fi;
        while s har noen fedre t
        do
            Rotklasse(t)
        od;
        while noen kant (s,t) eksisterer
        do
            if Peker(t) er udefinert
            then
                Peker(t) := r
            fi;
            if Peker(t) ≠ r
            then
                Avslutt("Grafen er syklisk")
            fi;
            fjern kanten (s,t);
            Push(t)
        od;
        if s ≠ r
        then
            if s er en variabelnode
            then
                lag substitusjonskomponenten r → s
            fi;
            if s er en funksjonsnode (med minst en sønn)
            then
                l := antall sønner;
                for j := 1 step 1 until l
                do
                    Sett-kant(Sønn(r,j),Sønn(s,j))
                od;
```

```
            fi;
            Fjern s og kantene ut i fra s
        fi
    od;
    Fjern noden r og kantene ut fra r
end
```

Paterson's og Wegman's unifikasjonsalgoritme returnerer et *sett* av substitusjonskomponenter som ikke uten videre kan betraktes som en idempotent mest generell unifikator.

En idempotent substitusjon må være slik at ingen variabel i variabeldelen av en substitusjon skal forekomme i noen term i termdelen av en substitusjon.

Men substitusjonskomponentene i en mest generell unifikator kan ordnes slik at substitusjonen entydig representerer en idempotent substitusjon. Hver substitusjonskomponent der dets variabel forekommer i termene til andre substitusjonskomponenter plasseres foran disse. Hvis

$$t_i \rightarrow x_i$$

er slik at x_i forekommer i t_j til

$$t_j \rightarrow x_j$$

så plasseres $t_i \rightarrow x_i$ foran $t_j \rightarrow x_j$.

Med en slik ordning kan en idempotent substitusjon bygges opp ved å starte med siste substitusjonskomponent

$$t_n \rightarrow x_n$$

og instansiere den foregående i denne. Generelt instansieres

$$t_{i-1} \rightarrow x_{i-1}$$

i de etterfølgende substitusjonskomponentene [Stickel 86].

Paterson's og Wegman's unifikasjonsalgoritme returnerer implisitt en idempotent mest generell unifikator fordi den returnerer en sekvens av substitusjonskomponenter som er ordnet som beskrevet ovenfor.

Termene i substitusjonskomponentene kan dermed returneres på listeform (som en trestruktur). En oppbygging av en eksplisitt idempotent mest generell unifikator på listeform vil i noen tilfelle kreve eksponensiell kompleksitet.

For en implementasjon i LISP av Paterson's og Wegman's unifikasjonsalgoritme, se vedlegg D.

2.3.3 Kompleksitet

Paterson's og Wegman's unifikasjonsalgoritme har lineær kompleksitet i kjøretid med hensyn på behandling av noder og kanter [Paterson, Wegman 78].

Fordi utrykk vanligvis blir gitt på listeform, må termene omformes til en ERA-graf. Ved denne omformingen må det for hver forekomst av en variabel

undersøkes om variabelen er en ny distinkt variabel eller om den allerede er satt inn i lista over distinkte variable. Vi får da en kompleksitet på maksimalt $(f + v * v_d)$, der f er antall funksjonssymboler, v er antall variabelsymboler og v_d er antall distinkte variable.

Omformingen av termene i termdelen i den mest generelle unifikatoren σ fra ERA-grafen til listeform har samme kompleksitet for hver term som omformingen av termene fra listeform til ERA-grafen.

Ved unifikasjon skjer det en forenkling av termene som skal unifiseres. Derfor vil kompleksiteten ved omformingen av alle termene i σ ofte være mindre enn kompleksiteten ved omformingen av unifikanden fra listestruktur til en ERA-graf.

Kompleksiteten av Corbin's og Bidoit's unifikasjonsalgoritme er maksimalt av orden $O(p^2)$, der p er antall symboler i de termene som skal unifiseres [Corbin, Bidoit 83].

Kompleksiteten ved omforming av unifikanden fra listeform til ERA-graf vil være den samme som for Paterson's og Wegman's algoritme.

Ved en omforming av termene i termdelen i mgu fra ERA-graf til listeform må substitusjonskomponentene ordnes for å få representert en idempotent mgu. Dermed vil denne omformingen være mere kompleks enn i Paterson's og Wegman's algoritme.

Underliggende operasjoner knyttet til hver node og hver kant vil øke kompleksiteten både i Corbin's og Bidoit's algoritme og i Paterson's og Wegman's algoritme. Men hvordan dette slår ut er avhengig av programmeringsspråket og implementasjonen av algoritmene. Avslutningsvis nøyer jeg meg med å konstatere at ingen av algoritmene får eksponensiell kompleksitet. Derimot må en i praksis regne med at Paterson's og Wegman's algoritme ikke helt har lineær kompleksitet.

2.4 Eksperimentelle resultater

Robinson's unifikasjonsalgoritme, den modifiserte Robinson's algoritme og Paterson's og Wegman's unifikasjonsalgoritme er alle implementert i Common-LISP på DEC 20.

Det er foretatt testkjøringer med implementasjoner av de forskjellige unifikasjonsalgoritmene. Termene som er prøvd ut er på formen [Corbin, Bidoit 83]:

$$VT_n = f(h(x_1, x_1), \ldots, h(x_{n-1}, x_{n-1}), y_2, \ldots, y_n, y_n)$$

$$HT_n = f(x_2, \ldots, x_n, h(y_1, y_1), \ldots, h(y_{n-1}, y_{n-1}), x_n)$$

Alle unifikasjonsalgoritmene er kjørt for $n = 7$ og $n = 10$. Den modifiserte Robinson's algoritme og Paterson's og Wegman's algoritme er kjørt for $n = 14$. For Robinson's unifikasjonsalgoritme var det ikke mulig å kjøre den for $n = 14$. Resultatene finnnes i tabellene 2.1 og 2.2.

Resultatene indikerer at Paterson's og Wegman's algoritme er noe mere effektiv enn den modifiserte Robinson's algoritme til Corbin og Bidoit. Dette gjelder for termer hvor unifikasjonen ville ha vært av eksponensiell kompleksitet ved bruk av Robinson's (opprinnelige) unifikasjonsalgoritme.

TIDSFORBRUK	n=7	n=10	n=14
Robinson's algoritme	2323ms	20535ms	—
Den modifiserte Robinson's algoritme	1155ms	1736ms	2676ms
Paterson's og Wegman's algoritme	1246ms	1790ms	2510ms

Tabell 2.1: Tidsforbruk ved unifikasjon av V_n og H_n for ulike n

PLASSFORBRUK	n=7	n=10	n=14
Robinson's algoritme	2485 ord	11239 ord	—
Den modifiserte Robinson's algoritme	5267 ord	7409 ord	9081 ord
Paterson's og Wegman's algoritme	6041 ord	7283 ord	8931 ord

Tabell 2.2: Plassforbruk ved unifikasjon av V_n og H_n for ulike n

En kan meget tydelig se den betydningen ulike måter å representere en term på har for effektiviteten. En tre-representasjon av termene vil av og til medføre eksponensiell kompleksitet av en unifikasjon, mens en representasjon som ERA-graf gjør det mulig å unngå dette.

2.5 Egenskaper ved substitusjoner og unifikasjoner

I dette avsnittet vil jeg presentere noen viktige teoretiske egenskaper ved substitusjon og unifikasjon. Framstillingen er et referat av en artikkel skrevet av Elmar Eder [Eder 85].

Utgangspunktet for Eder er problemet med å finne en mest generell unifikator for en familie av sett av termer på en mest mulig *fleksibel* måte.

Gitt for eksempel familien av sett av termer:

$$\{\{q, r\}, \{s, t\}\}$$

Den tradisjonelle metoden er å finne en unifikator τ for $\{q, r\}$ og så finne en mest generell unifikator ρ for $\{\tau s, \tau t\}$. Komposisjonen $\sigma = \rho \circ \tau$ er da den mest generelle unifikator for familien av sett. Denne måten er ikke så veldig fleksibel fordi den mest generelle unifikatoren for familien av sett må bygges opp sekvensielt.

Eder gir bevis for en noe mere fleksibel metode. Hvis de mest generelle unifikatorene er τ og μ for settene av termer $\{q, r\}$ og $\{s, t\}$, så kan eventuelt den mest generelle unifikator σ for familien av sett

$$\{\{q, r\}, \{s, t\}\}$$

fåes fra τ og μ.

Videre introduserer Eder svak unifikasjon. Hvis et sett av termer har en mest generell unifikator, kan en ny mest generell unifikator finnes for en variant av settet av termer ved enkel navneendring av variablene i den opprinnelig generelle unifikator. (Med varianter mener jeg termer som har fått nye variabelnavn der variabelnavnene ikke er brukt tidligere.) Svak unifikasjon gir grunnlaget for å skjønne hvordan og når en slik navneendring kan skje.

De viktige resultatene til Eder *forutsetter* at de mest generelle unifikatorer er idempotente substitusjoner. En idemptotent substitusjon er en substitusjon der $\sigma = \sigma \circ \sigma$. Alle (av de alminnelige) kjente unifikasjonsalgoritmene vil gi en idempotent substitusjon som en mest generell unifikator.

I Eder's framstilling er det også en interessant teori med hensyn på ordning og ekvivalensklasser av substitusjoner.

2.5.1 Ordning av substitusjoner

Eder definerer en ordningsrelasjon 2.11 og en ekvivalensrelasjon 2.12.

Definisjon 2.11 (Mere generell ($\leq$)) *Gitt to substitusjoner σ og τ. σ er mere generell enn τ, $\sigma \leq \tau$, hvis det eksisterer en substitusjon ρ slik at $\rho \circ \sigma = \tau$.*

Definisjon 2.12 (Ekvivalent ($\sim$)) *Hvis substitusjonene σ og τ oppfyller ordningen $\sigma \leq \tau$ og $\tau \leq \sigma$, så er σ og τ ekvivalente, $\sigma \sim \tau$.*

Mere generell relasjonen gir ikke noen partiell ordning av et sett av substitusjoner. Dette fordi et sett av substitusjoner med en øvre grense τ kan ha mange eller ingen elementer som gir en *minste* øvre grense.

Derimot vil settet av ekvivalensklasser med hensyn på ekvivalensrelasjonen 2.12 bli partielt ordnet av mere generell relasjonen 2.11.

Komposisjon er ikke kompatibel med ekvivalensrelasjonen. Det vil si at selv om vi har $\sigma \sim \sigma'$ trenger vi ikke ha $\tau \circ \sigma \sim \tau \circ \sigma'$.

Definisjon 2.13 (Strengt mere generell ($<$)) *Gitt to substitusjoner σ og τ. Da er σ strengt mere generell enn τ, $\sigma < \tau$, hvis og bare hvis σ er mere generell enn τ, men ikke ekvivalent med τ.*

Eder gir bevis for følgende setning:

Teorem 2.1 (Minimalt element) *En strengt mere generell relasjon er en relasjon der hvert ikk-tomt del-sett Σ av settet av substitusjoner har et minimalt element.*

En relasjon med egenskapen at hvert ikke-tomt del-sett Σ av et sett har et minimalt element kalles en Noetherisk relasjon.

2.5.2 Idempotente substitusjoner

Definisjon 2.14 (Idempotente substitusjoner) *En substitusjon σ er idempotent hvis og bare hvis $\sigma \circ \sigma = \sigma$.*

En idempotent substitusjon er slik at ingen variabel fra variabeldelen forekommer i noen av termene i termdelen av en substitusjon. Se avsnitt 2.3.2 for en nærmere forklaring.

Om vi har σ og τ som idempotente substiusjoner, så er ikke nødvendigvis komposisjonen $\sigma \circ \tau$ en idempotent substitusjon. Komposisjonen av to idempotente substitusjoner trenger heller ikke være ekvivalent med en idempotent substitusjon.

Om vi begrenser oss til idempotente substitusjoner, så er komposisjonen fortsatt ikke kompatibel med ekvivalensrelasjonen.

Definisjon 2.15 (Ekvivalensklasser av idempotente substitusjoner) *Bruk følgende notasjon og definer:*

I = *Settet av idempotente substitusjoner.*
$I^{\sim}$ = *Settet av ekvivalensklasser begrenset til idempotente substitusjoner.*
∞ = *Et vilkårlig objekt slik at* $\infty \notin I^{\sim}$

Definer $I^{\sim}_{\infty} = I^{\sim} \cup \{\infty\}$. *Utvid den partielle ordningen* $\leq$ *for* $I^{\sim}$ *til å gjelde* $I^{\sim}_{\infty}$ *ved å la* ∞ *være største element i* $I^{\sim}_{\infty}$.

Hvis $\sigma \in I$ *så betegner* $\sigma^{\sim}$ *den ekvivalensklassen som har* σ *som et element.*

Teorem 2.2 (Noetherisk) *Strengt mere generell relasjonen gir en irrefleksiv partiell ordning på settet* $I^{\sim}_{\infty}$. *Videre vil hvert ikke-tomt del-sett av* $I^{\sim}_{\infty}$ *ha et minimalt element (Noetherisk). For hver* $\Phi \in I^{\sim}$ *så er settet*

$$\{\Phi' \in I^{\sim}_{\infty} \mid \Phi' \leq \Phi\}$$

endelig.

For idempotente substitusjoner gjelder at hvert sett av idempotente substitusjoner som har en øvre grense også har en minste øvre grense (Se teorem 2.4).

2.5.3 Unifikasjon av en familie av sett

Definisjon 2.16 (Mgu for en familie av sett av termer) *La* $\mathcal{M}$ *være en familie av sett av termer. La* σ *være en substitusjon.*

1. σ *er en unifikator for* $\mathcal{M}$ *hvis og bare hvis* σ *er en unifikator for alle sett* $M \in \mathcal{M}$.

2. σ *er en mest generell unifikator for* $\mathcal{M}$ *hvis og bare hvis* σ *er nedre grense for settet av unifikatorer for* $\mathcal{M}$.

En mest generell unifikator for et sett av termer M kan tilsvarende defineres som den σ som er nedre grense for settet av unifikatorer for M (Sammenlign med definisjonen 2.9).

La $V(termdel(\sigma))$ være settet av variabelforekomster til termdelen av en substitusjon σ.

La $V(\mathcal{M})$ være settet av variabelforekomster i $\mathcal{M}$.

En familie av sett, som har en unifikator, har også en mest generell unifikator. En mest generell unifikator kan bestemmes ved en av de kjente unifikasjonsalgoritmene. De (alminnelige) kjente unifikasjonsalgoritmene gir en idempotent substitusjon som en mest generell unifikator. Videre vil $V(termdel(mgu))$ og variabeldelen til mgu'en være delmengder av $V(\mathcal{M})$.

Den mest kjente metoden for å finne en mgu for en familie av sett av termer er ved suksessiv komposisjon. Mere presist kan det utrykkes i følgende hjelpesetning([Eder 85, side 41]):

Hjelpesetning 2.3 (Komposisjon) *La $\mathcal{M}$ og $\mathcal{N}$ være vilkårlige familier av sett av termer og anta at σ er en mest generell unifikator for $\mathcal{M}$. Da har vi*

1. *$\mathcal{M} \cup \mathcal{N}$ kan unifiseres hvis og bare hvis $\sigma\mathcal{N}$ kan unifiseres.*
2. *Om τ er en mest generell unifikator for $\sigma\mathcal{N}$, så er $\tau \circ \sigma$ en mest generell unifikator for $\mathcal{M} \cup \mathcal{N}$.*

I avsnitt 2.6 gir jeg et induksjonsbevis for at den suksessive komposisjonen kan skje i vilkårlig rekkefølge forutsatt at unifikasjonsalgoritmen returnerer med en idempotent mest generell unfikator. Dette er i samsvar med Eder's mere algebraiske resultater.

I sin videre behandling velger Eder å arbeide med *ekvivalensklasser* av idempotente substitusjoner. Det finnes ingen analog setning til hjelpesetning 2.3 for ekvivalensklasser siden komposisjon ikke er kompatibel med ekvivalensklasser.

Definisjon 2.17 (Idempotent mgu) *La $\mathcal{M}$ være en endelig familie av endelige sett av termer. Elementet $mgu(\mathcal{M})$ i settet $I_\infty^\sim$ blir definert slik:*

1. *Hvis $\mathcal{M}$ kan unifiseres, da er $mgu(\mathcal{M})$ settet av alle de idempotente mest generelle unifikatorer av $\mathcal{M}$.*
2. *Hvis $\mathcal{M}$ ikke kan unifiseres, så er $mgu(\mathcal{M}) = \infty$.*

Legg merke til at hvis en mest generell unifikator finnes for $\mathcal{M}$, så finnes også en idempotent mgu for $\mathcal{M}$. $mgu(\mathcal{M})$ kan derfor ikke være tom.

Teorem 2.3 *La $\mathcal{M}$ og $\mathcal{N}$ være en endelig familie av endelige sett. Da har vi*

$$mgu(\mathcal{M} \cup \mathcal{N}) = sup\{mgu(\mathcal{M}), mgu(\mathcal{N})\}$$

$sup\{Sett\}$ står for minste øvre grense for settet.

Teorem 2.3 åpner opp for en litt annen måte å finne en mest generell unifikator for en familie av termer enn suksessiv komposisjon.

La τ være en idempotent substitusjon. Bygg opp $\mathcal{K}$ som en familie av uordnede par av termer. Hvert par skal bestå av variabelen og termen i en substitusjonskomponent til τ, og $\mathcal{K}$ skal bygges opp fra samtlige substitusjonskomponenter i τ.

Generelt hvis

$$\tau = \{t_1 \rightarrow v_1, \ldots, t_n \rightarrow v_n\}$$

så skal vi få

$$\mathcal{K} = \{\{v_1, t_1\}, \ldots, \{v_n, t_n\}\}$$

Hver idempotent substitusjon τ er en mest generell unifikator for familien av par av termer bygd opp fra alle substitusjonskomponentene i τ. Dette kan nå brukes til å bestemme minste øvre grense for to idempotente substitusjoner σ og τ.

Anta at $\mathcal{K}$ er bygd opp fra substitusjonskomponentene i τ som beskrevet ovenfor. Da finnes en minste øvre grense ρ av σ og τ hvis og bare hvis det finnes en mest generell unifikator for $\mathcal{K}$ instansiert med σ. For å finne ρ brukes en av de eksisterende unifikasjonsalgoritmene.

Hvis σ og τ er mgu for henholdsvis $\mathcal{M}$ og $\mathcal{N}$ kan tid spares fordi vi slipper å bryte opp $\mathcal{N}$ ved unifisering av $\sigma\mathcal{N}$. En nødvendig forutsetning for å spare tid er at unifikasjonen av $\mathcal{N}$ og varianter av $\mathcal{N}$ ikke må gjøres hver gang en mgu trenges. En mest generell unifikator av en variant av $\mathcal{N}$ fåes ved å gi nye variabelnavn for noen variable i termdelen i en av de mest generelle unifikatorene for $\mathcal{N}$.

Jeg avslutter dette avsnittet om unifikasjon av familier av sett av termer med å gjengi to intressante teoremer vist av Eder.

Teorem 2.4 (Gitter) *$I^{\sim}_{\infty}$ er et komplett gitter (lattice) med et minste element $\varepsilon^{\sim}$ og et største element ∞.*

Teorem 2.5 *La $\mathcal{M}$ være en familie av sett av termer der familien har en unifikator. Da finnes en idempotent mest generell unifikator for M, σ og variabeldelen til σ og $V(termdel(\sigma)$ vil være delmengder av $V(\mathcal{M})$.*

Her settes ingen krav om at $\mathcal{M}$ skal være en *endelig* familie av *endelige* sett.

2.5.4 Svak unifikasjon

Definisjon 2.18 (Svak unifikasjon) *La $\mathcal{M}$ være en familie av (ordnede) par av termer. Paret (σ, σ') av substitusjoner er en svak unifikator for $\mathcal{M}$ hvis og bare hvis $\sigma t = \sigma' t$ holder for alle $(t, t') \in \mathcal{M}$.*

Om det eksisterer en svak unifikator for en familie av par av termer, så sier vi at settet er svakt unifiserbart.

Mange kopier av en del av en formel vil ofte trenges for å utlede en komplementær formel. Mange kopier av en term t vil da kansje måtte unifiseres med termen t'. Da kan tid spares ved å unifisere svakt paret (t, t') fordi unifikasjon av et par av varianter av t og t' vil begrense seg til arbeidet med å gi nye navn til noen av variablene.

Dette holder hvis variantene til termene t og t' ikke har felles variable. Hvis derimot termene har felles variable, så er det nødvendig med ekstra unifikasjon av par av variabler.

2.6 Rekkefølgen ved unifikasjon av en familie av sett

I noen bevisprosedyrer (som f. eks. resolusjon) skal det finnes en familie av sett som validerer formelen. Dersom en unifikasjonsalgoritme brukes, så skal det testes på at det finnes en mest generell unifikator som unifiserer alle atomene i hvert sett av literaler i den familien av sett som (eventuelt) validerer formelen.

Unifikasjon av settene i familien kan skje i *vilkårlig* rekkefølge dersom en mest generell unifikator finnes for familien av sett. Forutsetningen er at den unifikasjonsalgoritmen som brukes, returnerer en *idempotent* generell unifikator. Eder gir et algebraisk bevis for dette [Eder 85].

En algoritme vil bli gitt som returnerer en mest generell unifikator for en familie av sett av atomer, om en mest generell unifikator eksisterer.

2.6.1 Finne en mest generell unifikator for en familie av sett ved komposisjon

Her vil jeg definere en algoritme som finner en mest generell unifikator av en familie av sett av atomer (eller termer). Algoritmen lager den mest generelle unifikator ved en komposisjon av den mest generelle unifikator for hvert sett av atomer.

Unifikasjonsalgoritmen, som brukes for å unifisere hvert sett av atomer, forutsettes å gi en idempotent mest generell unifikator. Dette kravet til unifikasjonsalgoritmen er ikke noe problem fordi de kjente unifikasjonsalgoritmene oppfyller dette kravet. Videre i rapporten lar jeg denne forutsetningen gjelde når ikke annet er sagt.

Etterpå gir jeg et bevis på at en mest generell unifikator av familien av sett kan finnes uansett hvilken rekkefølge man velger i komposisjonen av unifikatorene for hvert sett.

Variabelliste til Algoritme 2.6

$\mathcal{S}$:	En familie av sett av atomer (eller termer).
$\mathcal{S}^{ORD}$:	$\mathcal{S}$ ordnet.
S_i og S_{i+1}:	Sett av atomer (eller termer) som er indeksert med i, dvs. $S_i \in \mathcal{S}^{ORD}$ og $S_{i+1} \in \mathcal{S}^{ORD}$
$\mathcal{D}_i$:	Familien av de sett som allererde har fått en mest generell unifikator.
σ_i:	Unifikator for familien av sett $\mathcal{D}_i$.
τ_{i+1}:	Unifikator for $\sigma_i\{S_{i+1}\}$.
n:	Antall sett i $\mathcal{S}$, og i $\mathcal{S}^{ORD}$.

Algoritme 2.6 (Unifisering av en familie av sett)

begin

1. *Initialiser:*
   $\mathcal{D}_0 := \{\emptyset\}$;
   $\sigma_0 := \epsilon$;
   $i := 0$;
2. ***while*** $i < n$
   do
   - *2.1: Finn en mest generell idempotent unifikator* τ_{i+1} *for* $\sigma_i\{S_{i+1}\}$ *(Bruk en av de kjente unifikasjonsalgoritmene).*
   - *2.2: Tilordn følgende nye verdier:*
     $\sigma_{i+1} := \tau_{i+1} \circ \sigma_i$;
     $\mathcal{D}_{i+1} := \mathcal{D}i \cup \{S_{i+1}\}$;
     $i := i + 1$

od

3. *Returner den mest generelle unifikator σ_n som en mest generell unifikator for $\mathcal{S}$.*

end

Teorem 2.6 *La $\mathcal{S} = \{S_1, S_2, \ldots S_n\}$ være en endelig familie av endelig sett av atomer (termer). Anta at en mest generell idempotent unifikator α_i eksisterer for hvert av settene $S_i \in \mathcal{S}$ og at en unifikator finnes for familien av sett $\mathcal{S}$. Anta videre at settene i $\mathcal{S}$ er ordnet, f. eks. $\mathcal{S}^{ORD} = (S_1, \ldots, S_n)$. Bruk Algoritme 2.6 for å finne den mest generelle unifikator for $\mathcal{S}$.*

Da vil en mest generell unifikator kunne finnes for $\mathcal{S}$ ved bruk av Algoritme 2.6 for en vilkårlig permutasjon av $\mathcal{S}^{ORD}$.

Bevis for Teorem 2.6.
I beviset bruker jeg induksjon på antall sett i $\mathcal{S}$ (card $\mathcal{S}$) og hjelpesetning 2.3. Hvis familien $\mathcal{S}$ har en unifikator, så eksisterer en mest generell unifikator for $\mathcal{S}$.
La n være antall sett i $\mathcal{S}$. La $\mathcal{M}$ være $\mathcal{D}_i$, la σ være σ_i, la $\mathcal{N}$ være $\{S_{i+1}\}$ og la τ være τ_{i+1}.
Induksjonsbasis:
$n = 1$: (Trivielt)
$\sigma_0\{S_1\} = \epsilon\{S_1\} = \{S_1\}$.
τ_1 er en mest generell unifikator for $\{S_1\}$ og vi har $\sigma_1 = \tau_1 \circ \sigma_0 = \tau 1 \circ \epsilon = \tau_1$.
Iduksjonsskritt:
$n > 1$:
Anta at Teorem 2.6 holder for alle $k < n$. La $\mathcal{D}_{n-1}$ være $\mathcal{S} \setminus \{S_n\}$ der S_n er siste element av en vilkårlig permutasjon av $\mathcal{S}^{ORD}$. La σ_{n-1} være en mest generell unifikator for $\mathcal{D}_{n-1}$ (som vi antar eksisterer ved induksjonshypotesen). Sett $\mathcal{M} = \mathcal{D}_{n-1}$, $\mathcal{N} = \{S_n\}$ og $\sigma = \sigma_{n-1}$.
Siden vi vet at $\mathcal{S} = \mathcal{D}_{n-1} \cup \{S_n\} = \mathcal{M} \cup \mathcal{N}$ kan unifiseres, så kan $\sigma_{n-1}\{S_n\} = \sigma\mathcal{N}$ unifiseres (hjelpesetning 2.3 pkt. 1).
La $\tau_n = \tau$ være den mest generelle unfikator for $\sigma_{n-1}\{S_n\}$. Da har vi at $\tau_n \circ \sigma_{n-1} = \sigma_n = \tau \circ \sigma$ er den mest generelle unifikator for $\mathcal{D}_{n-1} \cup \{S_n\} = \mathcal{S} = \mathcal{M} \cup \mathcal{N}$ (hjelpesetning 2.3 pkt. 2).
Bevis slutt.

De mest generelle unifikatorer funnet ved bruk av Algoritme 2.6, er ikke nødvendigvis like for forskjellige permutasjoner av $\mathcal{S}^{ORD}$.

Eksempel 2.3 *La*

$$\sigma = \{f(x) \rightarrow y\}$$

og

$$\tau = \{s \rightarrow x\}$$

være to idempotente mest generelle unifikatorer for to sett av termer.

Da er komposisjonene

$$\tau \circ \sigma = \{s \rightarrow x, f(s) \rightarrow y\}$$

og

$$\sigma \circ \tau = \{f(x) \rightarrow y, s \rightarrow x\}$$

forskjellige.

Vi ser at $\tau \circ \sigma$ er en idempotent unifikator, mens $\sigma \circ \tau$ ikke er idempotent.

I avsnitt 2.3.2 er det indikert hvordan en idempotent substitusjon kan bygges opp fra en ikke idempotent substitusjon.

2.7 Hvordan unngå skolemfunksjoner

Det kan være ønskelig å unngå å lage skolemfunksjoner. Dette for å effektivisere bevisprosedyren.

Ved ordning og merking av alle distinkte variable i utsagnet F og ved modifikasjon av unifikasjonsalgoritmen er det mulig å unngå innsetting av skolemfunksjoner i F. Mere detaljert gjøres dette som beskrevet i resten av dette avsnittet.

Merk alle variable som skulle vært erstattet med en skolemfunksjon som negative. De andre variablene merkes som positive. Variablene ordnes slik at det kan bestemmes hvilke variable som er omfattet av kvantoren til en annen variabel. En partiell og en total ordning av variablene presenteres i avsnitt 3.5.1 og 3.5.2.

Unifikasjonsalgoritmene modifiseres ved at de negative variablene behandles som konstanter. De andre modifikasjonene er beskrevet nedenfor i avsnitt 2.7.1 og 2.7.2.

Ved å unngå skolemisering unngår man unifikasjon av skolemtermer med like funksjonssymboler [Nossum 84].

2.7.1 Unifikasjonsalgoritmer med eksplisitt forekomstsjekk

Eksempler på slike algoritmer er Robinson's unifikasjonsalgoritme og Den modifiserte Robinson's unifikasjonsalgoritme.

Legg inn disse endringene i unifikasjonsalgoritmen:

1. Forekomstsjekken sløyfes (blir gjort seinere).

2. Komposisjon av unifikatorer fra subtermene sløyfes (blir gjort seinere).

Kall substitusjonskomponentene oppnådd ved den ufullstendige unifikasjonsalgoritmen en mulig unifikator u. Den vil ha et endelig antall substitusjonskomponenter.

Utfør deretter følgende test for hver substitusjonskomponent i u [Nossum 84]:

Algoritme 2.7 (Har(t, x))
begin
    if *x forekommer i t*
        or *det eksisterer en negativ variabel y som forekommer i t*
            and
                kvantoren til x omfattter y
                or *en variabel z eksisterer der kvantoren til z*

omfatter y slik at Har(u(z), x)

then

*Returner(**true**)*

else

*Returner(**false**)*

fi

end

der

t: Termen som erstatter x
y: En negativ variabel
x, z: Positive variable
$u(z)$: Termen i substitusjonskomponenten $(u(z) \rightarrow z)$ i u

Hvis Har(t, x) er sann for noen substitusjonskomponent i u, så gir ikke u noen unifikator. I motsatt fall er *komposisjonen* av substitusjonskomponentene i u den mest generelle idempotente unifikatoren μ for to ikke skolemiserte termer.

Før substitusjonskomponentene i u testes kan det (kansje) være en fordel å ordne substitusjonskomponentene slik at variablene i variabeldelen er ordnet som beskrevet i avsnitt 3.5.

2.7.2 Unifikasjonsalgoritmer med implisitt forekomstsjekk

Disse unifikasjonsalgoritmene ordner ekvivalensklasser av subtermer korresponderende til strukturen i de to termene som skal unifiseres.

Hvis variabelen i substitusjonskomponenten forekommer i en term, blir dette oppdaget ved at ordningen av ekvivalensklassene leder til en syklisk graf.

Eksempler på slike unifikasjonsalgoritmer er Paterson's og Wegman's unifikasjonsalgoritme og Martelli's og Montenari's unifikasjonsalgoritme.

Disse algoritmene utvides til å omfatte ikke skolemiserte termer ved å sette et krav om at ordningen av ekvivalensklasser også må overenstemme med ordningen av variable som beskrevet i avsnitt 3.5. Se [Bibel 87, IV.8 og IV.9] for mere om unifikasjon av ikke skolemiserte termer i unifikasjonsalgoritmer med implisitt forekomstsjekk.

2.8 Svarekstraksjon

Mange viktige anvendelser av automatisk bevisføring innebærer å lage konstruktive bevis.

Hvis jeg f. eks. ønsker å vise at det eksisterer et program P som oppfyller spesifikasjonene S_P, da vil jeg ikke være helt tilfreds med bare å få svaret: Ja, program P kan lages. Jeg vil også ønske å se et program P. Dette får man til ved først å vise at $\neg S_P$ er utilfredstillbar. La σ være den mest generelle unifikatoren for alle de sett av komplementære forbindelser som gir at $\neg S_P$

er utilfredstillbar. Instansier σ i et utsagn G, slik at G (og dermed $\neg S_P \vee G$) er utilfredstillbart. G forutsettes valgt på en slik måte at $\sigma(G)$ representerer programmet P.

Dette er hovedprinsippet bak det som er kjent som svarekstraksjon. Svarekstraksjon kan sees på som en mekanisme som henter ut annen informasjon som tilegg til svaret: Utsagnet er tilfredstillbart.

Men det er noe en må passe på ved bruk av svarekstraksjon.

1. Utsagnet som σ settes inn i må være lukket og på rettet form.

2. Eventuelle skolemfunksjoner i σ må erstattes med de korresponderende $\exists$-kvantoriserte variablene før innsetting av σ i utsagnet G.

Første punkt representerer ikke noe problem. Andre punkt er heller ikke vanskelig, og det faller bort ved å bruke en unifikasjonsalgoritme som er slik at skolemisering unngåes.

Svarekstraksjon er helt fundemental for mange anvendelser av automatisk bevisføring. Som eksempler på dette nevner jeg logikk–programmeringsspråket PROLOG, bruk av en bevisprosedyre til å hente data fram fra en (relasjons) database og bruk av en bevisprosedyre til programsyntese.

Det er selvsagt mulig å kjøre en bevisprosedyre flere ganger og med ulik søking for å finne flere eksempler på tilfredstillbarheter av samme utsagn. Da får man flere eksempler på en konstruksjon og på den måten f. eks. flere ulike svar fra en database eller flere ulike program som oppfyller samme spesifikasjon ved programsyntese.

I generelle bevisprosedyrer, som for resolusjon og forbindelsesmetoden, kan man aldri vite når alle mulige konstruksjoner er hentet fram. Det har sammenheng med at en bevisprosedyre i predikatlogikk kun er en semialgoritme. Når det ikke er mulig å produsere flere konstruksjoner vil bevisprosedyren ofte kunne divergere.

Svarekstraksjon kan selvsagt brukes på normaliserte utsagn. PROLOG er et eksmpel på at σ settes inn på et normalisert utsagn.

2.9 Avsluttende kommentarer

2.9.1 Effektiv representasjon av en mest generell unifikator

Den idempotente mest generelle unifikator for et sett av literaler bør representeres på en slik måte at den er kompakt. Videre bør den være lett å slette slik at bevisprosedyren kan gjøre et tilbaketog lettest mulig.

En endelig asyklisk graf behøver ikke i alle tilfelle være den rette representasjonen. I LISP gir Paterson's og Wegman's algoritme en mest generell idempotent unfikator med de substituerende termene på listeform (som en trestruktur), og substitusjonskomponentene er ordnet i en sekvens. Dette er en velegnet form i LISP fordi LISP behandler trestrukturer meget effektivt.

I andre språk kan andre datastrukturer være mere gunstig. I denne oppgaven vil det føre for langt å gå mere i detalj om implementasjon av unifikasjonsalgoritmer.

2.9.2 Spesialiserte unfikasjonsalgoritmer

Ikke alle av de mere effektive unifikasjonsalgoritmene er omtalt i dette kapitlet. Se for eksempel [Martelli, Montanari 82]. I tillegg er det en rekke spesialiserte unifikasjonsalgoritmer. Her nevner jeg unifikasjonsalgoritmer for:

1. Asosiative og kommutative funksjoner.
2. Asosiative (men ikke kommutative) funksjoner.
3. Kommutative funksjoner.
4. Flersortig logikk.

Alle disse unfikasjonsalgoritmene gir et komplett sett av de mest generelle unifikatorer. Med unntak av unifikasjon av asosiative funksjoner så er settet av de mest generelle unifikatorer endelig.

Det finnes også unifikasjonsprosedyrer for høyre ordens logikk. Disse er i beste fall semialgoritmer.

Ingen av unifikasjons-metodene omtalt i dette avsnitt blir behandlet noe mere i denne oppgaven. Se [Huet 86], [Stickel 86] og [Bibel 87] for omtale og referanser. Videre har Siekmann skrevet oversiktsartikler om unifikasjon. Se for eksempel [Siekmann 84].

Kapittel 3

Normalisering av utsagn

3.1 Disjunktiv normalform

Et utsagn F i predikatlogikk er på disjunktiv normalform når det er på formen

$$F = \forall y_1 \ldots \forall y_m \exists x_1 \ldots \exists x_n F_0$$

der F_0 er en disjunksjon av klausuler

$$c_1 \vee \ldots \vee c_n$$

og hver klausul består av en konjunksjon av literaler

$$L_1 \wedge \ldots \wedge L_m$$

Et literal er et predikat eller negasjonen av et predikat. F_0 er matrisen til utsagnet F.

Utsagnet F må være lukket og rettet. Lukket betyr at alle variable er kvantoriserte (ikke frie). Rettet vil si at ingen variable er kvantorisert med mer enn en kvantor.

3.2 Normalisering i utsagnslogikk

Foreta følgende omforminger for å få utsagnet F på disjunktiv normalform.

1. Fjern ekvivalens- og implikasjonstegnet ved omformingene:

$$A \Leftrightarrow B := ((\neg A \vee B) \wedge (\neg B \vee A))$$

$$A \Rightarrow B := (\neg A \vee B)$$

2. Sørg for at alle negasjonene kommer nærmest et predikat ved omformingene:

$$\neg(A \vee B) := (\neg A \wedge \neg B)$$

$$\neg(A \wedge B) := (\neg A \vee \neg B)$$

$$\neg\neg A := A$$

3. Få utsagnet på disjunktiv normal form ved omformingen:

$$(A \wedge (B \vee C)) = ((B \vee C) \wedge A) := (A \wedge B) \vee (A \wedge C)$$

Den siste omformingen har i verste fall eksponsiell kompleksitet med hensyn på tid og størrelse. Men dette kan unngåes som beskrevet i avsnitt 3.4.

3.3 Normalisering i predikatlogikk

En prosedyre for å få normalisert et utsagn til Skolem disjunktiv normalform blir beskrevet her. Se [Bibel 87, III.4] og [Nossum 84].

3.3.1 Lukket og rettet form

Få utsagnet på lukket og rettet form med følgende skritt:

1. Fjern ekvivalenstegnet ved omformingen:

$$A \Leftrightarrow B := ((\neg A \vee B) \wedge (\neg B \vee A))$$

2. Lukk utsagnet ved å introdusere en $\forall$-kvantor for hver distinkt variabel x_i som forekommer fri i F.

3. Få utsagnet på rettet form ved om nødvendig å gi nye navn til variable, slik at ikke to kvantorer har samme variabel.

Kall utsagnet $F^{\bullet}$.

3.3.2 Negasjons normalform

1. Fjern implikasjonstegnet ved omformingen:

$$A \Rightarrow B := (\neg A \vee B)$$

2. Skyv negasjonstegnet innover slik at negasjonen bare omfatter et predikat. Bruk omformingene:

$$\neg(A \vee B) := (\neg A \wedge \neg B)$$

$$\neg(A \wedge B) := (\neg A \vee \neg B)$$

$$\neg\neg A := A$$

$$\neg(\forall x_1 \ldots \forall x_n F_0) := \exists x_1 \ldots \exists x_n \neg F_0$$

$$\neg(\exists y_1 \ldots \exists y_m F_0) := \forall y_1 \ldots \forall y_m \neg F_0$$

3.3.3 Negative og positive variable

Merk hver distinkt variabel som er $\forall$-kvantorisert som negativ, og merk hver distinkt variabel som er $\exists$-kvantorisert som positiv. De negative variablene kan nå sees på som konstanter. Kall utsagnet F^*.

3.3.4 Skolemfunksjoner

Introduser skolemfunksjoner ved for hver delformel

$$\forall y_i D$$

i utsagnet F^* å foreta substitusjonen

$$f(x_1, \ldots, x_n) \rightarrow y_i$$

der

$$f$$

er funksjonssymbolet til den nye skolemfunksjonen med $n \geq 0$ argumenter og der

$$x_1 \ldots x_n$$

alle er $\exists$-kvantoriserte (positive) variable med eksistenskvantorene $\exists x_1 \ldots \exists x_n$ som omfatter delformelen $\forall y_i D$.

Deretter kan $\forall$-kvantorene fjernes.

3.3.5 Skolem (preneks) normalform

Omform til skolem-normalform ved å flytte alle $\exists$-kvantorene lengst til venstre, slik at vi får utsagnet

$$F' = \exists x_1 \ldots \exists x_n F_0$$

der $x_1 \ldots x_n$ er alle de positive variablene, og F_0 er F^* skolemisert og med $\exists$-kvantorene fjernet. F' er nå også på preneks normalform.

3.3.6 Skolem disjunktiv normalform

Omform utrykkene i F_0 slik:

$$A \wedge (B \vee C) = (B \vee C) \wedge A := (A \wedge B) \vee (A \wedge C)$$

3.3.7 Unngå at to klausuler deler en variabel

For hver variabel x_i som forekommer i de to forskjellige klausulene c_j og c_{j+1} substituer inn i den ene klausulen

$$x_i \rightarrow x_{n+1}$$

der x_{n+1} er en ny variabel.

Erstatt prefikset til F_0

$$\exists x_1 \ldots \exists x_n$$

med et nytt prefiks

$$\exists x_1 \ldots \exists x_{n+1}$$

Etter at normaliseringen er foretatt kan eksistenskvantoren fjernes, slik at vi står igjen med matrisen F_0. Dette kan vi gjøre fordi alle variable i F_0 kun har eksistens-kvantorer, og disse kan nå regnes som underforstått.

3.4 Avgrenset form

Omforming til disjunktiv normalform som beskrevet i avsnitt 3.3.6 kan i verste fall ha eksponensiell kompleksitet. For å unngå dette kan utsagnet F omformes til avgrenset form, før den videre normaliseringen skjer [Bibel 87, III,4.6D].

Avgrenset form fåes ved å omforme F slik:
For hvert delutsagn i F som ikke er et literal,

$$G = \circ(G_1, \ldots, G_n)$$

konstruer delutsagnet

$$D_G = \forall x_1 \ldots x_m (P_G(x_1, \ldots, x_m) \Leftrightarrow \circ(L_{G_1}, \ldots, L_{G_n}))$$

der

$x_1 \ldots x_n$: er de frie variable i G
$P_G(x_1, \ldots, x_m)$: er et nytt atom
$\circ$: er det logiske konnektivet ved roten av G
$G_1 \ldots G_n$: er de delutsagn i G som er argumenter til $\circ$
$L_{G_1} \ldots L_{G_n}$: er literaler konstruert fra delformlene $G_1 \ldots G_n$

L_{G_i} er konstruert fra G_i $(1 \leq i \leq n)$ som følger:

if G_i er et literal
then
$L_{G_i} := G_i$
else
$L_{G_i} := P_{G_i}$
fi

P_{G_i} er et ny relasjon som har de frie variable i G_i som argumenter (tilsvarende P_G). La C_F være konjunksjonen av alle utsagn D_G. Da sier vi at

$$F^A = C_F \Rightarrow L_F$$

er utsagnet F på avgrenset form. Normalisering av F^A har lineær kompleksitet.

For å illustrere omforming til avgrenset form tar jeg med et eksempel.

Eksempel 3.1 *Utsagnet*

$$\exists x, y(R(x) \vee Q(y))$$

omformes til

$$(P_0 \Leftrightarrow \exists x, y P_1(x, y))$$
$$\wedge \forall x, y(P_1(x, y) \Leftrightarrow (R(x) \vee Q(y)))$$
$$\Rightarrow P_0$$

3.5 Alternativ til å introdusere skolemfunksjoner

3.5.1 Ordne variablene i en trestuktur

I stedet for eksplisitt å introdusere skolemfunksjoner som i avsnitt 3.3.4 , så kan variablene ordnes i en trestruktur. Denne trestrukturen skal svare til trestrukturen av F^*, men redusert til å omfatte de kvantoriserte delutsagnene av F^* [Bibel 87, IV,8.1D og 8.4T].

Eksempel 3.2

$$\exists x(\forall y_1(P(x, y_1)) \vee \forall y_2(Q(x, y_2)))$$

gir x som noden til roten i treet og y_1 og y_2 som bladnodene til hver av de to greinene i treet.

Den ordningen av variable som blir krevd i avsnitt 2.7 vil være den som vi får ved å følge en grein i treet fra roten til en bladnode. Dette er en partiell ordning.

Når trestrukturen er laget, så kan de negative variablene behandles som konstanter. $\forall$-kvantorene kan deretter fjernes.

I avsnitt 3.3.7 foretas introduksjon av nye variable, slik at to klausuler ikke deler samme variabel. Hvis dette gjøres her, så må den nye variabelen x_{n+1} settes rett foran eller rett etter (den korresponderende) variabelen x_i i trestrukturen. Unifikasjonsalgoritmen blir som beskrevet i avsnitt 2.7. For å finne ut hvilke positive variable som har en kvantor som omfatter en negativ variabel y_i, så er det nok å traversere greina fra y_i og opp mot roten i treet.

3.5.2 Ordne variablene i en sekvens

Det er også mulig å omforme et utsagn til preneks normalform rett etter omforming til negasjons-normalform. Dette gjøres ved omformingene:

$$(QxA) \circ B = Qx(A \circ B)$$

$$(C \circ (QyD) = (Qy(C \circ D)$$

der Q står for en av kvantorene $\forall$ eller $\exists$, $\circ$ er et av konnektivene $\vee$ eller $\wedge$ og A og D er et delutsagn som omfattes av kvantoren Q. Se [Jervell 86, 4.1].

Alle kvantorene med tilhørende variable vil danne en sekvens som et prefiks til matrisen F_0. Ved å merke variablene som positive og negative, slik som beskrevet i avsnitt 3.3.3, og beholde sekvensen fra prefikset, tar vi vare på den samme informasjonen som ved å introdusere skolemfunksjoner. På denne måten får vi her en sekvens som er en topologisk sortering[1] av trestrukturen fra utsagnet F^*.

Eneste forskjellen for bevisprosedyren er at det blir færre unifikasjoner å velge mellom. Dette fordi noen flere variable omfattes av kvantoren til en positiv variabel. Det gir samme virkning som de ekstra variablene vi får i skolemfunksjoner som innføres etter omforming til preneks normalform.

Unifikasjon foregår som beskrevet i avsnitt 2.7. Nye variable x_{n+1} som blir introdusert i avsnitt 3.3.7, plasseres rett etter eller foran den (korresponderende) variabelen x_i i sekvensen av variable.

[1] Partielt ordnet mengde som er gitt en total ordning.

3.6 Svarekstraksjon og normalisering

Er det ved svarekstraksjon ønskelig å sette σ inn i et utsagn $F^\bullet$ på lukket og rettet form, men ellers slik det var før normaliseringen, da vil to omforminger kreve spesiell behandling ved innsetting av σ.

Variabelskifte som beskrevet i avsnitt 3.3.7 og omformingen til avgrenset form i avsnitt 3.4 fører til at det introduseres nye variable som delvis erstatter de opprinnelige variablene.

Omformingen i avsnitt 3.3.7 blir foretatt for å redusere antall varianter eller kopier av F_0 som bevisprosedyren må lage. Men den er ikke nødvendig for å kunne bruke prosedyren CP_1^1 på utsagn omformet til disjunktiv normalform. Omformingen i avsnitt 3.4 sikrer lineær kompleksitet ved normalisering.

Disse to omformingene kan derfor unngås om ønskelig.

Men hvis disse omformingene er med i preprosseserings-skrittet, og σ skal settes inn i $F^\bullet$, så vil man komme riktig ut ved å sette σ inn i en disjunksjon av varianter av $F^\bullet$. I hver av variantene er det innsatt et sett av nye variable introdusert ved normaliseringen. Hver av disse variablene erstatter en av de opprinnelige variablene i $F^\bullet$ som ble splittet opp under normaliseringen.

Dette fordi de to omformingene som introduserer nye variable, splitter opp deler av utsagnet $F^\bullet$ i to deler og introduserer en ny variabel i den ene delen.

3.7 Normalisering til Skolem konjunktiv normalform

3.7.1 Skolem konjunktiv normalform

Denne formen brukes hvis man ønsker å vise at et utsagn F er utilfredstillbart.

Et utsagn på Skolem konjunktiv normalform er på formen

$$\forall x_1 \ldots \forall x_n F_0$$

der F_0 er en konjunksjon av klausuler

$$c_1 \wedge \ldots \wedge c_n$$

som hver består av en disjunksjon av literaler

$$L_1 \vee \ldots \vee L_n$$

Eksistenskvantorene er fjernet og skolemfunksjoner er satt inn i i matrisen F_0.

3.7.2 Normalisering

I forhold til normalisering til Skolem disjunktiv normalform gjøres disse endringene:

1. Tillukning skjer ved å føye til $\exists$-kvantorer for alle variable som forekommer frie i F.

2. $\exists$-kvantoriserte variable merkes som negative og $\forall$-kvantoriserte variable merkes som positive.

3. Skolemfunksjoner settes inn for alle $\exists$-kvantoriserte variable og variablene i skolemfunksjonene bestemmes av de $\forall$-kvantoriserte variable som omfatter delutsagn av typen $\exists y_i D$.

4. Skolem konjunktiv normalform oppnåes ved omformingen:

$$A \vee (B \wedge C) := (A \vee B) \wedge (A \vee C)$$

Kapittel 4

Forbindelsesmetoden

4.1 Forbindelsesmetoden for utsagnslogikk

I dette avsnittet vil forbindelsesmetoden for utsagnslogikk bli omtalt.

Forklaringene av de grunnleggende begreper vil være uformelle, i det jeg viser til presise definisjoner i [Bibel 87, II]

Forbindelsesmetoden er laget også for utsagn på vilkårlig form, men i min beskrivelse av metoden vil jeg gå ut fra at utsagnet er på disjunktiv (eventuelt på konjunktiv) normal form.

4.1.1 Blokkering av veier gjennom en matrise

Vei gjennom en matrise

La klausulene fra utsagnet F være kolonnene i matrisen til F. En vei gjennom matrisen til F er da en disjunksjon av literaler

$$L_1 \vee \ldots \vee L_n$$

der hvert literal er et element fra en klausul i F. En vei kan *ikke* inneholde mere enn ett literal fra samme klausul.

Blokkering av en vei

To literaler på en vei med samme atom, men hvor det ene atomet er negert og det andre ikke, kalles en komplementær forbindelse.

Eksempel på en komplementær forbindelse er

$$K \vee \neg K$$

En vei som inneholder en komplementær forbindelse sier vi er blokkert.

Blokkering av alle veier gjennom en matrise

Utsagnet F er gyldig hvis og bare hvis alle veier gjennom matrisen til utsagnet F er blokkert (eventuelt utilfredstillbar hvis matrisen er på konjunktiv normalform).

4.1.2 Beskrivelse av forbindelsesmetoden

Målet for forbindelsesmetoden er å teste om alle veier gjennom matrisen er blokkerte. Metoden beskriver tre typer skritt, utvidelse, sammentrekning og separasjon.

Disse skrittene er satt sammen slik, enten

utvidelse

eller

utvidelse etterfulgt av sammentrekning

eller

separasjon.

De tre typer av skritt vil først bli beskrevet uformelt og kansje ikke helt presist. For en mere presis beskrivelse viser jeg til prosedyren CP_1^0, eller den presise definisjonen i [Bibel 87, II, 4.2.D og 4.3.D].

Oppstart

Som første skritt i utledningen velges en startklausul. Dette skrittet etterfølges av en utvidelse.

Utvidelse

La en klausul c være enten en startklausul, eller den klausul som ble valgt til forrige utvidelse, eller den største klausulen i den ordnede sekvens av klausuler som vi har etter en sammentrekning.

Fra klausulen c velges et literal L for å forlenge den aktive vei p. Dette literalet er ikke med i noen komplementær forbindelse som er funnet ved tidligere utvidelser langs den aktive vei.

De literalene fra c, som alternativt kunne blitt valgt, blir merket som delmål. Literaler fra delmålet vil seinere kunne velges for å forlenge den aktive vei etter en sammentrekning.

Deretter fullføres utvidelsesskrittet ved at en klausul med minst et literal komplementært med et literal på aktiv vei blir valgt for utvidelse. Klausulen som velges for utvidelse må velges blant klausulene i et sett D. Settet D er alle de klausuler det ikke går noen aktiv vei gjennom, og det omfatter heller ikke klausulen som tidligere er valgt for utvidelse.

Alle literaler som er komplementære med literaler på den aktive vei p merkes nå med at de ikke er kandidater til neste forlengelse av den aktive vei.

Hvis ingen klausul kan velges for utvidelse, blir utvidelsen enten etterfulgt av separasjonskrittet, eller prosedyren stopper fordi utsagnet F ikke er gyldig.

Utsagnet F er ikke gyldig dersom den aktive veien p går gjennom alle klausuler i matrisen til utsagnet F. Men hvis det er noen klausuler som den aktive vei ikke går igjennom ($D \neq \emptyset$), så etterfølges utvidelsesskrittet av separasjonsskrittet.

Sammentrekning

Sammentrekning finner sted når den klausul som ble valgt for utvidelse kun har literaler som er komplementære med et literal på den aktive vei. Etter sammentrekningen blir den største klausulen c med delmål å betrakte som klausul valgt for utvidelse.

Settet D vil nå også omfatte klausuler uten delmål som er større enn c. Den aktive vei blir forkortet så den ikke lenger går gjennom c eller klausuler i D. Klausuler i D kan alltid velges for utvidelse, og alle literalene kan danne komplementær forbindelse med et element fra den aktive veien p.

Finnes ingen klausuler c med delmål langs den aktive vei, så er utsagnet *gyldig*.

Separasjon

Hvis et utvidelsesskritt ikke finner noen klausul for utvidelse, og $D \neq \emptyset$ da prøves separasjon. De klausuler som den aktive vei går gjennom, regnes nå som ferdig behandlet.

En startklausul velges fra D, og den aktive vei starter med et literal fra denne klausulen. Resten av utledningsskrittende omfatter bare D.

4.1.3 Prosedyren CP_1^0

Prosedyren CP_1^0 er hentet fra [Bibel 87, II,6.14.A]. Den definerer forbindelsesmetoden for utsagnslogikk. Se ellers [Bibel 87, II,4.3.D]

Prosedyre 4.1 (CP_1^0)
Variabelliste

D:	*Det sett av klausuler som til enhver tid kan velges for utvidelse.*
c:	*Den klausul som til enhver tid behandles av prosedyren.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei*
WAIT:	*Stakk med delmål for seinere prosessering.*
L:	*Sist valgte literal på aktiv vei.*
K:	*Et literal på aktiv vei*
K^1, L^1*:*	*Literaler som er elementer av klausulen c og som er komplementære til h.h.v. L og K.*

CP_1^0

comment *Initialisering.*
SKRITT 0:
D := F;
WAIT := ***NIL****;*
comment *Valg av startklausul.*
SKRITT 1:

```
p := ∅;
SKRITT 2:
select en klausul c fra matrisen D;
D := D\c;
comment Utvidelse.
SKRITT 3:
select et literal L fra klausulen c;
c := c\L;
if c ≠ ∅
then
        WAIT := Push(WAIT,(c,p,D))
fi;
p := p ∪ {L};
SKRITT 4:
if D = ∅
then
        Returner("Ugyldig")
fi;
SKRITT 5:
if det ikke finnes noen klausuler d∈D slik at L^1 ∈d
then
        if det ikke finnes noen klausuler d ∈ D slik at
                K^1 ∈d for noen K∈p
        then
                comment Separasjon.
                WAIT := NIL;
                go to 1;
        else
                select c fra matrisen D slik at K^1 ∈ c for
                        noen K∈p
        fi
else
        select c fra D slik at L^1 ∈ c
fi
SKRITT 6:
D := D \c;
SKRITT 7:
c := c\L^1;
for alle literaler K slik at K^1 ∈ c og K∈ p
do
    c := c\K^1
od
SKRITT 8:
if c ≠ ∅
then
        go to 3
fi
```

comment *Sammentrekning.*
SKRITT 9:
if *WAIT =* ***NIL***
then
    Returner("Gyldig")
fi
SKRITT 10:
(WAIT,(c,p,D)) := Pop(WAIT);
go to *3 ;*

CP_1^0 *SLUTT*

4.1.4 Kompletthet, konsistens, konfluens og begrensethet

Forbindelsesmetoden for utsagnslogikk er komplett og konsistent. Se [Bibel 87, II,5.5.C og 5.6.C].

Videre er forbindelsesmetoden konfluent. Det betyr at hvis utsagnet F er gyldig vil CP_1^0 returnere verdien gyldig for vilkårlige valg av klausuler og literaler.

Forbindelsesmetoden for utsagnslogikk er begrenset. CP_1^0 vil terminere etter maksimalt

$$\sharp F * \sharp D$$

utledningsskritt der

$$\sharp D$$

er antall veier gjennom matrisen til F og

$$\sharp F$$

er antall klausuler i matrisen til F. Se [Bibel 87, II,5.7.C og 5.8.C].

Et utledningsskritt er i denne sammenheng enten

en utvidelse

eller

en utvidelse etterfulgt av en sammentrekning

eller

en separasjon inklusive valg av ny startklausul.

4.2 Forbindelsesmetoden for predikatlogikk

4.2.1 Innledning

Avsnittet om forbindelsemetoden for predikatlogikk er organisert som beskrevet nedenfor.

Først kommer en uformell presentasjon av forbindelsesmetoden for predikatlogikk, der det som er forskjellig fra forbindelsesmetoden for utsagnslogikk

er beskrevet. Denne presentasjonen inkluderer også en kort beskrivelse av egenskapene til forbindelsesmetoden.

Deretter følger en prosedyre CP^1_{GRUNN}. Den skal være mest mulig i samsvar med den formelle beskrivelsen av forbindelsesmetoden i [Bibel 87, III, 6.5.D og 6.6.D]. Prosedyren er ikke komplett, siden forbindelsesmetoden ikke er konfluent for gyldige utsagn.

Så drøftes hva som skal til for å lage en komplett prosedyre. Deretter presenteres prosedyren CP^1_1 som er en komplett prosedyre. Denne har ikke med separasjonsskrittet, men prøver i stedet alternative valg av startklausuler.

4.2.2 Nærmere om forbindelsesmetoden for predikatlogikk

Noen egenskaper ved forbindelsesmetoden for predikatlogikk

Forbindelsesmetoden for predikatlogikk er komplett og konsistent [Bibel 87, III,6.8.C].

Men forutsetningen for at en bevisprosedyre i predikatlogikk er komplett er at utsagnet F er på lukket og rettet form. Rettet form betyr at to forskjellige kvantorer ikke binder samme variabel. Lukket form vil si at alle variable må være omfattet av en kvantor.

For predikatlogikk finnes ingen generelle bevisprosedyrer som er algoritmer. Derfor gir heller ikke forbindelsesmetoden grunnlag for noen *generell* bevisprosedyre som er en algoritme. Det vil si at bevisprosedyren kan divergere i et forsøk på å validere et utsagn som ikke er gyldig.

Forbindelsesmetoden er *hverken* konfluent eller begrenset i (første ordens logikk) predikatlogikk [Bibel 87, III,6.9.L og 6.10.L].

Beskrivelse av forbindelsesmetoden for predikatlogikk

Forbindelsesmetoden for predikatlogikk er definert mest mulig analogt med forbindelsesmetoden for utsagnslogikk.

Det som kommer i tillegg er unifikasjon av literaler for å teste om en forbindelse er komplementær ved en utvidelse. En forbindelse i predikatlogikk er bare komplementær hvis de to literalene

$$\sigma(L^1, L)$$

har en mest generell unifikator τ. σ er en komposisjon av alle generelle unifikatorer funnet ved tidligere utvidelser.

Ved en utvidelse vil kun *ett* literal fra den aktive vei bli valgt som et komplementært literal til et sett e av literaler fra klausulen c valgt for utvidelse.

Ved en sammentrekning beholdes σ uforandret.

Ved en separasjon settes σ til å være den tomme substitusjonen ε.

Hvis en klausul for utvidelse ikke finnes og separasjon ikke (kan) utføres, så lages en ny kopi av matrisen til F.

En ny kopi av matrisen til F, er det samme som en variant av F. Alle distinkte variable i F er nye og forskjellige fra alle andre kopier av matrisen til F.

Som for utsagnslogikk vil jeg i den videre framstilling forutsette at utsagnet F er på disjunktiv (eventuelt konjunktiv) normalform, hvis ikke annet

er sagt. Men forbindelsesmetoden er også definert for utsagn som ikke er på normalisert form.

Om utsagnet skal være på Skolem normal form eller ikke, avhenger av den unifikasjonsalgoritmen som brukes. Men hvis utsagnet ikke er på Skolem normalform, så må hver distinkt variabel være merket med riktig polaritet (som positiv eller negativ) og med skoping.

Ellers er forbindelsesmetoden for predikatlogikk svært lik forbindelsesmetoden for utsagnslogikk. For en mere presis beskrivelse viser jeg til prosedyrene CP^1_{GRUNN} og CP^1_1 i prosedyrene 4.2 og 4.3 og til definisjonene i [Bibel 87, III,6.5.D og 6.6.D].

Grunnprosedyren

Prosedyren er ment å være en forholdsvis presis beskrivelse av forbindelsesmetoden for predikatlogikk.

Prosedyre 4.2 (CP^1_{GRUNN})
Variabelliste

$F._k$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei.*
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F._1 \cup \ldots \cup F._i \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med delmål for seinere prosessering. Hver innførsel på stakken består av (c,p,A) der* $c \neq \emptyset$.
L:	*Literal som er element av aktiv vei p.*
K:	*Et literal som er element av klausulen c.*
e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1:	*Literal som er element av p og som er komplementært til literaler i e.*

CP^1_{GRUNN}

comment *Initialisering.*
SKRITT 1:
$i := 1;$
$D := F_{.1};$
$\sigma := \varepsilon;$
$A := \emptyset;$
$p := \emptyset;$
$WAIT :=$ ***NIL****;*
comment *Valg av startklausul.*
SKRITT 2:
select *en startklausul c fra matrisen D;*
$D := D \backslash c;$
$A := A \cup \{c\};$
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L;$
if $c \neq \emptyset$
then
    WAIT := Push(WAIT,(c,p,A))
fi*;*
$p := p \cup \{L\};$
comment *Ønskes ny kopi av F?*
SKRITT 4:
if *klausul ønskes fra ny kopi*
then
    comment *Økning av indeks.*
    $i := i + 1;$
    $D := D \cup F_{.i}$
fi*;*
SKRITT 5:
if *ingen klausul for utvidelse finnes på D*
then
    comment *Ikke mulig å teste på betingelsen nedenfor.*
    if *ingen klausul for utvidelse kan finnes på*
        framtidige kopier av F
    then
        comment *Separasjon.*
        WAIT := ***NIL****;*
        $p := \emptyset$
        $\sigma := \varepsilon$
        go to *2*
    else
        comment *Ny kopi.*

i := i+1;
D := D ∪ F.i
go to *5*
fi
else
select *en klausul c fra D slik at σ({L, K})*
er en komplementær og unifiserbar forbindelse for noen
K ∈ c og noen L ∈ p.
fi
comment *Valg av literalsett fra c for utvidelse.*
SKRITT 6:
D := D\c;
A := A∪{c};
select *et sett e som er delsett av c slik at*
σ({L¹} ∪ e) kan unifiseres
med τ som en mest generell unifikator,
og L¹ ∈ p.
comment *Selve utvidelsen.*
SKRITT 7:
c := c\e;
σ := τ ∘ σ;
SKRITT 8:
if *c ≠ ∅*
then
go to *3*
fi
comment *Sammentrekning.*
SKRITT 9:
if *WAIT =* ***NIL***
then
Returner("Gyldig")
fi
SKRITT 10:
(WAIT,(c,p,A)) := Pop(WAIT);
D := F.1 ∪ ... ∪ F.i \ A;
go to *3 ;*

CP^1_{GRUNN} *SLUTT*

4.2.3 Lage en forbindelsesprosedyre som er komplett

En søkestrategi, som er med på å sikre kompletthet for en forbindelsesprosedyre, er søking i bredden. Det betyr at veiene gjennom k kopier av matrisen må undersøkes fullstendig, før kopi $k+1$ blir undersøkt. Hvis dette kravet er oppfylt er man sikret mot at bare deler av utsagnet F blir undersøkt.

For at utsagnet F skal kunne valideres, må et sett av *grunntermer* (termer uten variabler) finnes som kan substitueres inn i de komplementære

forbindelsene som blokkerer alle veiene gjennom variantene (kopiene) av F. Kravet er at alle predikatene i hver komplementær forbindelse blir syntaktisk like etter substitusjonen. Dette får man til ved å finne en mest generell unifikator σ for familien av de sett av literaler som danner de komplementære forbindelsene. De termene i σ, som substitueres inn for variable, kan selv innheholde variable. Grunntermer kan lett oppnåes ved å erstatte disse variablene med et eller annet konstantsymbol.

Dermed kan valg av startklausul, klausul for utvidelse og literalsett for utvidelse påvirke substitusjonen σ som man forsøker å finne. Det er et åpent problem i hvilken grad det er nødvendig å prøve alle muligheter for disse valgene for å sikre at forbindelsesprosedyren er komplett.

Derimot trenger vi ikke prøve alle mulige valg av neste (delmål) literal til aktiv vei. Dette fordi alle literalene, som alternativt kunne blitt valgt, legges på stakken $WAIT$. Ved seinere sammentrekninger vil disse literalene bli prøvd [Bibel 87, III.7], og vi har her slik som forklart nedenfor kontroll ved konstruksjon av σ.

σ er en komposisjon av de mest generelle unifikatorene $\tau_1 \ldots \tau_m$ for de sett av literaler som lager blokkeringer på veiene gjennom matrisen. La T være settet av $\tau_1 \ldots \tau_m$.

Valget av neste literal på aktiv vei kan påvirke den rekkefølge som elementene i T plukkes ut i ved komposisjon av σ. Dette har allikevel ikke noen stor betydning, fordi denne rekkefølgen kan være vilkårlig dersom unifikasjonsalgoritmen returnerer med en idempotent substitusjon τ_i. Se avsnitt 2.6. Siden alle alminnelige brukte unifikasjonsalgoritmer gir en idempotent substitusjon, gir ikke dette noen vesentlig begrensning for bevisprosedyren. Hvis ikke noe annet er sagt vil jeg i resten av oppgaven anta at unifikasjonsalgoritmen som brukes gir en idempotent substitusjon.

Med dette er det klart hva som skal til for å lage en forbindelsesprosedyre som vi vet er komplett.

Kommentarer til den komplette prosedyren

Prosedyren CP_1^1 [Bibel 87, III.7.2.A] er laget slik at den prøver alle mulige valg av startklausul, klausul for utvidelse og literalsett for utvidelse.

Separasjonsskrittet er ikke med i prosedyren. Dettte fordi separasjonsskrittet kan sees på som det samme som å starte med en alternativ klausul i den første kopien av matrisen til utsagnet F. Se [Bibel 87, III.7.1.L].

Bredden i søket skjer ved at alle alternative valg blir prøvd ut for k kopier av matrisen til F, før kopi $k+1$ blir laget.

For første kopi blir disse alternative valg prøvd:

1. Valg av literalsett for utvidelse.
2. Valg av klausul for utvidelse.
3. Valg av startklausul.

For de neste kopiene blir disse alternative valg prøvd:

1. Valg av literalsett for utvidelse.

2. Valg av klausul for utvidelse.

Videre vil situasjonen rett foran alle de utvidelser som er foretatt etter at forrige kopi ble laget, være lagret på stakken $NEXTi$. CP_1^1 vil etter at en ny kopi er laget *igjen* prøve:

Alle utvidelser slik at også klausuler fra den nye kopien blir valgt.

Rekkefølgen som de alternative valgene blir prøvd, er som indikert ovenfor.

Den komplette prosedyren

Prosedyre 4.3 (CP_1^1)

Variabelliste

$F._k$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
j:	*Indeks som brukes til å angi ordning av klausuler eller literalsett som skal lagres på ALT1,ALT2 eller ALT3.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
d:	*En klausul $d \in D$ som er kandidat for en utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei.*
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F._1 \cup \ldots \cup F._i \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med delmål for seinere prosessering. Hver innførsel på stakken består av (c,p,A) der $c \neq \emptyset$.*
L:	*Literal som er element av aktiv vei p.*
K:	*Et literal som er element av klausulen c.*
e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1:	*Literal som er element av p og som er komplementært til literaler i e.*
ALT1:	*Stakk med alternative startklausuler. Hver innførsel består av startklausulen (c).*

ALT2:	*Stakk med alternative klausuer for utvidelse. Hver innførsel består av (L,c,A,σ,p,WAIT)*
ALT3:	*Stakk med alternative literalsett for utvidelse. Hver innførsel består av (c,e,A,σ,τ,p,WAIT).*
NEXTi,	
NEXTi+1:	*To stakker som gir situasjonen rett foran valg av en en ny klausul for utvidelse. Hver innførsel består av (L,A,σ,p,WAIT).*

CP_1^1

SKRITT 0:
Om F ikke inneholder noen eksistenskvantorer så bruk CP_1^0.
comment *Initialisering.*
SKRITT 1:
i := 1;
$D := F_{.1}$;
$\sigma := \varepsilon$;
$A := \emptyset$;
$p := \emptyset$;
WAIT := ***NIL****;*
ALT1 := ***NIL****;*
ALT2 := ***NIL****;*
ALT3 := ***NIL****;*
$\forall i$ *NEXTi :=* ***NIL****;*
comment *Valg av startklausul.*
SKRITT 2:
select *en nummerering* $c_1, \ldots, c_m$ *for alle klausuler i D;*
for *j := m,...,2*
do
    ALT1 := Push(ALT1,(c_j)*);*
od
$c := c_1$;
$D := D \backslash c$;
$A := \{c\}$;
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L$;
if $c \neq \emptyset$
then
        WAIT := Push(WAIT,(c,p,A))
fi*;*
$p := p \cup \{L\}$;
comment *Valg av klausul for utvidelse.*
SKRITT 4:

$k := i+1;$
$NEXTk := Push(NEXTk,(L,A,\sigma,p,WAIT));$
select *en ordning* $d_1,\ldots,d_m$ *av alle*
klausuler fra D *som er slik at* $\sigma(\{L_j,K_j\})$
er en komplementær og unifiserbar forbindelse for noen
$K \in d_j$ *og noen* $L_j \in p$ *og* $j = 1,\ldots,m;$
if $m=0$
then
go to 8
fi;
for $j := m,\ldots,2$
do
$ALT2 := Push(ALT2,(L,d_j,A,\sigma,p,WAIT));$
od;
$c := d_1;$
comment *Valg av literalsett fra* c *for utvidelse.*
SKRITT 5:
$D := D\backslash c;$
$A := A\cup\{c\};$
select *en ordning* $e_1,\ldots,e_m$ *av*
alle delsett av c *som er slik at*
$\sigma(\{L^1\}\cup e_j)$ *kan unifiseres*
med τ_j *som en mest generell unifikator,*
der vi krever at $\tau_j \neq \tau_{j^*}$
for $j \neq j^*$ *og* $L^1 \in p;$
for $j := m,\ldots,2$
do
$ALT3 := Push(ALT3,(c,e_j,A,\sigma,\tau_j,p,WAIT))$
od
comment *Selve utvidelsen.*
SKRITT 6:
$c := c\backslash e_1;$
$\sigma := \tau_1 \circ \sigma;$
if $c \neq \emptyset$
then
go to 3
fi
comment *Sammentrekning.*
SKRITT 7:
if $WAIT =$ **NIL**
then
$Returner(\text{"Gyldig"})$
fi
$(WAIT,(c,p,A)) := Pop(WAIT);$
$D := F._1 \cup \ldots \cup F._i \setminus A;$
go to 3 ;
comment *Alternativ utvidelse.*

SKRITT 8:
if $ALT3 \neq$ ***NIL***
then
$(ALT3,(c,e_1,A,\sigma,\tau_1,p,WAIT)) := Pop(ALT3);$
$D := F_{.1} \cup \ldots \cup F_{.i} \setminus A;$
go to *6*
fi
comment *Alternativ klausul for utvidelse.*
SKRITT 9:
if $ALT2 \neq$ ***NIL***
then
$(ALT2,(L,c,A,\sigma,p,WAIT)) := Pop(ALT2);$
$D := F_{.1} \cup \ldots \cup F_{.i} \setminus A;$
go to *5*
fi
comment *Alternativ startklausul.*
SKRITT 10:
if $ALT1 \neq$ ***NIL***
then
$(ALT1,(c)) := Pop(ALT1);$
$D := F_{.1} \setminus c;$
$A := \{c\};$
$\sigma := \varepsilon;$
$p := \emptyset;$
go to *3*
fi
comment *Ny kopi og tilbaketog etter at ny kopi er laget.*
SKRITT 11:
if $NEXTi =$ ***NIL***
then
comment *Lager ny kopi.*
$i := i+1$
fi
comment *Tilbaketog etter at ny kopi er laget.*
$(NEXTi,(L,A,\sigma,p,WAIT)) := Pop(NEXTi);$
$D := F_{.1} \cup \ldots \cup F_{.i} \setminus A;$
go to *4*

CP_1^1 *SLUTT*

Spesiell kommentarer til CP_1^1

Dette avsnittet inneholder noen mere spesielle kommentarer til CP_1^1 [Bibel 87, III.7.2.A]

1. I den første utgaven av [Bibel 87] er det en trykkfeil i prosedyren CP_1^1. En innførsel på $WAIT$ består der av (c, D, σ, p). σ skal *ikke* legges på

$WAIT$. Trykkfeilen er rettet opp i den andre utgaven [Bibel 87]. Den nevnes her fordi feilen kan være vanskelig å oppdage og i tillegg virke forvirrende.

2. På stakken $WAIT$ legges D, der D er det sett av klausuler som det velges blant ved utvidelser (og oppstart).

   Men ved sammentrekning vil D erstattes med det sett D^* av klausuler som er lagret på $WAIT$. Men om nye kopier av matrisen til F er blitt laget etter at D^* ble lagt på $WAIT$, vil CP_1^1 etter en sammentrekning få et sett D av klausuler som *ikke* omfatter de nye kopiene av F.

   Derfor introduserer jeg en liten forandring i min presentasjon av CP_1^1. For å få en D som også har klausuler fra de nye kopiene, lagrer jeg i stedet A på $WAIT$. A består av de klausuler som ikke skal kunne velges til noen utvidelse. Ved hjelp av A kan en D som omfatter alle de nye kopiene av matrisen til F bli konstruert.

3. Stakkene $NEXTk$ (for alle $k > 1$) kan implementeres som to stakker. Dette fordi det bare er stakkene $NEXTi$ og $NEXTi+1$ som inneholder innførsler når CP_1^1 behandler kopi i av matrisen til F.

   Videre kan en merke seg at CP_1^1 starter med å legge innførsler på stakken $NEXTi+1$. Det har som konsekvens at $NEXTk$ er tom for $k = 1$.

4.3 k-kompakte klasser av utsagn

La grunntermer være settet av alle termene som er bygd opp av de funksjonssymboler som forekommer i utsagnet F, og som ikke inneholder variable. Om ingen konstanter forekommer i settet av funksjonssymboler, så utvid settet med en konstant $Init$.

La en grunninstans av F være matrisen til F på Skolem normalform, der alle variable er substituert med grunntermer.

En lukket og rettet formel er k-kompakt om et sett av k eller færre grunninnstanser av formelen er gyldig.

Teorem 4.1 *Hvis F er et 1-kompakt utsagn er det gyldig om alle veier gjennom en grunninstans av F er blokkert.*

Se [Nossum 84, s 15] for teoremet med bevis.

Blokkering av veiene i en grunninstans er definert på samme måte som for utsagnslogikk.

Teorem 4.2 *Hvis F er en k-kompakt formel så er en hvilken som helst rettet variant av formelen*

$$H = (F \vee \ldots \vee F)(k \text{ disjunkter})$$

1-kompakt.

Se [Nossum 84, s 15 og 16] for teoremet med bevis.

Hvilken k-kompakt klasse som et utsagn tilhører, vet man i alminnelighet ikke noe om i det man setter i gang en bevisprosedyre. Men ved å innføre begrepet k-kompakte klasser, gir det grunnlag for en semidesisjonsprosedyre. En slik prosedyre kan fortelle om det finnes en grunninstans som validerer utsagnet (eller gjør det utilfredstillbar) for k eller færre varianter av F. Gitt at vi kun vil prøve å validere utsagn tilhørende opp til en bestemt k-kompakt klasse, så har vi en desisjonsprosedyre [Nossum 84].

Med dette er det avdekket en viktig sammenheng mellom en generell semidesisjonsprosedyre for predikatlogikk og en desisjonsprosedyre for klasser av predikatlogiske utsagn.

Kapittel 5

Skillemerkestrategien

5.1 Innledning

De komplette prosedyrene CP_1^0 og CP_1^1 vil i en del tilfeller teste om igjen på komplementære forbindelser som allerede er testet, for å finne ut om en vei er komplementær. Skillemerkestrategien er en avskjæringsstrategi for forbindelsesmetoden, der en unngår å teste på veier som inneholder en komplementær forbindelse som allerede er funnet.

Dette kapittel vil i sin helhet handle om skillemerkestrategien. Den er vist komplett og konsistent for utsagnslogikk i [Bibel 87, IV.4]. Etter en presentasjon av skillemerkestrategien for utsagnslogikk, utvides den til å gjelde predikatlogikk og den vises komplett og konsistent i avsnitt 5.3.3.

I min behandling av skillemerkestrategien, vil jeg anta at bevisprosedyrene arbeider med utsagn på normalform.

Strategien har sin parallell med forbindelses-graf-resolusjon. Der er strategien å unngå å resolvere på forbindelser det allerede er resolvert over, eller forbindelser nedarvet fra slike forbindelser.

Forbindelses-graf-resolusjon er vist som komplett og sunn for utsagnslogikk. For første ordens logikk er forbindelses-graf-resolusjon kun vist komplett og konsistent når en del restriksjoner er satt.

Definisjonen av forbindelses-graf-resolusjon er komplisert og vanskelig å forstå. Det er heller ikke kjent hva som skal til for å etablere ekvivalens mellom forbindelses-graf-resolusjon og skillemerkestrategien. Derfor viser jeg til omtalen i [Bibel 87, IV.4] for en mere inngående behandling av forbindelses-graf-resolusjon.

5.2 Skillemerkestrategien for utsagnslogikk

Prosedyren CP_2^0 definerer skillemerkestrategien for utsagnslogikk. Den behandler en matrise for utsagnet F på disjunktiv normalform.

I prosedyren skilles det mellom to typer komplementære forbindelser ved valg av literal for utvidelse:

1. Forbindelser der literalet på aktiv vei p er det sist valgte literalet til p.

2. Forbindelser der literalet på aktiv vei p *ikke* er det sist valgte literalet til p.

Først vil prosedyren forsøke å finne en klausul for utvidelse, slik at den komplementære forbindelsen har det sist valgte literal fra den aktive veien p.

Hvis ingen slike klausuler finnes, legges et skillemerke, $'dm'$, på stakken $WAIT$. Ved valg av literaler for utvidelse, legges en $'sc'$ innførsel på stakken $WAIT$, dersom den komplementære forbindelsen har med et annet literal fra den aktive veien enn det sist valgte.

Ved sammentrekningen samles alle $'sc'$ innførsler som leses fra stakken $WAIT$ i et sett av komplementære forbindelser, SC. Når et skillemerke, $'dm'$, leses fra $WAIT$, finner prosedyren den komplementære forbindelsen på SC med høyest indeks.

Det er en komplementær forbindelse som har et literal M på den aktive veien p i matrisen til F. Dette literalet ligger lengst til høyre i matrisen sammenlignet med literalene fra p i de andre forbindelsene i SC. Samtidig ligger M til venstre for den klausulen som var årsak til at skillemerke (som nå behandles) ble lagt på $WAIT$.

Klausulene til høyre for literal M har alle komplementære forbindelser der literalet fra aktiv vei var det sist valgte. Alle innførsler på stakken $WAIT$ med delmål fra disse klausulene fjernes ved sammentrekningen.

Jeg viser ellers til prosedyren CP_2^0 nedenfor, for en presis beskrivelse av skillemerkestrategien.

Prosedyre 5.1 (CP_2^0)

Variabelliste

D:	*Det sett av klausuler som til enhver tid kan velges for utvidelse.*
c:	*Den klausul som til enhver tid behandles av prosedyren.*
d:	*Klausul som er element av D.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei*
WAIT:	*Stakk med:* *- delmål* *- skillemerker* *- sett av forbindelser*
L:	*Sist valgte literal på aktiv vei.*
K:	*Et literal på aktiv vei*
K^1, L^1*:*	*Literaler som er elementer av klausulen c og som er komplementære til h.h.v. L og K.*
SC:	*Sett av forbindelser lest fra stakken WAIT.*
i:	*Indeks som peker på sist valgte klausul for utvidelse.*
j:	*Indeks som peker på det literal fra veien som er med i en komplementær forbindelse.*
merke:	*Forteller hva slags innførsel som ligger på stakken WAIT.* *- 'sg' = delmål* *- 'sc' = komplementær forbindelse med et literal fra aktiv vei som ikke er det sist valgte.*

- 'dm' = Et skillemerke. Blir lagt på stakken når det ikke finnes noen klausul for utvidelse med et komplementært literal til sist valgte literal på aktiv vei.

CP_2^0

comment *Initialisering.*
SKRITT 0:
D := F;
WAIT := ***NIL****;*
SKRITT 1:
$p := \emptyset$*;*
i := 0;
$SC := \emptyset$*;*
comment *Valg av startklausul.*
SKRITT 2:
select *en klausul c fra matrisen D;*
$D := D \backslash c$*;*
comment *Forlengelse av den aktive veien.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L$*;*
if $c \neq \emptyset$
then
    WAIT := Push(WAIT,('sg',i,(c,p,D)))
fi*;*
i := i+1;
$p := p \cup \{(L, i)\}$*;*
SKRITT 4:
comment *Teste på om utsagnet er ugyldig.*
if $D = \emptyset$
then
    Returner("Ugyldig")
fi*;*
comment *Valg av en klausul for utvidelse.*
SKRITT 5:
if *det finnes noen klausuler* $d \in D$ *slik at* $L^1 \in d$
then
    comment *Valg av en klausul som gir en komplementær forbindelse med sist valgte literal fra veien.*
    select *en klausul c fra D slik at* $L^1 \in d$
else
    if *det finnes noen klausul* $d \in D$ *slik at* $K^1 \in d$ *for noen* $(K, j) \in p$
    then
        comment *Valg av en klausul som gir en komplementær*

```
                    forbindelse med et literal på veien, ulik
                    det sist valgte literal på veien.
            select en klausul c fra D slik at K^1 ∈ c for
                    noen (K, j) ∈ p;
            WAIT := Push(WAIT,('dm',i,NIL))
        else
            comment Separasjon.
            WAIT := NIL;
            go to 1
        fi
fi;
comment Utvidelse.
SKRITT 6:
D := D \ c;
comment Fjerner literalet fra c komplementært med sist valgte
            literal på veien p, om et slikt literal finnes på c.
SKRITT 7:
c := c\L^1;
comment Fjerner de literaler fra c som er komplementært med
            et literal fra veien p ulik L.
for alle literaler K slik at K^1 ∈ c
    og (K, j) ∈ p for noen j
do
    c := c\K^1;
    select j slik at (K, j) ∈ p;
    WAIT := Push(WAIT,('sc',i,{j}))
od;
SKRITT 8:
if c ≠ ∅
then
        go to 3
fi;
comment Sammentrekning.
            Teste på om formelen er gyldig.
SKRITT 9:
if WAIT = NIL
then
        Returner("Gyldig")
fi;
comment Gå tilbake til et delmål.
SKRITT 10:
if merket på toppen av WAIT er 'sg'
then
        (WAIT,(merke,i,(c,p,D))) := Pop(WAIT);
        go to 3
fi;
comment Lagre en forbindelse på SC.
```

SKRITT 11:
if *merket på toppen av WAIT er 'sc'*
then
    (WAIT,(merke,i,SC')) := Pop(WAIT);
    SC := SC ∪ SC';
    go to *9*
fi;
comment *Hoppe over noen delmål.*
SKRITT 12:
if *merket på toppen av WAIT er 'dm'*
then
    SC := SC \ {j | j >indeks på toppen av WAIT} ;
    while *indeksen på toppen av WAIT ≥ maksimum av SC*
    do
        (WAIT,(merke,indeks,dummy)) := Pop(WAIT)
    od;
    go to *9*
fi;

CP_2^0 *SLUTT*

5.2.1 Spesielle kommentarer

De spesielle kommentarene gjelder en feil i prosedyren CP_2^0, og en liten kommentar angående beviset for at denne prosedyren er konsistent.

Feil i prosedyrern CP_2^0

Feilen befinner seg i punkt 13, og gjelder de linjene som ved sammentrekning behandler innførsler fra stakken WAIT merket 'sg' i CP_2^0 [Bibel 87, IV,4].

```
13: if the label of top of WAIT is 'sg' then
(*)   if i = index of top of WAIT then
        (WAIT,(label,index,(c,p,D))) ← pop(WAIT)
        if SC ≠ ∅ then
          SC ← SC\{ j|j>i}; WAIT ← push(WAIT,('sc',i,SC)); SC ← ∅;
        else (WAIT,(label,i,(c,p,D))) ← pop(WAIT)
```

Testen "if i = index of the top of WAIT" på linja merket (*), vil aldri slå til. Dette fordi stakken $WAIT$ redusert til innførsler av typen 'sg' er monotont stigende med hensyn på indeksen i. I prosedyren vil i alltid økes med 1 etter at en 'sg' innførsel er skrevet på $WAIT$, før noen innførsler av typen 'sc' eller av typen 'dm' legges på stakken. (Se punkt 3 i CP_2^0 [Bibel 87, IV,4]).

Det som i tilfelle ville vært gjort om testen (*) kunne slått til er følgende:

1. Alle forbindelse med indeks $j > i$ fjernes fra SC.

2. Deretter kopieres innholdet i SC til en ny '*sc*' innførsel på $WAIT$. Så tømmes SC.

Operasjonen beskrevet i 1. ovenfor er meningsløs. Når en '*dm*' innførsel leses fra $WAIT$ vil alle forbindelser med j større enn løpende indeks i bli fjernet fra SC. Så vil forbindelsen med maksimal j mindre eller lik i bli funnet.

Operasjonen beskrevet i 2. ovenfor kan ikke være riktig. Alle forbindelser som er overført fra 'sc' innførsler på $WAIT$ til SC skal være der så lenge $j \leq i$. I motsatt fall kan flere delmål enn tillatt bli fjernet fra WAIT, fordi SC ikke er garantert å inneholde den maksimale j. Se beviset for [Bibel 87, IV,4.4.T].

Disse linjene som er kommentert her finnes igjen i prosedyren CP_3^0 i [Bibel 87, IV,5]. Der defineres skillemerkemetoden for matriser av utsagn som ikke er på normalform. Siden slike matriser inneholder undermatriser i en trestruktur, ser disse linjene ut til å være riktige i denne prosedyren. Ved aksessering av en matrise på et nivå lengre nede i trestrukturen, vil det være riktig å tømme SC.

Det er derfor sannsynlig at disse linjene er kommet med i CP_2^0 ved en trykkfeil.

Kommentarer til beviset for at CP_2^0 er konsistent

I min oppgave bygger beviset for at skillemerkestrategien er konsistent for predikatlogikk, direkte på beviset for teorem [Bibel 87, IV,4.4T] som sier at CP_2^0 er konsistent og komplett. Derfor finner jeg det riktig å komme med en liten kommentar til beviset.

Den delen av beviset som går på konsistens går jeg ut fra er et underforstått induksjonsbevis, der induksjonen går over antall '*dm*' innførsler som eksplisitt blir funnet ved sammentrekning med testen

if merket på toppen av $WAIT$ er '*dm*'.

5.3 Skillemerkestrategien for predikatlogikk

5.3.1 Innledning

I dette avsnittet vil jeg behandle skillemerkestrategien for predikatlogikk. Jeg presenterer en bevisprosedyre CP_2^1, der skillemerkemetoden brukes for å validere predikatlogiske utsagn på disjunktiv normalform. CP_2^1 har innarbeidet alle alternative valg og tilbaketog fra CP_1^1, og skillemerkestrategien er innarbeidet mest mulig analogt til CP_2^0. Bevis for at CP_2^1 er konsistent og komplett følger i denne seksjon etter at prosedyren er presentert. Beviset for konsistens bygger direkte på beviset for at CP_2^0 er konsistent [Bibel 87, IV,4.4.T]. Beviset for kompletthet bygger på at CP_1^1 er komplett.

5.3.2 Bevisprosedyren CP_2^1 med kommentarer

Her gir jeg detaljerte kommentarer til bevisprosedyren CP_2^1. Kommentarene omfatter det som er *forskjellig* fra CP_1^1 og CP_2^0. Ellers viser jeg til kom-

mentarene til disse to siste prosedyrene.

Valg av klausul for utvidelse

I CP_2^0 blir alle komplementære forbindelser der L er det sist valgte literal på veien foretrukket. CP_2^1 følger samme søkestrategi.

Alle klausuler $d_i \in D$ der *sist* valgte literal på den aktive veien har en komplementær forbindelse blir prioritert og ordnet i en delsekvens $d_1, \ldots, d_k$.

I en annen delsekvens blir alle klausuler $d_j \in D$ som har en komplementær forbindelse med et annet literal enn det sist valgte på den aktive veien, ordnet i en delsekvens $d_{k+1}, \ldots, d_m$. Disse to delsekvensene blir så slått sammen til en sekvens $d_1, \ldots, d_m$.

I denne nye sekvensen kan en klausul $d \in D$ forekomme to ganger, dersom den har literalsett som danner komplementære forbindelser både med det siste valgte literal på den aktive veien og med et annet literal.

Klausulen d_1 blir umiddelbart valgt for utvidelse, mens de andre klausulene i sekvensen $d_2, \ldots, d_m$ blir lagt på $ALT2$ stakken. Hvilke av delsekvensene hver klausul tilhører blir markert med den logiske variabelen bdm.

Valg av literalsett for utvidelse

I CP_1^1 inneholder en komplementær forbindelse kun et literal fra den aktive veien. Dette i motsetning til CP_2^0 der flere literaler fra veien kan inngå i en komplementær forbindelse.

Tilsvarende vil CP_2^1 bare kunne tillate at et literal fra den aktive veien er med i en komplementær forbindelse. Dette medfører at kun en 'sc' innførsel legges på stakken $WAIT$, etter at en 'dm' innførsel er lagt på $WAIT$.

Stakkene

På stakkene $ALT2$, $ALT3$, $NEXTi$ og $NEXTi+1$ legges den ekstra informasjon som er nødvendig fordi skillemerkestrategien er innarbeidet i prosedyren.

Selve prosedyren

Prosedyre 5.2 (CP_2^1)

Variabelliste

$F_{.k}$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
j:	*Indeks som brukes til å angi ordning av klausuler eller literalsett som skal lagres på ALT1,ALT2 eller ALT3.*
k:	*Antall klausuler på ALT2 som har komplementære forbindelser med sist valgte literal L på den aktive vei.*
g:	*Indeks for å angi en NEXTg stakk.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
d:	*En klausul $d \in D$ som er kandidat for en utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den*

	aktive vei.
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F_{.1} \cup \ldots \cup F_{.i} \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med:* *- delmål* *- skillemerker* *- sett av forbindelser*
bdm:	*Bolsk variabel.* ***false*** *dersom e fra c danner en komplementær forbindelse med sist valgte literal L på den aktive veien p.* ***true*** *dersom e fra c danner en komplementær forbindelse med et annet literal M på veien.*
merke:	*Forteller hva slags innførsel som ligger på stakken WAIT.* *- 'sg' = delmål* *- 'sc' = komplementær forbindelse med literal fra aktiv vei som ikke er det sist valgte literal på aktiv vei.* *- 'dm' = Et skillemerke. Blir lagt på stakken når det ikke finnes noen klausul for utvidelse med et komplementært literal til sist valgte literal på aktiv vei.*
L:	*Sist valgte literal på den aktiv vei p.*
M:	*Et literal på den aktive vei p som ikke er det sist valgte literal L på den aktive vei.*
ind1:	*Indeks som peker på det sist valgte element L på den aktive vei.*
ind2:	*Indeks som peker på literalet M på den aktive vei.*
K:	*Literal som er element av klausulen c.*
e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1, M^1:	*Literal som er element av p og som er komplementært til literaler i e.*
SC:	*Sett av forbindelser lest fra stakken WAIT ved sammentrekning.*
ALT1:	*Stakk med alternative startklausuler. Hver innførsel består av startklausulen (c).*
ALT2:	*Stakk med alternative klausuer for utvidelse. Hver innførsel består av ((L,ind1),SC,c,A,σ,p,bdm,WAIT)*

ALT3: *Stakk med alternative literalsett for utvidelse.*
Hver innførsel består av (SC,c,e,A,σ,τ,p,bdm,ind2,WAIT).

NEXTi,
NEXTi+1: *To stakker som gir situasjonen rett foran valg av en*
en ny klausul for utvidelse.
Hver innførsel består av ((L,ind1),SC,A,σ,p,WAIT).

CP_2^1

SKRITT 0:
Om F ikke inneholder noen eksistenskvantorer så bruk CP_2^0.
comment *Initialisering.*
SKRITT 1:
i := 1;
$D := F_{.1}$;
$\sigma := \varepsilon$;
$A := \emptyset$;
$p := \emptyset$;
$SC := \emptyset$;
ind1 := 0;
bdm := ***true****;*
WAIT := ***NIL****;*
ALT1 := ***NIL****;*
ALT2 := ***NIL****;*
ALT3 := ***NIL****;*
$\forall i$ *NEXTi :=* ***NIL****;*
comment *Valg av startklausul.*
SKRITT 2:
select *en nummerering* $c_1, \ldots, c_m$ *for alle klausuler i D;*
for *j := m,...,2*
do
    ALT1 := Push(ALT1,(c_j)*);*
od
$c := c_1$;
$D := D \backslash c$;
$A := \{c\}$;
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L$;
if $c \neq \emptyset$
then
        WAIT := Push(WAIT,('sg',ind1,(c,p,A)))
fi*;*
ind1 := ind1 + 1;
$p := p \cup \{(L, ind1)\}$;

comment *Valg av klausul for utvidelse.*
SKRITT 4:
g := i+1;
NEXTg := Push(NEXTg,((L,ind1),SC,A,σ,p,WAIT));
select *en nummerering $d_1, \ldots, d_k$ av alle*
klausuler fra D som er slik at $\sigma(\{L, K_j\})$
er en komplementær og unifiserbar forbindelse for noen
$K_j \in d_j$, $j = 1, \ldots, k$ og L er sist valgte
literal på den aktive veien p;
select *en nummerering $d_{k+1}, \ldots, d_m$ av alle*
klausuler fra D som er slik at $\sigma(\{M_j, K_j\})$
er en komplementær og unifiserbar forbindelse for noen
$K_j \in d_j$, $j = k+1, \ldots, m$ og noen
$(M_j, ind2_j) \in p \setminus \{(L, ind1)\}$;
if *m=0*
then
go to *11*
fi;
if *k=0*
then
k_1 := 2
else
k_1 := k+1
fi
bdm := ***true;***
for *$j := m, \ldots, k_1$*
do
ALT2 := Push(ALT2,((L,ind1),SC,d_j,A,σ,p,bdm,WAIT));
od;
if *$k > 0$*
then
bdm := ***false;***
for *$j := k, \ldots, 2$*
do
ALT2 := Push(ALT2,((L,ind1),SC,d_j,A,σ,p,bdm,WAIT));
od
fi;
$c := d_1$;
comment *Valg av literalsett fra c for utvidelse.*
SKRITT 5:
D := D$\setminus$c;
A := A$\cup$ {c};
if *bdm =* ***true***
then
WAIT := Push(WAIT,('dm',ind1,NIL))
For alle $(M, ind2) \in p \setminus (L, ind1)$,
finn alle delsett e_j av c som er slik at

$\sigma(\{M^1\} \cup e_j)$ *kan unifiseres*
med τ_j *som en mest generell unifikator,*
der vi krever at $\tau_j \neq \tau_{j^*}$
for $j \neq j^*$;
select *nummering* $e_1, \ldots, e_m$ *av*
alle de delsett e_j *av c som er funnet.*
for $j := m, \ldots, 2$
do
$ALT3 := Push(ALT3,(SC,c,e_j,A,\sigma,\tau_j,p,bdm,ind2_j,WAIT))$
od
else
select *en nummerering* $e_1, \ldots, e_m$ *av*
alle delsett av c som er slik at
$\sigma(\{L^1\} \cup e_j)$ *kan unifiseres*
med τ_j *som en mest generell unifikator,*
der vi krever at $\tau_j \neq \tau_{j^*}$
for $j \neq j^*$ *og* L^1 *er sist valgte literal til veien p;*
for $j := m, \ldots, 2$
do
$ALT3 := Push(ALT3,(SC,c,e_j,A,\sigma,\tau_j,p,bdm,dummy,WAIT))$
od
fi
comment *Selve utvidelsen.*
SKRITT 6:
$c := c \backslash e_1$;
$\sigma := \tau_1 \circ \sigma$;
if $bdm =$ **true**
then
$WAIT := Push(WAIT,('sc',ind1,\{ind2\}))$
fi
if $c \neq \emptyset$
then
go to *3*
fi
comment *Sammentrekning.*
SKRITT 7:
if $WAIT =$ **NIL**
then
Returner("Gyldig")
fi
comment *Gå tilbake til et delmål.*
SKRITT 8:
if *merket på toppen av WAIT er 'sg'*
then
$(WAIT,(merke,ind1,(c,p,A))) := Pop(WAIT)$;
$D := F_{.1} \cup \ldots \cup F_{.i} \setminus A$;
go to *3*

```
fi;
comment Lagre en forbindelse på SC.
SKRITT 9:
if merket på toppen av WAIT er 'sc'
then
      (WAIT,(merke,ind1,SC')) := Pop(WAIT);
      SC := SC ∪ SC';
      go to 7
fi;
comment Hoppe over noen delmål.
SKRITT 10:
if merket på toppen av WAIT er 'dm'
then
      SC := SC \ {ind2 | ind2 >indeks på toppen av WAIT} ;
      while indeksen på toppen av WAIT ≥ maksimum av SC
      do
          (WAIT,(merke,indeks,dummy)) := Pop(WAIT)
      od;
      go to 7
fi;
comment Alternativ utvidelse.
SKRITT 11:
if ALT3 ≠ NIL
then
      (ALT3,(SC,c,e_1, A, σ, τ_1,p,bdm,ind2,WAIT)) := Pop(ALT3);
      D := F.1 ∪ ... ∪ F.i \ A;
      go to 6
fi
comment Alternativ klausul for utvidelse.
SKRITT 12:
if ALT2 ≠ NIL
then
      (ALT2,((L,ind1),SC,c,A,σ,p,bdm,WAIT)) := Pop(ALT2);
      D := F.1 ∪ ... ∪ F.i \ A;
      go to 5
fi
comment Alternativ startklausul.
SKRITT 13:
if ALT1 ≠ NIL
then
      (ALT1,(c)) := Pop(ALT1);
      D := F.1 \ c;
      A := {c};
      σ := ε;
      p := ∅;
      SC := ∅;
      ind1 := 0;
```

go to *3*
fi
comment *Ny kopi og tilbaketog etter at ny kopi er laget.*
SKRITT 14:
if *NEXTi =* ***NIL***
then
comment *Lager ny kopi.*
i := i+1
fi
comment *Tilbaketog etter at ny kopi er laget.*
(NEXTi,((L,ind1),SC,A,σ,p,WAIT)) := Pop(NEXTi);
$D := F_{.1} \cup \ldots \cup F_{.i} \setminus A$;
go to *4*

CP_2^1 *SLUTT*

5.3.3 Bevis for at CP_2^1 er komplett og konsistent

Argumentasjonen skal vise at CP_2^1 er konsistent og komplett.

Men beviset for at CP_2^1 er komplett setter ingen krav til hvilke betingelser som skal oppfylles for at et delmål kan fjernes under sammentrekning. Da kan eventuelle varianter av skillemerkestrategien brukes, forutsatt at vi fortsatt kan vise konsistens.

Noe av den innsikten som ligger i beviset for kompletthet er:

1. Alle delmål på stakken $WAIT$, vil før eller seinere behandles dersom deduksjonen terminerer med verdien "Gyldig". Dette medfører at fjerning av delmål forkorter deduksjonen.

2. Den kortere deduksjonen vil ha de samme utvidelsesskrittene som den opprinnelige deduksjonen, bortsett fra de deduksjonsskrittene som er fjernet.

3. CP_2^1 prøver alternative valg. Den finner enten deduksjonen beskrevet i punkt 2, eller en annen deduksjon som gir "Gyldig", dersom utsagnet er gyldig.

I denne rapporten lar jeg et skritt i utledningssekvensen omfatte hele eller *deler* av et utledningsskritt slik som forutsatt i [Bibel 87, II,5.8C].

Beviset for kompletthet er i hovedsak bygd opp med argumentasjon for:

1. At den valgte søkestrategien i CP_2^1 er komplett. Se Teorem 5.1.

2. At "gale" valg med etterfølgende delsekvens av utvidelser kan fjernes, uten at noe annet forandres i utledningssekvensen. Se Hjelpesetning 5.1.

3. At fjerning av delmål fører til konsekvenser som beskrevet nedenfor:

   - Utvidelser kan bli fjernet, men ikke forandret.

Prosedyre	Utledningssekvens	Knyttet til:	Beskrivelse
CP_1^1	U_1		Forbindelsesmetoden uten avskjæringer.
CP_A^1	U_A	Teorem 5.1	Bruker samme søkestrategi som CP_2^1. Ellers er den som CP_1^1.
CP_B^1	U_B	Hjelpesetning 5.1	Får ikke lov å gjøre "gale" valg. Ellers som CP_A^1.
CP_C^1	U_C	Hjelpesetning 5.2	Fjerner delmål. Ellers som CP_B^1.
CP_2^1	U_2	Teorem 5.2	Skillemerkstrategien innarbeidet. Tillater "gale" valg. Ellers som CP_C^1.

Tabell 5.1: Sammenhengen mellom prosedyrene i Vedlegg A og beviset for at CP_2^1 er komplett

- Sammentrekninger til et delmål som beholdes vil fortsatt finne sted. Sekvensen av utvidelsene som umiddelbart følger etter beholdes. Spesielt beholdes alltid den siste sammentrekningen når $WAIT$-stakken er tom (dvs den sammentrekningen som validerer utsagnet). En annen måte å se det på, er at sammentrekningene til de delmål som fjernes slåes sammen med sammentrekningen til delmålet som beholdes.
- Sekvensen av utvidelser og sammentrekninger er uforandret.

Se Hjelpesetning 5.2

4. At hvis utledningssekvensen vi får ved Hjelpesetning 5.2 terminerer med å validere matrisen, da kan delsekvenser som begynner med "gale" valg og avsluttes med tilbaketog settes inn, og sekvensen vi får, validerer fortsatt utsagnet. Se Teorem 5.2, Hjelpesetning 5.3 og Hjelpesetning 5.4.

I beviset viser jeg til tre prosedyrer som jeg kaller CP_A^1, CP_B^1 og CP_C^1. Disse prosedyrene er bare ment å å bli brukt som støtte ved analysen av CP_2^1. Sammenhengen mellom de forskjellige deler av beviset og disse prosedyrene, går fram av Tabell 5.3.3. Prosedyrene finnes i Vedlegg A.

Beviset for at skillemerkestrategien er komplett er ikke avhengig av den valgte søkestrategi i CP_2^1.

Beviset for at CP_2^1 er konsistent bygger direkte på beviset for at prosedyren CP_2^0 for utsagnslogikk er konsistent. Se beviset for [Bibel 87, IV,4.4.T] og 5.4. Disse to bevisene må derfor leses i sammenheng.

Teoremer og hjelpesetninger med bevis

Teorem 5.1 (Søkestrategi) *CP^1_A vil terminere med verdien "Gyldig" hvis utsagnet F er gyldig.*

Bevis:
Vi kan anta at CP^1_1 er komplett.

CP^1_A vil velge startklausuler på samme måte som i CP^1_1.

Deretter vil den i skritt 4 først velge alle de klausuler som har et literal som er komplementært med siste literal L på aktiv vei. Disse legges nå på stakken $ALT2$.

Så vil prosedyren velge alle de klausuler som har et literal som er komplementært med et literal $K \neq L$ på den aktive vei. Disse klausulene legges på stakken $ALT2$, under alle de klausulene med komplementære literaler med siste literal på aktiv vei.

Klausuler fra D kan forekomme to ganger på stakken $ALT2$.

I skritt 5 vil hver klausul som er merket med at det har literaler komplementært med sist valgte literal L på aktiv vei, bli behandlet slik at alle literalsett e komplementære med L blir behandlet eller lagt på stakken $ALT3$.

Klausuler merket med at de har literaler komplementære med andre literaler på aktiv vei, vil tilsvarende forårsake at alle literalsett e komplementære med et literal $K \neq L$ blir behandlet eller lagt på stakken $ALT3$.

På grunn av ordningen av klausulene i stakken $ALT2$, så vil alle komplementære forbindelser med siste literal L på aktiv vei, bli valgt før alle komplementære forbindelser med andre literaler på aktiv vei.

Alle literalersett fra en klausul c med komplementære forbindelser med et literal på aktiv vei blir prøvd, om nødvendig ved at samme klausul blir behandlet i to omganger som beskrevet ovenfor.

Betingelsen $\tau_j \neq \tau_{j^*}$ for $j \neq j^*$ vil ikke bli prøvd dersom e_j er et literalsett komplementært med siste literal på veien, og e_{j^*} er et literalsett komplementært med et annet literal på veien. Dette vil allikevel bare føre til at et endelig antall ekstra literalsett fra klausulen c unødvendig blir lagt på stakken $ALT3$. Dette antallet overstiger ikke det antall literalsett fra c som blir lagt på stakken $ALT3$ i prosedyren CP^1_1. Om prosedyren terminerer blir ikke påvirket av denne svakheten i testen i CP^1_A (Alternativt kan en slik test legges inn i prosedyrene CP^1_A og CP^1_2).

Vi kan konkludere med at den valgte søkestrategi fortsatt sikrer at alle mulige utledningssekvenser blir prøvd ut. □

Hjelpesetning 5.1 (Fjerne alternative valg) *Anta at CP^1_A lager en utledningssekvens U_A, som avsluttes med en sammentrekning som validerer utsagnet F. Anta videre at sekvensen U_A består av oppstart, utvidelser og sammentrekninger. Dessuten er det delsekvenser som begynner med "gale" valg av startklausul, klausul for utvidelse og literalsett for utvidelse, og der delsekvensene avsluttes med et tilbaketog. Et "galt" valg er her et valg som gir en delsekvens som* ikke *bidrar til de skrittene som validerer utsagnet F.*

Da kan vi fjerne alle "gale" valg med etterfølgende delsekvenser fra utledningssekvensen U_A og vi får en redusert utledningssekvens U_B som fortsatt validerer utsagnet F.

Sekvensen U_B består ikke av andre typer skritt enn:

1. *Oppstart.*
2. *Utvidelse.*
3. *Utvidelse i kombinasjon med at det lages en ny kopi av matrisen til utsagnet til F.*
4. *Sammentrekning.*

Bevis:
Beviset er et induksjonsbevis der induksjonen går over antall "gale" valg som CP_A^1 foretar.
Induksjonsbasis: $k = 0$
Trivielt. Sekvensen som CP_A^1 lager og den reduserte sekvensen U_B blir den samme.
Induksjonsskrittet: $k > 0$
Anta at induksjonshypotesen er sann for $k = n - 1$. Vil da vise at den er sann for $k = n$
Deler resten av beviset opp i 5 tilfeller.

1. Sekvensen som følger etter det k'te "gale" valget inngår i en større delsekvens som følger etter det i'te "gale" valget ($i < n$). Ved induksjonshypotesen kan sekvensen etter det i'te "gale" valget fjernes, og dermed kan også sekvensen etter det k'te "gale" valget fjernes.

2. Valg av alternative sett av literaler for utvidelse.
   Anta at CP_A^1 lager delsekvensen

   $$\cdots \vdash E_{LB} \vdash \cdots (\vdash E_{LV\,k} \vdash E_{LRk} \vdash \cdots \vdash E_M) \vdash \cdots$$

   der

E_{LB}:	Er den del av utvidelsen som utføres før valg av literalsett.
$E_{LV\,k}$:	Det k'te gale valg, som her er valg av literalsett.
E_{LRk}:	Fullføringen av utvidelsen.
E_M:	Et forsøk på utvidelse som leder til tilbaketog og alternativt valg av literalsett for utvidelse, om slikt valg er mulig.

   Siden delsekvensen

   $$(\vdash E_{LV\,k} \vdash E_{LRk} \vdash \cdots \vdash E_M)$$

   ikke bidrar til å validere utsagnet F, kan den fjernes.

   Ved induksjonshypotesen er de andre delsekvensene som begynner med "galt" valg og ender med tilbaketog fjernet.

Et alternativt valg av literalsett forutsetter tilbaketog til situasjonen rett foran valg av literalsett for utvidelse. Da kan bare *ett* av de alternative valgene gi bidrag til den del av den reduserte utledningen som validerer utsagnet F. Dermed kan vi redusere utvidelsen til

$$\cdots \vdash E_{LB} \vdash E_{LVR} \vdash E_{LRR} \vdash \cdots$$

der

E_{LVR}: Det riktige valg av literalsett.
E_{LRR}: Fullføring av utvidelsen.

Vi trenger heller ikke stakken $ALT3$, når den reduserte sekvensen lages.

3. Valg av alternative klausuler for utvidelse.
   Anta at CP^1_A lager delsekvensen

$$\cdots \vdash E_{KB} \vdash \cdots (\vdash E_{KVk} \vdash E_{KRk} \vdash \cdots \vdash E_M) \vdash \cdots$$

der

E_{KB}: Er den del av utvidelsen som utføres før valg av klausul.
E_{KVk}: Det k'te gale valg, som her er valg av klausul.
E_{KRk}: Fullføringen av utvidelsen.
E_M: Et forsøk på utvidelse som leder til tilbaketog og alternativt valg av klausul for utvidelse, om slikt valg er mulig.

Siden delsekvensen

$$(\vdash E_{KVk} \vdash E_{KRk} \vdash \cdots \vdash E_M)$$

ikke bidrar til å validere utsagnet F, kan den fjernes.

Ved induksjonshypotesen er de andre delsekvensene som begynner med "galt" valg og ender med tilbaketog fjernet.

Et alternativt valg av klausul forutsetter tilbaketog til situasjonen rett foran valg av klausul for utvidelse. Da kan bare *ett* av de alternative valgene gi bidrag til den del av den reduserte utledningen som validerer utsagnet F. Dermed kan vi redusere utvidelsen til

$$\cdots \vdash E_{KB} \vdash E_{KVR} \vdash E_{KRR} \vdash \cdots$$

der

E_{KVR}: Det riktige valg av klausul.
E_{KRR}: Fullføring av utvidelsen.

Vi trenger heller ikke stakken $ALT2$, når den reduserte sekvensen lages.

4. Valg av alternative startklausuler.
Anta at CP_A^1 lager delsekvensen

$$\vdash S_{KB} \vdash \cdots (\vdash S_{KVk} \vdash \cdots \vdash E_M) \vdash \cdots$$

der

S_{KB}: Initialisering.
S_{KVk}: Det k'te gale valg, som her er valg av startklausul.
E_M: Et forsøk på utvidelse som leder til tilbaketog og alternativt valg av startklausul, om slikt valg er mulig.

Siden delsekvensen

$$(\vdash S_{KVk} \vdash \cdots \vdash E_M)$$

ikke bidrar til å validere utsagnet F, kan den fjernes.

Ved induksjonshypotesen er de andre delsekvensene som begynner med "galt" valg og ender med tilbaketog fjernet.

Et alternativt valg av startklausul forutsetter tilbaketog til situasjonen rett etter initialisering. Da kan bare *ett* av de alternative valgene gi bidrag til den del av den reduserte utledningen som validerer utsagnet F. Dermed kan vi redusere oppstart til

$$\vdash S_{KB} \vdash S_{KVR} \vdash \cdots$$

der

S_{KVR}: Det riktige valg av startlausul.

Vi trenger heller ikke stakken $ALT1$, når den reduserte sekvensen lages.

5. Valg av hvor langt tilbaketog som skal foretas etter at ny kopi er laget.
Anta at CP_A^1 lager delsekvensen

$$\cdots \vdash E_{GBk} (\vdash E_{Gk} \vdash \cdots \vdash E_{M0}) \vdash E_{NYKOPI} \vdash \cdots (\vdash E_{Nk} \vdash \cdots \vdash E_M) \vdash \cdots$$

der

E_{GBk}:	Begynnelse på en utvidelse.
E_{Gk}:	Valg (eller forsøk på valg) av en klausul for utvidelse.
E_{M0}:	Det tilbaketoget som leder til at ny kopi blir laget.
E_{NYKOPI}:	Økning av indeks for ny kopi.
E_{Nk}:	Første valg av klausul for utvidelse etter at en ny kopi er laget. Skrittet innebærer tilbaketog til E_{GBk}. Deretter er skrittet som E_{Gk}, men slik at klausuler fra den nye kopien kan velges.
E_M:	Et mislykket forsøk på utvidelse. Det leder til alternativt første valg av klausul for utvidelse om slikt valg er mulig. Et slikt valg betyr tilbaketog til et annet utvidelsesskritt i sekvensen, etter at ny kopi ble laget. Eventuelt kan skrittet lede til at enda en ny kopi blir laget (hvis $NEXTi$-stakken er tom).

Anta at sekvensen

$$(\vdash E_{Nk} \vdash \cdots \vdash E_M)$$

ikke bidrar til å validere utsagnet F. E_{Nk} (eventuelt E_{Gk}) representerer her det k'te "gale" valget. Derfor kan denne delsekvensen fjernes.

Ved induksjonshypotesen er de andre delsekvensene som begynner med "galt" valg og ender med tilbaketog fjernet.

Et alternativt valg av lengden av tilbaketoget forutsetter tilbaketog til en situasjon før den nye kopien av matrisen til F ble laget. Da kan bare *ett* første valg E_{Nk} på ny kopi gi bidrag til den del av den reduserte utledningen som validerer utsagnet F. Og vi trenger ikke stakkene $NEXTi$ når den reduserte sekvensen lages. Vi kan nå konstruere en noe redusert sekvens

$$\cdots \vdash E_{GBR}(\vdash E_{GR} \vdash \cdots \vdash E_{M0}) \vdash E_{NYKOPI} \vdash E_{NR} \vdash \cdots$$

der

E_{GBR}:	Begynnelse på en utvidelse før en ny kopi er laget.
E_{GR}:	Fortsettelse av utvidelsen, med valg av en klausul.
E_{NR}:	Det første valg av klausul for utvidelse etter at en ny kopi er laget. Skrittet vil si rett tilbaketog til situasjonen rett etter begynnelse av utvidelsen E_{GBR}.

Det er naturlig å fjerne delsekvensen

$$(\vdash E_{GR} \vdash \cdots \vdash E_{M0})$$

på grunn av det tilbaketoget som følger med skrittet E_{NR}. Dette kan vi gjøre siden det skritt som (eventuelt) validerer F alltid kommer til slutt i sekvensen U_A. Vi får nå den reduserte sekvensen

$$\cdots \vdash E_{GBR} \vdash E_{NYKOPI} \vdash E_{NR} \vdash \cdots$$

□

Prosedyren CP_B^1 lager sekvensen U_B.

Hjelpesetning 5.2 (Fjerne delmål) *Anta at CP_B^1 lager en utledningssekvens U_B som bare består av utvidelser, sammentrekninger og oppstart. La CP_C^1 være en prosedyre som fjerner delmål ved sammentrekning, og ellers er som CP_B^1.*

Da finnes en utledningssekvens U_C laget av CP_C^1 slik at:

1. *For hver sammentrekning til et delmål som ikke lages eller fjernes av CP_C^1 så fjernes delsekvensen*

$$T_{sg(f)} \vdash E_{(i,1)} \vdash \cdots \vdash E_{(i,k)}$$

i U_B fra U_C.

$T_{sg(f)}$ er sammentrekningen til delmålet som fjernes eller ikke lages av CP_C^1. Delsekvensen $E_{(i,1)} \vdash \cdots \vdash E_{(i,k)}$ er de etterfølgende utvidelsene fram til neste sammentrekning.

2. *For hver sammentrekning til et delmål som beholdes av CP_C^1, så lages en delsekvens*

$$\mathcal{SEQ}_j = \mathcal{T}_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,k)}$$

i U_C som erstatter delsekvensen

$$SEQ_j = T_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,k)}$$

i U_B. $\mathcal{T}_{sg(b)}$ og $T_{sg(b)}$ er sammentrekningene til delmålet $sg(b)$ i henholdsvis CP_C^1 og CP_B^1. $E_{(j,1)} \vdash \cdots \vdash E_{(j,k)}$ er de etterfølgende utvidelsene fram til neste sammentrekning. Disse utvidelsene er de samme i CP_B^1 og CP_C^1.

3. *For den sammentrekning T_{VALID} som returnerer "Gyldig" fra CP_B^1, så lager CP_C^1 en sammentrekning $\mathcal{T}_{VALID}$ som også returnerer "Gyldig".*

Rekkefølgen av sekvensene $\mathcal{SEQ}_j$ i U_C er den samme som rekkefølgen av sekvensene SEQ_j i U_B. Hvis U_B har T_{VALID} som sitt siste skritt, så har U_C skrittet $\mathcal{T}_{VALID}$ som sitt siste skritt.

.

Bevis:
Forskjellen mellom CP_B^1 og CP_C^1 er at CP_C^1 ikke lager eller fjerner noen delmål fra stakken $WAIT$, mens CP_B^1 behandler alle delmålene som legges på stakken $WAIT$. Beviset er et induksjonsbevis, der induksjonen går over det antall delmål som fjernes eller ikke lages av CP_C^1 men som CP_B^1 behandler.
Induksjonsbasis: $k = 0$
Trivielt. CP_C^1 vil lage samme utledningssekvens som CP_B^1.
Induksjonsskritt: $k > 0$
Anta at induksjonshypotesen er sann for $k = n - 1$. Vil da vise at den er sann for $k = n$.

Beviset deles opp i 3 tilfeller, der de to første tilfellene er generelle, mens det siste tilfellet er et spesialtilfelle av de to første tilfellene.

Tilfelle 1. viser hva som skjer når et delmål blir fjernet av CP_C^1 i en sammentrekning.

Tilfelle 2. viser hva som skjer når et delmål i CP_B^1 aldri blir lagt på stakken $WAIT$ av CP_C^1.

Tilfelle 3. viser hva som skjer når sammentrekningen T_{VALID} i CP_B^1 kommer rett etter en delsekvens som fjernes eller ikke lages av CP_C^1.

Tilfellene:

1. Det k'te delmål $sg(f)$ fjernes ved en sammentrekning i CP_C^1.

   Anta at CP_B^1 lager følgende del av en utledningssekvens:

   $$\cdots \vdash E_{sg(b)} \vdash \cdots \vdash E_{sg(f)} \vdash \cdots$$

   $$\vdash T_{sg(f)} \vdash E_{(i,1)} \vdash \cdots \vdash E_{(i,m)}$$

   $$\vdash T_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)} \vdash \cdots$$

   Vi har her:

   - $sg(f)$ er det siste av delmålene på $WAIT$-stakken som fjernes av CP_C^1 når prosedyren trekker seg sammen til delmålet $sg(b)$.
   - $sg(b)$ er et delmål som både CP_B^1 og CP_C^1 trekker seg sammen til.
   - $E_{sg(f)}$ og $E_{sg(b)}$ er utvidelser som legger delmålene $sg(f)$ og $sg(b)$ på stakken $WAIT$.
   - $T_{sg(f)}$ og $T_{sg(b)}$ er sammentrekninger til delmålene $sg(f)$ og $sg(b)$.
   - Delsekvensen $E_{(i,1)} \vdash \cdots \vdash E_{(i,m)}$ der $(m \geq 1)$, er de utvidelsene som er laget av CP_B^1 etter sammentrekningen til delmålet $sg(f)$.
   - Delsekvensen $E_{(j,1)} \vdash \cdots \vdash E_{(j,l)}$ der $(l \geq 1)$, er de utvidelsene som er laget av CP_B^1 etter sammentrekningen til delmålet $sg(b)$ og fram til neste sammentrekning.

   Da vil CP_C^1 lage følgende del av en utledningssekvens:

   $$\cdots \vdash E_{sg(b)} \vdash \cdots \vdash E_{sg(f)} \vdash \cdots$$

   $$\vdash \mathcal{T}_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)}$$

   Argumentene for at vi får en slik utledningssekvens er som følger:

   Sammentrekningen $\mathcal{T}_{sg(b)}$ vil fjerne delmålet $sg(f)$ fra stakken $WAIT$ og dessuten alle innførsler av typen 'dm' og 'sc' som ligger over delmålet $sg(b)$ på stakken $WAIT$. Til slutt vil sammentrekningen fjerne delmålet $sg(b)$ fra $WAIT$, og samtidig sette c, D og p lik verdiene til delmålets $sg(b)$ sin innførsel på $WAIT$-stakken.

   Siden $sg(f)$ blir fjernet ved sammentrekningen $\mathcal{T}_{sg(b)}$, vil heller ikke utvidelsene

   $$\vdash E_{(i,1)} \vdash \cdots \vdash E_{(i,m)}$$

   kunne bli laget av CP_C^1.

Når CP_C^1 har gjort sammentrekningen til delmålet $sg(b)$, vil dette gi nesten samme situasjon ved hopp til skritt 3 i CP_B^1 og CP_C^1.

Men det vil være to forskjeller:

(a) σ_C laget av CP_C^1 vil være ulik σ_B laget av CP_B^1. Forskjellen er at σ_B har noen substitusjonskomponenter som σ_C ikke har.

Instansiering med σ_C kan ikke hindre CP_C^1 i å unifisere noen literaler son CP_B^1 er i stand til å unifisere. For anta det motsatte. Da *finnes* en substitusjonskomponent i σ_B, men ikke i σ_C som ved substitusjon enten

> gjør to termer med forskjellige predikat- eller funksjonssymboler like

eller

> gjør to termer like, der den ene termen er en variabel for eksempel v_0, og den andre termen en term der v_0 forekommer.

(b) I forbindelse med en utvidelse i CP_C^1 kan det være nødvendig å lage en ny kopi av matrisen til F. Dette fordi den nye kopien kan ha vært laget tidligere av CP_B^1 i en utvidelse som ikke gjennomføres av CP_C^1. Men dette forandrer ikke noe på selve utvidelsesskrittet.

Da har vi at etter sammentrekningen $\mathcal{T}_{sg(b)}$ kan CP_C^1 velge

- det samme literal for å utvide den aktive vei, og
- den samme klausul c for utvidelse, og
- samme literalsett e_j for utvidelse

som CP_B^1 etter sammentrekningen $T_{sg(b)}$. Vi kan derfor konkludere med at CP_C^1 vil lage de samme utvidelsene

$$\vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)}$$

etter sammentrekningen til delmålet $sg(b)$ som CP_B^1.

2. Det k'te delmål $sg(f_n)$ lages ikke av CP_C^1, som en følge av at et annet delmål er fjernet.

Anta at CP_B^1 lager følgende del av en utledningssekvens:

$$\cdots \vdash E_{sg(b)} \vdash \cdots \vdash E_{sg(f_1)} \vdash \cdots$$

$$\vdash T_{sg(f_1)} \vdash \cdots \vdash E_{sg(f_n)} \vdash \cdots$$

$$\vdash T_{sg(f_n)} \vdash E_{(i,1)} \vdash \cdots \vdash E_{(i,m)}$$

$$\vdash T_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)} \vdash \cdots$$

Vi har her:

- $sg(f_1)$ er et delmål nederst på $WAIT$-stakken som fjernes av CP_C^1 når prosedyren trekker seg sammen til delmålet $sg(b)$.

- $sg(f_n)$ er et delmål som ikke blir laget av CP_C^1, men som i CP_B^1 starter den siste sekvens av utvidelser før prosedyren trekker seg sammen til $sg(b)$.
- $sg(b)$ er et delmål som både CP_B^1 og CP_C^1 trekker seg sammen til.
- $E_{sg(b)}$, $E_{sg(f_1)}$ og $E_{sg(f_n)}$ er utvidelser som legger delmålene $sg(b)$, $sg(f_1)$ og $sg(f_n)$ på stakken $WAIT$.
- $T_{sg(b)}$, $T_{sg(f_1)}$ og $T_{sg(f_n)}$ er sammentrekninger til delmålene $sg(b)$, $sg(f_1)$ og $sg(f_n)$.
- Delsekvensen $T_{sg(f_1)} \vdash \cdots \vdash E_{sg(f_n)} \cdots$ fram til siste utvidelse før sammentrekningen $T_{sg(f_n)}$ for $(n \geq 2)$, består av $T_{sg(f_1)}$ og utvidelsene etter sammentrekningen til delmålet $sg(f_1)$ og (eventuelt) sammentrekningene til delmålene $sg(f_2) \ldots sg(f_{n-1})$ og en sekvens av utvidelser etter hver av disse sammentrekningene. Delmålene $sg(f_2) \ldots sg(f_{n-1})$ er alle delmål som ikke blir laget av CP_C^1.
- Delsekvensen $E_{(i,1)} \vdash \cdots \vdash E_{(i,m)}$ for $(m \geq 1)$, er de utvidelsene som er laget av CP_B^1 etter sammentrekningen til delmålet $sg(f_n)$.
- Delsekvensen $E_{(j,1)} \vdash \cdots \vdash E_{(j,l)}$ for $(l \geq 1)$ er de utvidelsene som er laget av CP_B^1 etter sammentrekningen til delmålet $sg(b)$ og fram til neste sammentrekning.

Da vil CP_C^1 lage følgende del av en utledningssekvens:

$$\cdots \vdash E_{sg(b)} \vdash \cdots \vdash E_{sg(f_1)} \vdash \cdots$$

$$\vdash \mathcal{T}_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)} \vdash \cdots$$

Argumentene for at vi får en slik utledningssekvens er som følger:

Ved induksjonshypotesen har vi at CP_C^1 ikke vil lage eller fjerne sammentrekningene $T_{sg(f_1)} \cdots T_{sg(f_{n-1})}$ med de etterfølgende utvidelsene fram til $T_{sg(f_n)}$ inklusive utvidelsen $E_{sg(f_n)}$.

Men da kan heller ikke delmålet $sg(f_n)$ legges på stakken i $WAIT$ i CP_C^1 og følgelig kan ikke prosedyren lage delsekvensen

$$\vdash T_{sg(f_n)} \vdash E_{(i,1)} \vdash \cdots \vdash E_{(i,m)}$$

Sammentrekningen $\mathcal{T}_{sg(b)}$ i CP_C^1 er da den sammentrekningen som fjerner delmålet $sg(f_1)$ fra stakken $WAIT$ og trekker seg sammen til delmålet $sg(b)$. Vi har da at delsekvensen

$$\vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)}$$

av utvidelser kan lages av CP_C^1 ved samme argumentasjon som for det første tilfellet.

3. T_{VALID} kommer rett etter en sekvens som fjernes av CP_G^1.

   Anta at CP_G^1 lager siste del av en av en utledningssekvens:

$$\cdots \vdash T_{sg(f_\psi)} \vdash E_{(i,1)} \vdash \cdots \vdash E_{(i,m)} \vdash T_{VALID}$$

Sammentrekningen T_{VALID} returnerer "Gyldig" fordi $WAIT$-stakken er tom. Den kommer i dette spesialtilfelle i stedet for sekvensen

$$\vdash T_{sg(b)} \vdash E_{(j,1)} \vdash \cdots \vdash E_{(j,l)}$$

for de to første tilfellene.

Sammentrekningen $T_{sg(f_\psi)}$ trekker seg sammen til delmålet $sg(f)$ for det første tilfelle eller delmålet $sg(f_n)$ for det andre tilfellet. Utledningen foran $T_{sg(f_\psi)}$ er enten som i det første tilfellet eller som i det andre tilfellet.

På samme måte som for de to første tilfellene vil CP_C^1 lage siste del av utledningssekvensen slik:

$$\cdots \vdash \mathcal{T}_{VALID}$$

der $\mathcal{T}_{VALID}$ er den sammentrekning som fjerner minst et delmål som er $sg(f)$ for det første tilfellet og $sg(f_1)$ for det andre tilfellet, og som deretter returnerer "Gyldig" etter å ha fjernet (eventuelle) andre delmål og innførsler av typen 'sc' og 'dm'.

Siden CP_C^1 ikke gjør andre ting forskjellig fra CP_B^1 enn å fjerne noen delmål og etterfølgende utvidelser, permuteres ingen av delsekvensene som begynner med en sammentrekning og etterfølges av utvidelser. Det vil tilsvarende ikke skje noe med en T_{VALID} som kommer etter en sekvens av utvidelser som beholdes. □

Vi vil nå se på hva som skjer ved overgangen fra CP_C^1 til CP_2^1. Definisjonen og hjelpesetningene som følger leder opp til teoremet som etablerer kompletthet for CP_2^1.

Definisjon 5.1 (Nivåer) *La U være en redusert utledningssekvens uten de delsekvenser som begynner med "gale" alternative valg og ender med tilbaketog.*

La $\mathcal{U}$ være alle de delsekvenser som starter opp med å utføre et alternativt valg til et av valgene i U, og ender med tilbaketog. La $\mu \in \mathcal{U}$ betegne en slik delsekvens og la alle $\mu \in \mathcal{U}$ være reduserte delsekvenser. Da sier vi at $\mathcal{U}$ representerer det høyeste nivå e ved ekspansjon av U.

Definer tilsvarende rekursivt de underliggende nivåene, ved å la $\mathcal{U}^*$ være alle reduserte sekvenser vi får ved å la $\mu^* \in \mathcal{U}^*$ starte med å utføre et alternativt valg til en sekvens $\mu \in \mathcal{U}$. Hvis $\mathcal{U}$ representerer nivå k, så representerer $\mathcal{U}^*$ nivå $k-1$.

Som vi ser av definisjonen, sier antall nivåer hvor dyp nesting vi har av delsekvenser som begynner med alternative valg og ender med tilbaketog, når vi ekspanderer en redusert utledningssekvens med slike sekvenser.

Hjelpesetning 5.3 (Endelig antall nivåer) *Anta at U er en endelig redusert utledningssekvens, som består av oppstart, utvidelser, generering av nye kopier og sammentrekninger. Da har vi at U gir opphav til et endelig antall* nivåer, *når U ekspanderes med delsekvenser som starter opp med gjennonmføring av alternativt valg og ender med tilbaketog.*

Bevis:
Siden sekvensen U er endelig, ble et endelig antall kopier k av matrisen til F generert da U ble konstruert.

Gitt matrisen til utsagnet F og gitt at k kopier tillates brukt. Konstruer da en sekvens U_{MAX} som er den lengste reduserte utledningssekvens som er mulig å konstruere. Med lengste sekvens mener jeg den sekvens som har maksimalt antall utvidelser. Det er ingen forutsetning at U_{MAX} validerer utsagnet F. U_{MAX} er en endelg sekvens, og argumentasjonen er den samme som i beviset for Hjelpesetning 5.4. Der påvises hvordan en redusert sekvens over et endelig antall kopier, har et endelig antall sammentrekninger, og et endelig antall utvidelser mellom hver sammentrekning.

La e være antall valg i U_{MAX}.

Vil nå vise følgende induksjonshypotese:

> Antall nivåer som U gir opphav til, når U ekspanderes med delsekvenser som begynner med alternative valg og ender med tilbaketog, er maksimalt e nivåer.

Induksjonen går over antall valg e i U_{MAX}.
Induksjonsbasis: $e = 0$
Trivielt. Hverken U_{MAX} eller U har noen valg.
Induksjonsskrittet: $e > 0$
Anta at induksjonshypotesen er sann for $e = n - 1$. Vil vise at den er sann for $e = n$.
Gitt sekvensen U over k kopier og utsagnet F til U. Da kan vi anta at en U_{MAX} med n antall valg er konstruert. La $\mathcal{U}_{MAX}$ være settet av alle mulige reduserte sekvenser som begynner med et alternativt valg til et valg i U_{MAX} og som ender med tilbaketog. Disse sekvensene representerer høyeste nivået e_0 når U_{MAX} ekspanderes.

Hver sekvens $\mu_{MAX} \in \mathcal{U}_{MAX}$ vil ha en lengde på maksimalt $n - 1$ valg. Dette fordi minst et valg er gjort i det delsekvensen μ_{MAX} begynner. Den kan da sees på som del av en vilkårlig redusert sekvens U_{VILK} over k kopier av F, med en lengde på n antall valg eller kortere.

La $\mathcal{U}$ være settet av de reduserte sekvensene som representerer høyeste nivå e_1 når U ekspanderes. Med samme argumentasjon som for μ_{MAX} kan $\mu \in \mathcal{U}$ ikke ha mere enn $n - 1$ mulige valg.

La $\mathcal{U}^*$ være settet av de reduserte sekvensene som begynner med et alternativt valg til et valg i en vilkårlig $\mu \in \mathcal{U}$. Disse sekvensene representerer nivå $e_1 - 1$. Ved induksjonshypotesen har nivå $e_1 - 1$ maksimal høyde $n - 1$. Siden e_1 er et nivå høyere, har den maksimal høyde n. □

Hjelpesetning 5.4 (Finne utledningen som gir "Gyldig") *Anta at CP_C^1 lager en utledningssekvens U_C som avsluttes med $\mathcal{T}_{VALID}$. La videre k være det antall kopier av F som CP_C^1 genererer.*

Vi kan dessuten anta at CP_2^1 ikke finner noen utledningssekvens som redusert til oppstart, utvidelser, generering av ny kopier og sammentrekninger, er forskjellig fra U_C og avsluttes med et skritt som validerer F.

Da vil CP_2^1 finne en utledningssekvens U_2. Denne sekvensen vil være den samme som U_C når den er redusert ved at delsekvenser som begynner med alternative "gale" valg er fjernet. Utledningssekvensen U_2 avsluttes med $\mathcal{T}_{VALID}$ som U_C

Bevis:
Vi beviser hjelpesetningen ved et induksjonsbevis der induksjonen går over antall kopier av F som lages av CP_2^1.
Induksjonsbasis: $k = 0$
Trivielt. F inneholder ingen eksistenskvantorer. Prosedyren CP_2^0 brukes i stedet for CP_C^1 og CP_2^1.
Induksjonsskritt: $k > 0$
Anta at induksjonshypotesen er sann for $k = n - 1$. Når CP_2^1 lager en ny kopi n av F, vil den også på den nye kopien forsøke:

- Alle mulige valg av klausuler for utvidelse.
- Alle mulige valg av literalsett for utvidelser.
- For alle mulige utvidelser på kopi $n-1$, de alternative valg av klausuler om igjen, slik at også klausuler fra ny kopi velges.
- Alternative valg av startklausuler (Hvis $n = 1$).

CP_2^1 vil behandle ferdig alle mulige alternative valg for $n - 1$ kopier, før den lager en ny kopi n. Vi kan anta at utsagnet F, og dermed hver variant av F har et endelig antall klausuler med et endelig antall literaler hver. Dermed blir antall mulige valg av klausuler endelig når en ny kopi behandles. Kravet om at et delsett av literaler e_j fra klausulen c må velges slik at $\tau_j \neq \tau_{j^*}$ for $j \neq j^*$, fører til at samme literalmengde fra samme klausul ikke kan velges i alternative valg når en komplementær forbindelse med et bestemt literal K skal dannes. Vi har da et endelig antall valg av alternative literalsett fra hver klausul ved utvidelse.

Dermed gir sekvensen U_C direkte opphav til et endelig antall delsekvenser som begynner med "galt" alternativt valg og ender med tilbaketog.

Vi vil nå argumentere for at en slik delsekvens er endelig. Anta i første omgang at U_C gir opphav til delsekvenser som selv er reduserte når U_C ekspanderes ved at alternative valg prøves.

Kall en slik sekvens U_G. La U_G være en sekvens som bare består av utvidelser. Siden en ny kopi bare kan lages etter et tilbaketog, må en slik sekvens være endelig.

Nå tillater vi U_G også å inneholde sammentrekninger. Ved hver sammentrekning fjernes minst et delmål fra stakken $WAIT$.

For n kopier kan $WAIT$ til enhver tid inneholde maksimalt

$$(n * \sum_{i=1}^{m}(l_i - 2)) + 1 - s$$

delmål[1] der:

[1] For hver klausul c som legges på stakken $WAIT$, er literalet på aktiv vei og minst et literal valgt for utvidelse fjernet fra c. Hele utrykket er korrigert for startklausulen der kun literalet på aktiv vei er fjernet.

n: Antall kopier av matrisen til F.
l_i Antall literaler i klausulen c_i i matrisen til F.
m: Er antall klausuler i matrisen til F.
s: Er antall sammentrekninger som er gjennomført fra starten av en (redusert) utledning.

Siden s er monotont stigende i en redusert sekvens U_G, og resten av utrykket er konstant når n kopier behandles, er antall mulige sammentrekninger i U_G endelig.

Men en slik redusert delsekvens U_G kan når den ekspanderes, igjen gi direkte opphav til nye reduserte sekvenser U_{G^*}, som igjen gir direkte opphav til nye reduserte sekvenser, osv. Disse sekvensene vil ved samme argumentasjon som for U_G ha en endelig lengde.

Når vi forlater antakelsen om at "gale" alternative valg bare gir reduserte delsekvenser, vil U_C gi delsekvenser som inneholder nye delsekvenser som begynner med alternative valg, som igjen inneholder nye delsekvenser som begynner med alternative valg, osv. Disse sekvensene er nestet inn i hverandre i et antall nivåer.

Det gjenstår å vise at antall nivåer som disse delsekvensene er nestet i, er endelig. Ved Hjelpesetning 5.3 har vi dette.

Med dette har vi et endelig antall vilkårlige sekvenser med endelig lengde, som begynner med "galt" valg og ender med tilbaketog. De kan settes inn når U_C ekspanderes.

Fordi CP_2^1 forsøker alle mulige valg, så må den lage alle de delsekvenser som satt sammen danner sekvensen U_C som CP_C^1 laget (antakelsen i hjelpesetningen er at CP_2^1 ikke finner noen annen sekvens som validerer F).

Alle "gale" alternative valg begynner enten i oppstartingsskrittet eller i utvidelsesskrittet. Da har vi at CP_2^1 avslutter med $\mathcal{T}_{VALID}$, om CP_C^1 (gitt riktige valg) avslutter med $\mathcal{T}_{VALID}$. □

Teorem 5.2 (Fjerning av delmål) *Anta at CP_A^1 har terminert med en utledning som validerer utsagnet F. Da finnes en utlednig laget av CP_2^1 som validerer utsagnet F.*

Bevis:
Hvis prosedyren CP_A^1 har terminert, så er siste skrittet i utledningssekvensen skrittet T_{VALID}.

Hjelpesetning 5.1 gir at utledningssekvensen fra CP_A^1 kan reduseres til en sekvens U_B med oppstart, utvidelser, generering av nye kopier og sammentrekninger, der T_{VALID} kommer tilslutt.

Hjelpesetning 5.2 gir at for den reduserte delsekvensen U_B, så kan delmål fjernes ved sammentrekninger, og siste skrittet T_{VALID} vil (eventuelt) erstattes med et skritt $\mathcal{T}_{VALID}$, som validerer formelen i utledningssekvensen U_C.

La CP_C^1 være en prosedyre som ikke har mulighet til å foreta tilbaketog, men ellers er som CP_2^1. Vi kan da anta at CP_C^1 kan lage en utledningssekvens U_C, som validerer utsagnet F, forutsatt at prosedyren hele tida tar de rette valg av:

- Startklausul.
- Klausul for utvidelse fra rett kopi av matrisen til F.
- Literalsett for utvidelse.

Det står igjen å vise at CP_2^1 vil lage en sekvens som terminerer med en sammentrekning som validerer F.

Fordi U_C er forskellig fra U_B, så vil ikke de delsekvenser som begynner med "galt" valg og ender med tilbaketog være de samme som de CP_1^1 laget, når U_C ekspanderes til sekvensen U_2.

Vi får to tilfeller:

1. CP_2^1 terminerer med en sekvens U_2. Redusert til oppstart, utvidelser, generering av nye kopier og sammentrekninger er den forskjellig fra U_C. Men siden CP_2^1 bare terminerer når den utfører en sammentrekning T_{VALID}, så vil U_2 validere utsagnet F.

2. CP_2^1 finner ingen utledningssekvens som redusert til oppstart , utvidelser, generering av nye kopier og sammentrekninger er forskjellig fra U_C og som terminerer.

   Men siden U_C er en endelig redusert utledningssekvens, går den ogå over et endelig antall kopier av F, før den avsluttes med $\mathcal{T}_{VALID}$. Da gir hjelpesetning 5.4 at CP_2^1 vil lage en utledning U_2. Når den reduseres ved at alle delsekvenser som begynner med gale valg og ender med tilbaketog fjernes, så vil den være lik U_C. Følgelig vil U_2 avsluttes med utledningsskrittet $\mathcal{T}_{VALID}$.

□

Teorem 5.3 (Kompletthet) *Prosedyren CP_2^1 vil terminere og returnere med verdien "Gyldig" etter en utldedning der det predikatlogiske utsagnet F er gitt, dersom utsagnet F er gyldig.*

Bevis:
Teorem 5.1 og Teorem 5.2 gir tilsammen resultatet. □

Teorem 5.4 (Konsistens) *Hvis prosedyren CP_2^1 returnerer med verdien "Gyldig" etter en utledning der utsagnet F er gitt, så er utsagnet F gyldig.*

Bevis:
Argumentasjonen går her ut på å vise at prosedyren CP_1^1 gir en utledning som returnerer med verdien "Gyldig", om CP_2^1 returnerer med verdien "Gyldig".

Utledningen fra CP_2^1 vil bestå av oppstart, utvidelser, generering av nye kopier og sammentrekninger. Dessuten vil alternative valg bli prøvd. Vi kan bruke Hjelpesetning 5.1 til å redusere utledningen slik at den *ikke* inneholder delsekvenser som begynner med et "galt" alternativt valg og ender med tilbaketog. Kall den reduserte utledningssekvensen U_C.

Da har vi en utledningssekvens som er svært lik den som prosedyren CP_2^0 gir. Forskjellen er:

- Unifikasjon blir brukt for å bestemme de forbindelsene som er komplementære.

- Flere kopier av F blir laget (om nødvendig).

Nå må vi påvise at dersom prosedyren CP_C^1 lager en redusert sekvens U_C som terminerer med verdien "Gyldig", så kan prosedyren CP_B^1 også lage en sekvens U_B som terminerer med verdien "Gyldig". Her vil beviset for at CP_2^0 er konsistent [Bibel 87, IV.4.4] langt på vei holde. Men i *tillegg* er det nødvendig å se på konsekvensene av at unifikasjon brukes til å bestemme de komplementære forbindelsene.

I CP_2^0 vil noen veier ikke bli testet som følge av at noen delmål fjernes ved sammentrekninger. Disse veiene har alle komplementære forbindelser som allerede er funnet. Kall settet av disse forbindelsene W.

La oss nå se på CP_C^1 i forhold til CP_B^1. I utgangspunktet har vi ingen garanti for at forbindelsene i W (tilsvarende settet W for CP_2^0) som er funnet, fortsatt er komplementære etter at nye utvidelser er foretatt av CP_B^1. Dette fordi unifikatoren σ som instansierer termene, stadig får nye substitusjonskomponenter etterhvert som forskjellige aktive veier prøves ut.

Kall settet av veier som ikke prøves som aktiv vei av CP_C^1, men som prøves av CP_B^1 for Q. Ta en av veiene $q \in Q$. Den er allerede blokkert av en komplementær forbindelse $w \in W$. Hvis forbindelsen ikke lenger er komplementær når den igjen testes av CP_B^1 i forbindelse med at veien q prøves som aktiv vei, så er det fordi den mest generelle unifikatoren σ instansierer termer med resultatet at:

1. To forskjellig funksjoner i to literaler fører til at de ikke lenger kan unifiseres.

2. I en funksjon i et av literalene forekommer det en variabel v_0, slik at unifikasjon med termen i det andre literalet, som er variabelen v_0, blir forhindret.

La oss nå se nærmere på en av de komplementære forbindelsene $w \in W$, som allerede er funnet ved traversering av den aktive veien p i CP_C^1, og som blokkerer veien q.

Den består av et literal $K \in p$ hentet fra klausulen c_K. Videre består den av et literalsett e fra klausulen c_e valgt for utvidelse (i det en 'dm' innførsel ble lagt på stakken $WAIT$).

Vi ønsker nå at CP_B^1 skal lage en utledningssekvens tilsvarende den CP_1^0 lager, slik at alle forbindelsene $w \in W$ som vi ønsker skal blokkere veiene $q \in Q$, fortsatt er komplementære når CP_B^1 traverserer veiene q.

Dette får vi til ved å la CP_B^1 hente klausulene c_K og c_e for hver komplementær forbindelse $w \in W$ fra hver sin nye kopi F_K og F_e av matrisen til utsagnet F. Ingen andre klausuler velges fra disse kopiene seinere i utledningen.

Med dette oppnår vi at ingen seinere utvidelser gir substitusjonen σ substitusjonskomponenter som ved seinere instansiering forhindrer unifikasjon av literalsettet K og literalsettet e i den komplementære forbindelsen $w \in W$.

Denne konstruksjonen fører til at beviset for at CP_2^0 er konsistent [Bibel 87, IV.4.4] også gir at CP_C^1 er konsistent. Da har vi at prosedyren CP_B^1 kan lage en utledningssekvens U_B som avsluttes med skrittet T_{VALID} om U_C avsluttes med skrittet T_{VALID}.

Sekvensen U_B kan nå ekspanderes med de delsekvenser som begynner med "gale" alternative valg og avsluttes med tilbaketog til utledningssekvensen U_A. Vi bruker her samme argumentasjon som i beviset for Teorem 5.2, der vi viser at U_C kan ekspanderes til U_2.

Sekvensen U_A kan så omformes til sekvensen U_1, fordi forskjellen melleom sekvensene kun er bruk av forskjellige søkestrategier. Argumentasjonen er her helt analog til argumentasjonen for Teorem 5.1. □

Kapittel 6

Andre avskjæringer

6.1 Innledning

Noen andre avskjæringsstrategier i tillegg til skillemerkestrategien vil bli diskutert i dette kapittel.

Først vil reduksjoner bli behandlet. Omtalen vil begrense seg til normaliserte utsagn i predikatlogikk og utsagnslogikk.

Så vil en liten avskjæringsmulighet ved valg av klausul for utvidelse bli behandlet. Valg av klausul for utvidelse kan begrenses til den nye kopien, for *den første* utvidelsen etter at en ny kopi av matrisen til F er laget.

Til slutt diskuteres forbindelsesmetoden og avskjæringer med hensyn på predikatlogiske utsagn på vilkårlig form (utsagn som ikke er normaliserte).

6.2 Reduksjoner

6.2.1 Reduksjoner i predikatlogikk

I utsagnslogikk gjøres de reduksjonene som omtales her, i en egen prosedyre som utføres før prosedyren CP_1^0 settes i gang [Bibel 87, II.6]. Også her er forutsetningen at utsagnet F er på normal form.

I predikatlogikk integreres reduksjonene dynamisk i den generelle bevisprosedyren [Bibel 87, IV.6]. Men ved å sette visse restriksjoner, er det allikevel mulig å la noen av reduksjonene utføres i en egen prosedyre som kjøres før bevisprosedyren settes i gang [Bayerl 86]. Dette gjelder blant annet rent literal reduksjon, tautologisk reduksjon, multippel reduksjon, c inneholdt i d reduksjon og enhetsresolusjon. Men i dette avsnittet vil jeg videre nøye meg med å omtale reduksjoner integrert dynamisk i bevisprosedyren.

Det er to måter å integrere reduksjonene på. Den ene måten forutsetter at substitusjonen σ som bevisprosedyren instansierer termene med før unifikasjon, ikke forandres fordi en reduksjon utføres. Hvis σ forandres når reduksjonen utføres er det nødvendig å inkludere muligheten for tilbaketog. Dette for å sikre at bevisprosedyren fortsatt er komplett.

I det følgende beskriver jeg dynamisk integrasjon av reduksjoner, slik at σ *ikke* forandres. Jeg gir som et eksempel allikevel beskrivelse av dynamisk integrasjon av *en* reduksjon, der σ forandres.

Rent literal reduksjon

Et rent literal er et literal som instansiert med substitusjonen σ, *ikke* har noe komplementært literal i noen klausul som kan velges for utvidelse, inklusive alle mulige varianter av klausulene i matrisen til F.

I bevisprosedyren integreres denne reduksjonen ved at en klausul med et rent literal $\sigma(L)$ ikke velges for utvidelse.

Tautologisk reduksjon

Hvis en klausul d instansiert med substitusjonen σ, har to komplementære literaler, så velges den *ikke* for utvidelse.

c inneholdt i d reduksjon

Gitt to klausuler c og d. Hvis $\sigma(c) \subseteq \sigma(d)$, velges *ikke* d for utvidelse (også kalt "subsumption").

Multippel reduksjon

Hvis flere literaler i en klausul instansiert med σ blir like, blir de alle regnet som et literal. Denne redukjsonen er allerede delvis integrert i bevisprosedyrene CP_1^1 og CP_2^1 ved at kravet om at

$$\tau_j \neq \tau_{j^*}$$

for to alternative valg av literalsettene e_j og e_{j^*} for utvidelse (Skritt 5 i prosedyrene). I tillegg må CP_1^1 og CP_2^1 inneholde en mekanisme slik at disse literalene blir regnet som ett delmål ved forlengelse av veien.

Enhetsresolusjon

Anta at vi har en matrise av et utsagn på formen

$$\mathcal{F} = F \cup \{c \cup \{L^1\}\} \cup \{L\}$$

der L^1 og L er slik at atomene i literalene er like når literalene blir instansiert med substitusjonen σ. Da kan matrisen reduseres til

$$\mathcal{F} = F \cup \{c\}$$

og vi kaller reduksjonen enhetsresolusjon.

Dersom vi nøyer oss med å kreve at $\sigma(L^1, L)$ er komplementær, så er det nødvendig å introdusere tilbaketog. Dette for å gi mulighet for at enhetsresolusjon alternativt ikke velges. Dette er et eksempel på hvordan en reduksjon integreres i bevisprosedyren, slik at tilbaketog blir nødvendig.

Klausul med et literal på aktiv vei

En klausul med et literal som også finnes på aktiv vei, velges *ikke* for utvidelse. I utsagnslogikk er denne begrensningen komplett og konsistent [Bibel 87, IV.6.5T].

På samme måte som for de andre reduksjonene omtalt ovenfor, ser dette ut til å holde også for predikatlogikk. Men vi må kreve at literalet på aktiv vei og literalet i den klausul som *ikke* velges for utvidelse er like når de instansieres med σ.

6.2.2 Reduksjoner i utsagnslogikk

I dette avsnittet omtales reduksjoner i utsagnslogikk som jeg mener ikke uten videre kan eller bør anvendes i predikatlogikk. For en nærmere definisjon og beskrivelse viser jeg til [Bibel 87, IV.6].

Prawitz' matrisereduksjon

Prawitz' matrisereduksjon angir når og hvordan en matrise kan deles i to. En bevisprosedyre (F. eks. CP_1^0) kan så anvendes på de nye matrisene.

Problemet med denne reduksjonen i predikatlogikk, er at vi har en σ instansiert i de to nye matrisene som fører til at alle veiene i de to matrisene blir blokkert. Da finnes det ikke nødvendigvis noen σ som instansiert i den opprinnelige matrisen, fører til at alle veiene gjennom den blir blokkert.

Den samme innvending vil også gjelde en restriksjon på valg av en klausul for utvidelse behandlet i [Bibel 87, IV.6.2T]. Dette fordi denne restriksjonen bygger på Prawitz' matrisereduksjon.

Utvidelse med faktorisering

Utvidelse med faktorisering gir betingelser for når et literal *utenfor* aktiv vei, kan velges til å danne en komplementær forbindelse med et literalsett i en klausul valgt for utvidelse.

Literalet utenfor aktiv vei er fra et delmål som seinere vil være med på en aktiv vei.

I predikatlogikk (F. eks. prosedyrene CP_1^1 og CP_2^1) vil utvidelser som inkluderer faktorisering introdusere nye valg. Selv om vi kunne (det er ikke sikkert at vi kan) tillate *flere* literaler i eller utenfor aktiv vei å danne en komplementær forbindelse med et literalsett valgt for utvidelse, er det etter min mening tvilsomt om vi får noen mere effektiv bevisprosedyre. Dette på grunn av mange nye alternative valg som lett kan bli etterfulgt av tilbaketog seinere.

6.3 Første utvidelse på en ny kopi

Etter at en ny kopi av matrisen til utsagnet til F er generert, så vil prosedyrene CP_1^1 og CP_2^1 foreta et tilbaketog til begynnelsen av en utvidelse foretatt før ny kopi ble generert (Se skritt 11 i CP_1^1). Alternativt vil prosedyren forsøke et tilbaketog til en annen utvidelse.

I det *første* utvidelsesskrittet etter et slikt tilbaketog kan valget av en klausul for utvidelse begrenses til klausuler fra den nye kopien av matrisen til utsagnet F.

Teorem 6.1 (Første utvidelse på en ny kopi) *Anta at vi har prosedyren CP_1^1 (eller CP_2^1). La E_{FIRST} være første utvidelse etter at en ny kopi ble laget av matrisen til F eller første utvidelse etter et alternativt valg av tilbaketog for å starte opp utvidelsene med den nye kopien.*

Da kan valget av en klausul for utvidelsen E_{FIRST} begrenses til klausuler fra den nye kopien. Det samme gjelder de alternative valg av klausuler til E_{FIRST}. CP_1^1 (eller CP_2^1) vil fortsatt være komplett.

Bevis:
Anta at CP_1^1 (eller CP_2^1) lager en del av en utledningssekvens

$$\cdots \vdash E_{OLD1} \vdash \cdots \vdash BT_{NEWCOPY} \vdash \cdots$$

$$\vdash BT_1(\vdash E_{FOLD1} \vdash \cdots \vdash E_{FOLDn}) \vdash E_{FNEW} \vdash \cdots$$

der:

E_{OLD1}:	Utvidelsen dit tilbaketoget BT_1 skjer, etter at en ny kopi er laget.
$BT_{NEWCOPY}$:	Tilbaketog som fører til at ny kopi blir laget.
BT_1:	Tilbaketog til E_{OLD1} etter at en ny kopi er generert.
E_{FNEW}:	Første utvidelsen som velger klausul fra *ny* kopi etter tilbaketoget BT_1.

Sekvensen

$$(\vdash E_{FOLD1} \vdash \cdots \vdash E_{FOLDn})$$

er en sekvens av utvidelser etter tilbaketoget BT_1 der klausulene for utvidelser velges fra de gamle kopiene.

E_{FOLD1} blir i sekvensene over det samme skritt som E_{FIRST} i teoremet.

Hvis $BT_{NEWCOPY}$ trekker seg sammen til E_{OLD1}, har vi det spesialtilfelle at BT_1 og $BT_{NEWCOPY}$ er samme tilbaketog til et siste mislykket forsøk på utvidelse før ny kopi ble laget. Da har vi den kortere sekvensen

$$\cdots \vdash E_{OLD1} \vdash BT_{NEWCOPY} \vdash E_{FNEW} \vdash \cdots$$

I dette spesialtilfelle må klausulen, som velges for utvidelse, hentes fra ny kopi, når den er generert.

Vi har at E_{FOLD1} som er første utvidelse etter at en ny kopi er laget, enten er samme utvidelse som E_{OLD1} eller samme utvidelse som en av de alternative utvidelsene (utvidelse med alternativt valg av klausul og/eller literalsett) til E_{OLD1}. Siden alle alternative utvidelser blir prøvd før en ny kopi blir generert, så vil spesielt sekvensen

$$(\vdash E_{FOLD1} \vdash \cdots \vdash E_{FOLDn})$$

bli laget før ny kopi blir generert.

Tilbaketog til alle utvidelser foretatt etter utvidelsen E_{OLD1} (eller etter en alternativ utvidelse til E_{OLD1}), vil bli prøvd etter at en ny kopi er generert. Derfor vil CP_1^1 (eller CP_2^1) lage utledningssekvensen

$$\cdots (\vdash E_{FOLD1} \vdash \cdots \vdash E_{FOLDn})$$

$$\vdash E_{OLDn+1} \vdash \cdots \vdash BT_{NEWCOPY} \vdash \cdots \vdash BT_{n+1} \vdash E_{FNEW} \vdash \cdots$$

der:

E_{OLDn+1}:	En utvidelse (eller forsøk på utvidelse) som erstattes av utvidelsen E_{FNEW}.
BT_{n+1}:	Tilbaketog til skrittet E_{OLDn+1}.

La E_{FNEW} i sekvensen over være samme utvidelse som E_{FIRST} i teoremet når vi kun velger fra ny kopi.

Med dette har vi vist at vi kan begrense valget av klausuler i første utvidelse etter at ny kopi er laget til klausuler fra den nye kopien. □

6.4 Vilkårlige matriser og effektivisering

Til nå har vi sett på forbindelsesmetoden anvendt på matriser på normal form.

Forbindelsesmetoden er ikke begrenset til utsagn på normal form. Den er også definert for utsagn på vilkårlig form.

En matrise for et utsagn består av klausuler som danner kolonnene i matrisen. En klausul består igjen av et literal *eller en ny delmatrise* når utsagnet ikke er normalisert. På den måten får vi en trestruktur av matriser.

En vei vil traversere matrisen inkludert dens submatriser. Et utsagn på ikke normal form er blokkert dersom alle veiene gjennom matrisen til utsagnet er blokkert.

Dersom et element L i en klausul c er en delmatrise, er veien p blokkert i L hvis alle veiene gjennom delmatrisen L er blokkert.

Det som er sagt over gjelder både utsagnslogikk og predikatlogikk. Se forøvrig [Bibel 87, II.3 og III.6] for mere presise definisjoner.

Den klassiske prosedyren for å transformere et utsagn i predikatlogikk til normal form vil i noen tilfeller føre til en eksponensiell økning av størrelsen av matrisen til utsagnet. Dette kan unngås som forklart i avsnitt 3.4.

Det er ikke før i de aller siste årene man er blitt klar over hvordan eksponensiell kompleksitet ved normalisering kan unngås. Det er en viktig årsak til at det er lagt ned mye arbeid i å utvikle bevisprosedyrer for utsagn på vilkårlig form. Dette er da ment som en effektivisering av bevisprosedyren.

En prosedyre CP_3^0 for utsagnslogikk og for matriser på vilkårlig form finnes i [Bibel 87, IV.5]. Denne prosedyren er ganske kompleks selv om den bare omfatter utsagnslogikk. Den har innarbeidet skillemerkestrategien som i CP_2^0, og en utvidelse av skillemerkestrategien som kalles "fjerning av potensielt overflødige delmål".

Som et annet forslag til effektivisering beskrives i [Bibel 87, IV.8] hvordan man kan unngå å sette inn skolemfunksjoner. Se også avsnitt 2.7 i denne rapporten.

Videre omtales en avskjæringsstrategi som kalles "oppsplitting ved behov". Se [Bibel 87, IV.10].

Gitt et utsagn:

$$\forall x(L(x) \Rightarrow L(vare1) \wedge L(vare2))$$

To varianter av utsagnet vil være nødvendig for å verifisere det. Men hvis man tillater at hver komplementær forbindelse unifiseres uavhengig av hverandre, er det nok med en variant av utsagnet for å verifisere det.

En modifisert unifikasjonsalgoritme tillater en slik forenkling av unifikasjonen. Oppsplitting ved behov er en avskjæringsstrategi utviklet for ikke normaliserte utsagn.

Det er liten grunn til å tro at det i seg selv gir noen effektivitetsgevinst å bruke en bevisprosedyre for utsagn på vilkårlig form i stedet for å normalisere utsagnet (slik jeg ser det). Arbeidet med å normalisere utsagnet blir bare flyttet over fra normaliseringsprosedyren til selve bevisprosedyren.

Men det kan allikevel være argumenter for *ikke* å normalisere et utsagn. Her nevner jeg to slike argumenter:

1. Det er ønskelig å bruke en avskjæringsstrategi som bare er beregnet for utsagn på vilkårlig form.

2. Det er hensiktsmessig med en utledning på en form som ligner den matematikere bruker.

Argumentet for en normalisering av utsagn er at bevisprosedyrer for utsagn på vilkårlig form øker kompleksiteten. Dette gjelder både ved definisjon av bevisprosedyren og ved implementering av prosedyren i form av et datamaskinprogram. Det er også naturlig å tenke seg at større kompleksitet kan gjøre det vanskeligere å vise at bevisprosedyren er komplett og konsistent.

Kapittel 7

Søkestrategier

Dette kapitlet vil kort ta opp valgkriterier for valg av:

- Startklausul.
- Neste literal på aktiv vei.
- Klausul for utvidelse.
- Literalsett for utvidelse.

Problemet er å finne valgkriterier som gir en mest mulig effektiv bevisprosedyre.

Dessuten vil disse valgene bli tatt opp i sammenheng med skillemerkestrategien.

7.1 Valg av neste literal på aktiv vei

Ved valg av literal til å forlenge aktiv vei er det naturlig å velge det literal som gir minst arbeid ved unifikasjon eller forsøk på unifikasjon når en komplementær forbindelse skal finnes.

Kriterier for valg kan her være:

- Det literal som inneholder minst antall distinkte variable.
- Det literal som har minst antall funksjonssymboler.
- Det literal der antall nivåer av deltermer er minst.
- En kombinasjon av de tre første valgkriteriene.

7.2 Valg av startklausul

Ved valg av startklausul vil den klausul som gir færrest delmål nederst på stakken $WAIT$, være grei å velge først. Det betyr at den klausulen med færrest antall literaler velges først.

Hvis flere klausuler har samme antall literaler, så vil forventet arbeid ved unifikasjon av literalene i klausulen med andre komplementære literaler være

grunnlaget for det andre valgkriteriet. For hver literal kan vi se på antall distinkte variable, antall funskjsonssymboler, antall nivåer av deltermer eller en kombinasjon av disse størrelsene. Dette summeres da opp for alle literalene i klausulene. Den klausulen som får den minste verdien velges.

7.3 Valg av klausul for utvidelse

De samme kriterier som for valg av startklausul i avsnitt 7.2 kan brukes.

Ellers kan relevante valgkriterier være å velge den klausul som har oppnådd flest forbindelser med aktiv vei ved tidligere valg eller den klausul som inneholder flest literaler som ved tidligere valg inngikk i en forbindelse med aktiv vei. Alle varianter av klausulen teller med her.

7.4 Valg av literalsett for utvidelse

Målet kan her være å få færrest mulig delmål på stakken $WAIT$. Da velges det literalsett som har flest literaler.

Men det kan hende det er hensiktsmessig å gjøre den mest generelle unifikatoren σ minst mulig i den forstand at den for hver vellykket unifikasjon får færrest mulig nye unifikasjonskomponenter. Dette for å sikre størst mulig frihet ved framtidige valg av klausuler og literalsett for utvidelse. Da vil valgkriteriet være literalsettet med færrest literaler.

Hvis flere literalsett har samme antall literaler, kan det andre valgkriteriet være det samme som det andre valgkriteriet ved valg av startklausul.

7.5 Valgkriterier når skillemerkestrategien implementeres

Dette er kriterier for å få en størst mulig avskjæring tidlig i utledningen.

Første kriterium er å prioritere valg av de literalsett og klausuler som gir en komplementær forbindelse med sist valgte literal på aktiv vei. Prosedyren CP_2^1 har innebygd en slik søkestrategi. Deretter prioriteres valg av klausuler og literalsett slik at det komplementære literal på aktiv vei ligger lengst til venstre (dvs. er valgt tidligst mulig for å forlenge aktiv vei).

7.6 Test av forskjellige valgkriterier

Da det ofte er vanskelig å foreta teoretiske beregninger av kompleksiteten av bevisprosedyrer, vil det være naturlig å implementere forskjellige søkestrategier i en bevisprosedyre. Søkestrategiene kan da testes ved å kjøre eksempler. Resultatene av disse testene kan da være et utgangspunkt for hvilke teoretiske beregninger som vil være fornuftige å foreta.

Kapittel 8

Forslag til videre arbeid

8.1 Innledning

Dette kapittelet vil ta for seg muligheter for videre arbeid, med basis i Bibel's forbindelsesmetode og Nossum's k-kompakthetsteori.

Et avsnitt vil kort beskrive forslag til implementasjon av et automatisk bevisføringssystem.

Deretter vil jeg se på muligheter for og problemer med å utvikle søkestrategier.

Så blir noen ideer for nye avskjæringer beskrevet, og problemer med videre utvikling av komplette avskjæringsstrategier blir i noen grad skissert.

Til slutt blir et par ideer til videre utvikling av Nossum's k-kompakthetsteori presentert.

Det er svært usikkert om og i hvilken grad ideene i dette kapitlet kan gi grunnlag for resultater. Men noen av ideene kan kansje være verd å studere videre.

Videre utvikling med hensyn på implementasjon av et automatisk bevisføringssystem burde kunne gi grunnlag for visse eksperimentelle resultater.

8.2 Videre utvikling av Thoralf

8.2.1 Kort om implementasjonen

Thoralf er en implementasjon av Nossum's k-kompakthets bevisprosedyre. Systemet er utviklet av Rolf Nossum med basis i Nossum's k-kompakthetsteori og Bibel's forbindelsesmetode. Se [Nossum 84], [Nossum 85 II] og [Nossum 85].

Bevisprosedyren er en del av Proversystemet for automatisk programverifikasjon. Det er utviklet ved Universitetet i Oslo, Institutt for informatikk.

Thoralf er skrevet i Pascal og det kjøres på et nettverk av SUN arbeidsstasjoner.

Systemet tar utgangspunkt i en implementasjon av prosedyren CP_1^1 (Se [Bibel 87, III.7.2.A]). Men implementasjonen er noe forskjellig fra denne prosedyren på den måten at separasjon er inkludert, mens tilbaketog og mulighet for alternative valg ikke er implementert.

Det antall kopier (varianter) av matrisen som Thoralf får lov å bruke bestemmes av brukeren på forhånd. Antallet k angir maksimal k-kompakthet

av de utsagn som valideres.

Thoralf er kalt opp etter Thoralf Skolem. For en nærmere beskrivelse av systemet, se [Nossum 85].

8.2.2 Noen forslag til forbedringer av Thoralf

Her vil jeg kort nevne noen forslag for å forbedre Thoralf.

1. Legge inn en effektiv unifikasjonsalgoritme. Aktuelle kanidater er for eksempel den modifiserte Robinson's unifikasjonsalgoritme, unifikasjonsalgoritmen til Paterson og Wegman eller unifikasjonsalgoritmen til Martelli og Montanari. De to første er behandlet i avsnitt 2.3. Se ellers [Corbin, Bidoit 83], [Paterson, Wegman 78] og [Martelli, Montanari 82]. Det kan også finnes andre unifikasjonsalgoritmer som bør vurderes.

2. Effektivisere Thoralf ved å legge inn skillemerkestrategien. Se kapittel 5.

3. Legge inn reduksjoner. Se avsnitt 6.2.

4. Forsøke å forbedre søkestrategien. Se kapittel 7.

5. Implementere en komplett prosedyre som bruker forbindelsesmetoden. Denne prosedyren bør finnes i to utgaver. Den ene utgaven lager et på forhånd bestemt antall kopier av den normaliserte matrisen til utsagnet F. Den andre utgaven lager nye kopier etterhvert som de trenges. Den bør spørre brukeren om en ny kopi skal lages eller om kjøringen skal stoppes.

6. Implementere et tillegg til normaliseringsprosedyren som beskrevet i avsnitt 3.4. Tillegget skal garantere at normaliseringen ikke får eksponensiell kompleksitet.

Tillegget til normaliseringsprosedyren og en komplett prosedyre bør være et alternativ som brukeren kan velge å kjøre eller ikke. Dette fordi normaliseringen som beskrevet i avsnitt 3.4 kan ødelegge noe av den opprinnelige strukturen i utsagnet. Dette kan ha betydning ved svarekstraksjon. Videre vil de tilbaketog som er nødvendig i en komplementær prosedyre føre til vesentlig tap av effektivitet.

En forbedret unifikasjonsalgoritme, skillemerkestrategien og reduksjoner kan legges inn i Thoralf som standard. Det samme vil være tilfelle med en forbedret søkestrategi.

Det vil være særlig nødvendig å eksperimentere for å finne ut hvilke søkestrategier som i praksis gir forbedringer. Videre bør alle forbedringer testes ved å kjøre realistiske eksempler. Dette for å finne ut hvor effektive de er i praksis.

8.3 Søkestrategier

I kapittel 7 om søkestrategier er det nevnt noen valgkriterier som umiddelbart kan se ut til å gi en effektivitetsgevinst.

I det videre arbeid med forbindelsesmetoden bør mulige søkestrategier studeres grundig med hensyn på effektivitet. En slik studie bør også innebære å finne nye fornuftige kriterier for valg.

Forskjellige strategier for valg av startklausul, nytt literal til aktiv vei, klausul for utvidelse og literalsett for utvidelse bør implementeres i Thoralf. Effektiviteten av de forskjellige strategiene kan da måles.

Resultatene fra slike målinger kan da brukes som grunnlag for å velge ut noen søkestrategier for et grundig studium. Et slikt studium vil si å foreta teoretiske beregninger som kan fortelle mere om hvor effektive de er. Slike teoretiske beregninger kan lett bli vanskelige og omfattende.

Det er også mulig å se på søkestrategier for forbindelsesprosedyrer for ikke normaliserte utsagn. Da vil nye kriterier for valg komme inn. Dette fordi et element i en klausul enten kan være en delmatrise eller et literal.

8.4 Avskjæringsstrategier

8.4.1 Avskjæringsstrategier og resolusjon

For resolusjon finnes det flere komplette avskjæringsstrategier. Noen av disse kan være verd å studere med sikte på å finne tilsvarende avskjæringsstrategier for forbindelsesmetoden.

For en kort innføring i resolusjon med noen avskjæringer viser jeg til [Stickel 86]. Ønskes en mere omfattende behandling av resolusjon og avskjæringsstrategier vil [Loveland 78] og [Chang, Lee 73] kunne leses.

Siden forbindelsesmetoden også kan sees på som en variant av lineær resolusjon [Bibel 87, IV.1], trenger en ikke studere avskjæringsstrategier som ikke er kompatible med lineær resolusjon.

At en avskjæringsstrategi A ikke er kompatibel med en annen avskjæringsstrategi B betyr at en bevisprosedyre som bruker både strategien A og strategien B ikke er komplett, selv om hver av strategiene hver for seg er komplette.

P_1 og N_1 resolusjon

Dette avsnittet tar for seg P_1 og N_1 resolusjon som bygger på avskjæringsstrategien "set of support". I denne sammenheng er det også naturlig å nevne muligheten for en preprosesseringsprosedyre som i visse tilfeller kan sile ut utsagn som ikke er gyldige. Se [Stickel 86].

En positiv klausul inneholder ingen negerte predikater. En negativ klausul inneholder kun negerte predikater. En blandet klausul inneholder både negerte predikater og predikater som ikke er negerte.

Et normalisert utsagn F må inneholde minst en positiv klausul og minst en negativ klausul, for å ha mulighet til å være gyldig (eventuelt inkonsistent).

Dette fordi en tolkning som gir verdien "falsk" til hvert predikat i utsagnet F ikke kan være noen modell for F, hvis ikke F inneholder en negativ klausul. Tilsvarende vil en tolkning som gir verdien "sann" til hvert predikat i F ikke kunne være noen modell for F, hvis ikke F inneholder en positiv klausul [Stickel 86].

En preprosesseringsprosedyre kan kjøres etter normalisering av et utsagn. Den skal gi beskjed hvis utsagnet ikke inneholder minst en positiv klausul og minst en negativ klausul.

I resolusjon har vi P_1 og N_1 resolusjon som er en spesialisering av "set of support" strategien og som bygger på det faktum at det normaliserte utsagnet må ha minst en positiv og en negativ klausul. P_1 pg N_1 resolusjon er likeverdige avsskjæringsstrategier (varianter av hverandre).

P_1 resolusjon setter som krav at en av foreldreklausulene skal være en positiv klausul. N_1 resolusjon setter som krav at en av foreldreklausulene skal være en negativ klausul.

P_1 og N_1 resolusjon er komplett og kompatibel med lineær resolusjon.

Det kan være grunn til å se på muligheten for en tilsvarende avskjæringsstrategi for forbindelsesmetoden, i det jeg velger å ta utgangspunkt i P_1 resolusjon.

Sett som tilleggskrav at alle mulige forbindelser som prøves for komplementaritet, skal være slik at enten literalet L fra den aktive veien p er element i en positiv klausul, eller at klausulen c som eventuelt velges for utvidelse er en positiv klausul.

Videre arbeid vil være å finne ut om en slik avskjæringsstrategi er en komplett avskjæringsstrategi for forbindelsesmetoden (eventuelt finne de modifikasjoner som gjør strategien komplett).

8.5 Avskjæring i prosedyrer for utsagn som ikke er normaliserte

En hvilken som helst bevisprosedyre for ikke normaliserte utsagn vil bli ganske komplisert. Det er etter min mening en god nok grunn til ikke å prioritere et studium av slike prosedyrer.

Men det er utviklet avskjæringsstrategier for forbindelsesmetoden for bevisprosedyrer som tar utsagn som ikke er normaliserte. Det kan også være et ønske om å beholde et utsagn mest mulig uforandret (for eksempel fordi svarekstraksjon brukes). Da kan et videre studium av slike bevisprosedyrer være av interesse.

Det videre arbeid vil være å løfte en forbindelsesprosedyre, som tar ikke normaliserte utsagn i utsagnslogikk, til en forbindelsesprosedyre for ikke normaliserte utsagn i predikatlogikk. Denne må sjekkes for å se om den er komplett og konsistent (Se kapittel 4).

Så er det naturlig å inkorporere skillemerkestrategien i prosedyren, og igjen sjekke om den fortsatt er komplett og konsistent (Se kapittel 5).

Deretter må vi finne ut om fjerning av "potensielt overflødige delmål" gir en komplett og konsistent prosedyre. Se prosedyren CP_3^0 i [Bibel 87, IV.5].

Videre kan det være nyttig å studere hvilke reduksjoner som kan brukes i en forbindelsesprosedyre for ikke normaliserte utsagn på predikatlogisk form.

8.6 Alternative valg og tilbaketog

I utsagnslogikk er det tilstrekkelig å finne det sett av literaler som blokkerer alle veiene gjennom matrisen til utsagnet F.

For predikatlogikk er det i tillegg nødvendig å finne en mest generell unifikator σ_N som unifiserer de sett av komplementære literaler som blokkerer alle veiene gjennom k varianter av matrisen til F.

I utsagnslogikk er det derfor tilstrekkelig at alle veier gjennom matrisen til F blir undersøkt, inntil bevisprosedyren finner de blokkeringer som validerer F.

For predikatlogikk må en i tillegg søke etter en mest generell unifikator, σ_N.

Ved hver utvidelse vil et literalsett fra en klausul c bli valgt til å danne et komplementært sett av literaler sammen med et literal fra aktiv vei.

Anta at det er foretatt n utvidelser. Da får vi for alle n komplementære sett av literaler den mest generelle unifikator:

$$\sigma_n = \tau_n \circ \sigma_{n-1}$$

Ved "galt" valg av literalsett for utvidelse eller klausul for utvidelse vil σ_n ha minst en *substitusjonskomponent*, slik at seinere instansiering av unifikatoren hindrer blokkering av en vei q. Galt valg av startklausul kan også tilsvarende hindre blokkering av en vei.

Alternative valg av startklausul, klausul for utvidelse og literalsett for utvidelse sikrer mot muligheten for at instansiering av "gal" σ_n ødelegger et bevis.

Alternative valg av startklausuler gjør også at separasjonsskrittet kan sløyfes i prosedyrene CP_1^1 og CP_2^1. Se [Bibel 87, III.7.1.L].

Problemet med å redusere antall alternative valg, som må prøves, kan deles i to:

1. Bestemme så langt som det er mulig og slik at kompletthet beholdes hvilke literalsett og klausuler som det aldri er nødvendig å velge for utvidelse. Tilsvarende å bestemme hvilke startklausuler som ikke er nødvendig å velge.

   Denne form for avskjæring reduserer muligheten for å få en σ_n som ved seinere instansiering hindrer blokkering av en vei.

2. Bestemme så langt det er mulig og slik at kompletthet beholdes, de sett av literaler og klausuler som er slik at vilkårlig valg blant disse kan foretas uten at alternative valg er nødvendig.

8.6.1 Fjerning av alternative valg

Anta at vi legger inn separasjonsskrittet i en forbindelsesprosedyre for predikatlogikk. Da har vi en prosedyre som undersøker alle veiene gjennom kopiene til matrisen til F.

Anta at en vilkårlig σ_n, som bygges opp ved en sekvens av utvidelsesskritt, ikke ødelegger et bevis ved seinere instansiering.

Da vil vi ha en prosedyre som er komplett uten at vi behøver å prøve de alternative valg av startklausul, klausul for utvidelse og literalsett for utvidelse.

Dette leder til prosedyren CP^1_{RASK}, som presenteres i dette avsnittet. Separasjon må i en prosedyre for predikatlogikk legges inn med mulighet for tilbaketog.

Gitt at k kopier av matrisen til utsagnet F er generert. La og situasjonen være den at ingen nye komplementære forbindelser kan dannes mellom et literal på veien p og en klausul i D.

Da er det naturlig å forsøke separasjonsskrittet. Men det ikke er mulig å avgjøre om en klausul c kan velges for utvidelse fra en kopi som ennå ikke er generert. Derfor må vi her ha som et alternativt valg at ny kopi lages. Dette valget er det eneste valget mellom alternativer med tilbaketog i prosedyren CP^1_{RASK}.

Prosedyre 8.1 (CP^1_{RASK})

Variabelliste

$F._k$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
d:	*En klausul $d \in D$ som er kandidat for en utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei.*
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F._1 \cup \ldots \cup F._i \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med delmål for seinere prosessering.* *Hver innførsel på stakken består av (c,p,A)* *der* $c \neq \emptyset$.
L:	*Literal som er element av aktiv vei p.*
K:	*Et literal som er element av klausulen c.*
e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1:	*Literal som er element av p og som er komplementær til literaler i e.*

NEXTi,

NEXTi+1: *To stakker som gir situasjonen rett foran valg av en en ny klausul for utvidelse. Brukes ved tilbaketog for å prøve en ny kopi i stedet for separasjon. Hver innførsel består av (L,A,σ,p,WAIT).*

CP^1_{RASK}

SKRITT 0:
Om F ikke inneholder noen eksistenskvantorer så bruk CP^0_1.
comment *Initialisering.*
SKRITT 1:
$i := 1;$
$D := F_{.1};$
$\sigma := \varepsilon;$
$A := \emptyset;$
$p := \emptyset;$
$WAIT :=$ ***NIL***;
$\forall i$ $NEXTi :=$ ***NIL***;
comment *Valg av startklausul eller første klausul etter en separasjon.*
SKRITT 2:
select *en klausul* $c \in D;$
$D := D \backslash c;$
$A := A \cup \{c\};$
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L;$
if $c \neq \emptyset$
then
    $WAIT := Push(WAIT,(c,p,A))$
fi;
$p := p \cup \{L\};$
comment *Valg av klausul for utvidelse.*
SKRITT 4:
select *en klausul c fra D slik at* $\sigma(\{L, K\})$
    er en unifiserbar forbindelse for noen $K \in c$ *og noen* $L \in p;$
if not *en slik klausul finnes*
then
    comment *Forbered separasjon.*
    $k := i+1;$
    $NEXTk := Push(NEXTk,(L,A,\sigma,p,WAIT));$
    if $D \neq \emptyset$
    then
        comment *Separasjon.*
        $WAIT := NIL;$

```
            p := ∅;
            σ := ε;
            go to 2
        else
            go to 8
        fi
fi;
comment Valg av literalsett fra c for utvidelse.
SKRITT 5:
D := D\c;
A := A∪{c};
select et sett e som er delsett av c slik at
        σ({L^1} ∪ e) kan unifiseres
        med τ som en mest generell unifikator,
        og L^1 ∈ p;
comment Selve utvidelsen.
SKRITT 6:
c := c\e;
σ := τ ∘ σ;
if c ≠ ∅
then
        go to 3
fi
comment Sammentrekning.
SKRITT 7:
if WAIT = NIL
then
        Returner("Gyldig")
fi
(WAIT,(c,p,A)) := Pop(WAIT);
D := F.1 ∪ ... ∪ F.i \ A;
go to 3 ;
comment Tilbaketog og ny kopi.
SKRITT 8:
if NEXTi = NIL
then
        comment Lager ny kopi.
        i := i+1
fi
comment Tilbaketog der separasjon ikke prøves..
(NEXTi,(L,A,σ,p,WAIT)) := Pop(NEXTi);
D := F.1 ∪ ... ∪ F.i \ A;
go to 4

CP^1_RASK SLUTT
```

I [Bibel 87, III.6.9.L] er det vist med et eksempel at forbindelsesmetoden

generelt ikke er konfluent i predikatlogikk.

Matrisen

$$\begin{array}{cccc} P(x) & \neg P(a) & Q(a) & \neg Q(a) \\ \neg P(y) & & & \end{array}$$

er eksemplet på dette.

Ved valg av klausulen

$$\{P(x), \neg P(y)\}$$

som startklausul, vil en uendelig utledning kunne lages. Dette vil skje dersom nye kopier av matrisen til utsagnet blir laget uten at de eksisterende kopiene blir fullstendig undersøkt.

For eksemplet over vil et bevis kunne oppnåes med CP^1_{RASK}, også med valg av

$$\{P(x), \neg P(y)\}$$

som startklausul. Dette fordi første kopi undersøkes fullstendig ved at separasjon forsøkes. Her er ikke nødvendig med alternative valg av startklausul (utover det som implisitt ligger i separasjonsskrittet).

De alternative valgene, som fjernes i CP^1_{RASK}, er de alternative valgene vi kan fjerne uten helt å gi opp håpet om at bevisprosedyren fortsatt er komplett.

Separasjonsskrittet må være med for å sikre at hver kopi av matrisen til F blir fullstendig undersøkt.

Videre arbeid vil her være å finne et eksempel på at prosedyren CP^1_{RASK} *ikke* er komplett, eller eventuelt å forsøke å vise at prosedyren er komplett.

Det er også en mulighet for at en eller to av de alternative valgene kan fjernes med behold av kompletthet. De alternative valgene er fortsatt valg av startklausul, valg av klausul og literalsett for utvidelse. Også denne muligheten må undersøkes.

8.7 Utvidelse av k-kompakthetsbegrepet

8.7.1 Innledning

I dette avsnittet vil jeg definere k-konfluens og k-begrensning med utgangspunkt i k-kompakthetsbegrepet (Se [Nossum 84, s 15 og 16] eller avsnitt 4.3).

Men først ønsker jeg en mere presis beskrivelse av hva et logisk regnesystem er for noe. Følgende definisjon er hentet fra [Bibel 87, II.5.2]:

Definisjon 8.1 (Et logisk regnesystem) *Et logisk regnesystem (av gjenkjennende eller genererende type) består av*

- *Et sett $\mathcal{F}$, der elementene i settet blir kalt utsagn.*
- *Et sett $\Sigma(F)$ knyttet til $F \in \mathcal{F}$ gjennom funksjonen Σ, der elementene i $\Sigma(F)$ blir kalt strukturer.*
- *En binær utledningsrelasjon $\vdash$ på settet $\mathcal{E}$.*

  Settet $\mathcal{E}$ er sett av par $(F, \Sigma(F))$, der $F \in \mathcal{F}$.

Vi har videre et delsett $\mathcal{E}_0 \subseteq \mathcal{E}$ som vi kaller aksiomer, og et element $S_0 \in \Sigma(F)$ som vi kaller terminalstrukturen.

Dersom $(E_1, E_2) \in \ \vdash$ holder for vilkårlige $E_1, E_2 \in \mathcal{E}$, blir dette til vanlig uttrykt ved notasjonen $E_1 \vdash E_2$, og notasjonen blir kalt et utledningsskritt.

- *En sekvens $(E_1, \ldots, E_n)$ $(n \geq 1)$, slik at $E_i \in \mathcal{E}$, $i = 1, \ldots, n$, kalles en utledning av E_n fra E_1 og blir skrevet som $E_1 \vdash \ldots \vdash E_n$ eller som $E_1 \vdash^n E_n$. n kalles lengden av utledningen.*

  Dersom E er et aksiom $(E \in \mathcal{E}_0)$ og $F \in \mathcal{F}$, har vi:

  1. *$\{(F, S_0)\} \vdash^* E$ blir kalt et bevis i et gjenkjennende logisk regnesystem.*
  2. *$E \vdash^* \{(F, S_0)\}$ blir kalt et bevis i et genererende regnesystem.*

  Hvis det eksisterer en utledning for F der $F \in \mathcal{F}$, så sier vi at F kan utledes, og det skrives $\vdash F$.

Notasjonene fra definisjon 8.1 blir brukt i definisjonene 8.2 og 8.3.

Den videre framstilling vil ta sitt utgangspunkt i et regnesystem av generende type. Der begynner vi med et utsagn assosiert med en struktur (eller sett av strukturer), og forsøker å vise at terminalstrukturen kan utledes.

Definisjonene for k-konfluens og k-begrensethet kan lett utvides til å gjelde regnesystemer av gjenkjennende type.

I den videre framstilling, lar jeg et k-kompakt utsagn være et utsagn som valideres ved bruk av maksimalt k varianter av utsagnet.

Definisjonen gitt av [Nossum 84, s 15] lar et k-kompakt utsagn være et utsagn som vises utilfredstillbart ved bruk av maksimalt k varianter av utsagnet. Negasjonen av utsagnet blir k-kompakt etter definisjon gitt i avsnitt 4.3 i denne oppgaven.

8.7.2 k-konfluens

I dette avsnittet definerer jeg k-konfluens.

Definisjon 8.2 (k-konfluens) *Gitt et predikatlogisk regnesystem $\mathcal{L}$. Bruk variablene fra definisjonen av et logisk regnesystem og la*

$\mathcal{F}$:	*Settet av alle predikatlogiske utsagn på lukket og rettet form.*
$\mathcal{Z}^k$:	*Settet av alle k-kompakte utsagn på lukket og rettet form ($\mathcal{Z}^k \subseteq \mathcal{F}$).*
E, F:	*Utsagn som er element av $\mathcal{F}$.*
Z, V:	*Utsagn som er element av $\mathcal{Z}^k$.*
Z_0:	*Et sett av par, der hvert par er på formen (Z, I_0), for $I_0 \in \Sigma(Z)$ og $Z \in \mathcal{Z}^k$ (I_0 kalles også en initialstruktur).*
E_i, F_j:	*To sett av par. Hvert par i det første settet er på formen (E, U_i) for $U_i \in \Sigma(E)$ og $E \in \mathcal{E}$. Hvert par i det andre settet er på formen (F, U_j) for*

$U_j \in \Sigma(F)$ *og* $F \in \mathcal{F}$

V_n: *Et sett som består av et par på formen* $\{(V, S_0)\}$, *der* $S_0 \in \Sigma(V)$, *og der* $V \in \mathcal{Z}^k$. S_0 *er terminalstrukturen i regnesystemet.*

Anta at vi har et vilkårlig utsagn $Z \in \mathcal{Z}^k$, *og to vilkårlige utsagn* $E, F \in \mathcal{F}$, *og gitt to utledningssekvenser* $Z_0 \vdash^* E_i$ *og* $Z_0 \vdash^* F_j$. *Da kalles det logiske regnesystemet,* $\mathcal{L}$, k*-konfluent om vi har* $E_i \vdash^* V_n$ *og* $F_j \vdash^* V_n$.

Definisjonen av k-konfluens innebærer at det logiske regnesystemet skal komme fram til at et k-kompakt utsagn er gyldig uansett hvilken vei det velger.

$$\max\{k \mid \mathcal{L} \text{ er konfluent}\}$$

hadde vært fint å kunne beregne, men vi kjenner ikke noen måte å beregne den på.

8.7.3 k-begrensethet

I nær tilknytning til k-konfluens kan vi definere k-begrensethet.

Definisjon 8.3 (k-begrenset) *Anta at vi har et logisk regnesystem* $\mathcal{L}$.

Bruk de samme variable som i definisjonen for k*-konfluens og i definisjonen av et logisk regnesystem. I tillegg lar vi:*

F_n: *Et sett av par, der hvert par er på formen* (F, U_j), *og der* $U_j \in \Sigma(F)$ *og* $F \in \mathcal{F}$.

Anta at vi har en $Z \in \mathcal{Z}^k$.
Da er regnesystemet k*-begrenset om vi har*

$$\max\{n \mid Z_0 \vdash^n F_n \text{ for noen } F \in \mathcal{F}\} < \infty$$

Merk at denne definisjonen *ikke* forutsetter at det logiske regnesystemet er k-konfluent.

Men hvis vi har et k-konfluent regnesystem, så er det også k-begrenset.

For gitt en $Z_0 = \{(Z, I_0^1), \ldots, (Z, I_0^r)\}$, og $Z \in \mathcal{Z}$, så vil vi ved definisjonen av k-konfluens få en $V_n = \{(V, S_0)\}$, uansett hvilken vei regnesystemet tar.

Vi kan uten tap av generalitet forutsette at en utledning, som validiserer et utsagn, har et endelig antall skritt.

Ved å sette V_n for F_n får vi

$$\max\{n \mid Z_0 \vdash^n V_n\} < \infty$$

Begrensethetsklassen til $\mathcal{L}$ inneholder konfluensklassen. Begge hadde vært nyttige å kunne beregne, men ingen beregning er funnet her.

Referanser

[Bayerl 86] Bayerl, S. et. al. (1986) An implementation of a PROLOG-like theorem prover based on the connection method. (I: AIMSA 86. North-Holland)

[Bibel 87] Bibel, Wolfgang. (l987) Automated theorem proving. 2. rev. ed. Vieveg. Braunschweig, Wiesbaden.

[Chang, Lee 73] Chang, Chin-Liang og Lee, Richard Char-Tung. (1973) Symbolic logic and mechanical theorem proving. Academic press. New York.

[Corbin, Bidoit 83] Corbin, Jacques og Bidoit, Michel. (1983) A rehabilitation of Robinson's unification algorithm. (I: Information processing. 83 Ed R. E. A. Mason. Elsevier. Amsterdam, s. 909 - 914)

[Eder 85] Eder, Elmar. (1985) Properties of substitution and unification. (I: Journal of symbolic computation. Vol. 1, s. 31 - 46)

[Huet 86] Huet, Gerard (1986) Deduction and computation. (I: Fundamentals of artificial intelligence. Ed. W. Bibel and Ph. Jorrand. Springer. Berlin, s. 39 - 74)

[Jervell 86] Jervell, Herman Ruge. (l986) Forelesninger i logikk. Universitetet i Oslo. Oslo. *Upublisert manuskript.*

[Loveland 78] Loveland, Donald W. (l978) Automated theorem proving: A logical basis. North-Holland. Amsterdam.

[Martelli, Montanari 82] Martelli, Alberto og Montanari, Ugo. (l982) An efficient unification algorithm. (I: ACM Transactions on programming languages and systems. Vol. 4 No. 2, s. 258 - 282)

[Nossum 84] Nossum, Rolf. (1984) Decision algorithms for program verification. University of Oslo. Oslo.

[Nossum 85] Nossum, Rolf. (1985) Programmer's guide to the PROVER system — Thoralf a theorem prover. University of Oslo. Oslo.

[Nossum 85 II] Nossum, Rolf. (1985) Automated theorem proving methods. (I: BIT. Bind 25 s. 51 - 64)

[Paterson, Wegman 78] Paterson, M. S. og Wegman, M. N. (1978) Linear unification. (I: Journal of computer and system sciences. Vol. 16, s. 158 - 167)

[Siekmann 84] Siekmann, J. H. (1984) Universal unification. (I: 7th international conference on automated deduction. Ed. R. E. Shostak. Springer. Berlin, s. 1 - 42)

[Stickel 86] Stickel, Mark E. (1986) An introduction to automated deduction. (I: Fundamentals of artificial intelligence. Ed. W. Bibel and Ph. Jorrand. Springer. Berlin, s. 75 - 132)

Vedlegg A

Prosedyrer brukt i beviset for skillemerkemetoden

Prosedyren CP_A^1 følger samme søkestrategi som prosedyren CP_2^1. Ellers er den som CP_1^1.

Prosedyre A.1 (CP_A^1)

Variabelliste

$F._k$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
j:	*Indeks som brukes til å angi ordning av klausuler eller literalsett som skal lagres på ALT1,ALT2 eller ALT3.*
k:	*Antall klausuler på ALT2 som har komplementære forbindelser med sist valgte literal L på den aktive vei.*
g:	*Indeks for å angi en NEXTg stakk.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
d:	*En klausul $d \in D$ som er kandidat for en utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei.*
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F._1 \cup \ldots \cup F._i \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med delmål.*
bdm:	*Boolsk variabel.*

	false *dersom e fra c danner en komplementær forbindelse med sist valgte literal L på den aktive veien p.*
	true *dersom e fra c danner en komplementær forbindelse med et annet literal M på veien.*
L:	*Sist valgte literal på den aktiv vei p.*
M:	*Et literal på den aktive vei p som ikke er det sist valgte literal L på den aktive vei.*
K:	*Literal som er element av klausulen c.*
e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1, M^1*:*	*Literal som er element av p og som er komplementært til literaler i e.*
ALT1:	*Stakk med alternative startklausuler. Hver innførsel består av startklausulen (c).*
ALT2:	*Stakk med alternative klausuer for utvidelse. Hver innførsel består av (L,c,A,*σ*,p,bdm,WAIT)*
ALT3:	*Stakk med alternative literalsett for utvidelse. Hver innførsel består av (c,e,A,*σ*,*τ*,p,WAIT).*
NEXTi, *NEXTi+1:*	*To stakker som gir situasjonen rett foran valg av en en ny klausul for utvidelse. Hver innførsel består av (L,A,*σ*,p,WAIT).*

CP^1_A

comment *Samme søkestrategi som i* CP^1_2*, men ellers som* CP^1_1*.*
SKRITT 0:
Om F ikke inneholder noen eksistenskvantorer så bruk en prosedyre for utsagnslogikk analog med CP^1_A*.*
comment *Initialisering.*
SKRITT 1:
i := 1;
$D := F_{.1}$*;*
$\sigma := \varepsilon$*;*
$A := \emptyset$*;*
$p := \emptyset$*;*
bdm := ***true****;*
WAIT := ***NIL****;*
ALT1 := ***NIL****;*
ALT2 := ***NIL****;*
ALT3 := ***NIL****;*
$\forall i$ *NEXTi :=* ***NIL****;*
comment *Valg av startklausul.*
SKRITT 2:
select *en nummerering* $c_1, \ldots, c_m$ *for alle klausuler i D;*

for $j := m,\ldots,2$
do
    $ALT1 := Push(ALT1,(c_j))$;
od
$c := c_1$;
$D := D \setminus c$;
$A := \{c\}$;
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \setminus L$;
if $c \neq \emptyset$
then
        $WAIT := Push(WAIT,(c,p,A))$
fi;
$p := p \cup \{L\}$;
comment *Valg av klausul for utvidelse.*
SKRITT 4:
$g := i+1$;
$NEXTg := Push(NEXTg,(L,A,\sigma,p,WAIT))$;
select *en nummerering* $d_1,\ldots,d_k$ *av alle*
        klausuler fra D som er slik at $\sigma(\{L,K_j\})$
        er en komplementær og unifiserbar forbindelse for noen
        $K_j \in d_j$, $j = 1,\ldots,k$ *og L er sist valgte*
        literal på den aktive veien p;
select *en nummerering* $d_{k+1},\ldots,d_m$ *av alle*
        klausuler fra D som er slik at $\sigma(\{M_j,K_j\})$
        er en komplementær og unifiserbar forbindelse for noen
        $K_j \in d_j$, $j = k+1,\ldots,m$ *og noen*
        $M_j \in p \setminus \{L\}$;
if *m=0*
then
        go to *11*
fi;
if *k=0*
then
        $k_1 := 2$
else
        $k_1 := k+1$
fi
$bdm :=$ **true**;
for $j := m,\ldots,k_1$
do
    $ALT2 := Push(ALT2,(L,d_j,A,\sigma,p,bdm,WAIT))$;
od;
if $k > 0$
then

```
        bdm := false;
        for j := k, ..., 2
        do
            ALT2 := Push(ALT2,(L,d_j,A,σ,p,bdm,WAIT));
        od
fi;
c := d_1;
comment Valg av literalsett fra c for utvidelse.
SKRITT 5:
D := D\c;
A := A∪ {c};
if bdm = true
then
        For alle M ∈ p \ L,
        finn alle delsett e_j av c som er slik at
        σ({M^1} ∪ e_j) kan unifiseres
        med τ_j som en mest generell unifikator,
        der vi krever at τ_j ≠ τ_j*
        for j ≠ j*;
        select nummering e_1, ..., e_m av
                alle de delsett e_j av c som er funnet.
        for j := m, ..., 2
        do
            ALT3 := Push(ALT3,(c,e_j,A,σ,τ_j,p,WAIT))
        od
else
        select en nummerering e_1, ..., e_m av
                alle delsett av c som er slik at
                σ({L^1} ∪ e_j) kan unifiseres
                med τ_j som en mest generell unifikator,
                der vi krever at τ_j ≠ τ_j*
                for j ≠ j* og L^1 er sist valgte literal til veien p;
        for j := m, ..., 2
        do
            ALT3 := Push(ALT3,(c,e_j,A,σ,τ_j,p,WAIT))
        od
fi
comment Selve utvidelsen.
SKRITT 6:
c := c\e_1;
σ := τ_1 ∘ σ;
if c ≠ ∅
then
        go to 3
fi
comment Sammentrekning.
SKRITT 7:
```

```
if WAIT = NIL
then
        Returner("Gyldig")
fi
comment Gå tilbake til et delmål.
(WAIT,(c,p,A)) := Pop(WAIT);
D := F.1 ∪ ... ∪ F.i \ A;
go to 3
comment *** Skrittene 8,9 og 10 brukes ikke i denne prosedyren.
comment Alternativ utvidelse.
SKRITT 11:
if ALT3 ≠ NIL
then
        (ALT3,(c,e1, A,σ,τ1,p,WAIT)) := Pop(ALT3);
        D := F.1 ∪ ... ∪ F.i \ A;
        go to 6
fi
comment Alternativ klausul for utvidelse.
SKRITT 12:
if ALT2 ≠ NIL
then
        (ALT2,(L,c,A,σ,p,bdm,WAIT)) := Pop(ALT2);
        D := F.1 ∪ ... ∪ F.i \ A;
        go to 5
fi
comment Alternativ startklausul.
SKRITT 13:
if ALT1 ≠ NIL
then
        (ALT1,(c)) := Pop(ALT1);
        D := F.1 \ c;
        A := {c};
        σ := ε;
        p := ∅;
        go to 3
fi
comment Ny kopi og tilbaketog etter at ny kopi er laget.
SKRITT 14:
if NEXTi = NIL
then
        comment Lager ny kopi.
        i := i+1
fi
comment Tilbaketog etter at ny kopi er laget.
(NEXTi,(L,A,σ,p,WAIT)) := Pop(NEXTi);
D := F.1 ∪ ... ∪ F.i \ A;
go to 4
```

CP^1_A SLUTT

Prosedyren CP^1_B foretar ingen alternative valg med mulighet for tilbaketog. Den må derfor gjøre alle valg riktig. Ellers er den som CP^1_A.

Prosedyre A.2 (CP^1_B)

Variabelliste

$F._k$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
d:	*En klausul $d \in D$ som er kandidat for en utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei.*
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F._1 \cup \ldots \cup F._i \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med delmål.*
bdm:	*Boolsk variabel.* ***false*** *dersom e fra c danner en komplementær forbindelse med sist valgte literal L på den aktive veien p.* ***true*** *dersom e fra c danner en komplementær forbindelse med et annet literal M på veien.*
L:	*Sist valgte literal på den aktiv vei p.*
M:	*Et literal på den aktive vei p som ikke er det sist valgte literal L på den aktive vei.*
K:	*Literal som er element av klausulen c.*
e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1, M^1:	*Literal som er element av p og som er komplementært til literaler i e.*

CP^1_B

comment *Ikke alternative valg, ellers som* CP^1_A
SKRITT 0:
Om F ikke inneholder noen eksistenskvantorer
så bruk en prosedyre for utsagnslogikk analog til CP^1_B.
comment *Initialisering.*
SKRITT 1:
$i := 1;$
$D := F_{\cdot 1};$
$\sigma := \varepsilon;$
$A := \emptyset;$
$p := \emptyset;$
$SC := \emptyset;$
bdm := ***true***;
WAIT := ***NIL***;
comment *Valg av startklausul.*
SKRITT 2:
select *en startklausul* $c \in D$
$D := D \backslash c;$
$A := \{c\};$
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L;$
if $c \neq \emptyset$
then
    WAIT := *Push(WAIT,(c,p,A))*
fi;
$p := p \cup \{L\};$
comment *Valg av klausul for utvidelse.*
SKRITT 4:
if *klausul skal hentes fra ny kopi*
then
    comment *Lag en ny kopi.*
    $i := i+1;$
    $D := D \cup F_{\cdot i}$
fi
if *en klausul* $c \in D$ *finnes, slik at* $\sigma(\{L, K\})$
  er en komplementær og unifiserbar forbindelse for noen
  $K \in c$ *og L er sist valgte literal på den aktive veien p*
then
    select *en slik klausul* $c \in D;$
    bdm := ***false***
else
    if *en klausul* $c \in D$ *finnes slik at* $\sigma(\{M, K\})$

er en komplementær og unifiserbar forbindelse for noen
$K \in c$ og noen $M \in p \setminus \{L\}$
then
select *en slik klausul $c \in D$;*
bdm := ***true***
else
Returner("Tilbaketog ikke mulig")
fi
fi;
comment *Valg av literalsett fra c for utvidelse.*
SKRITT 5:
$D := D \setminus c$;
$A := A \cup \{c\}$;
if *bdm =* ***true***
then
select *et sett e som er delsett av c slik at*
$\sigma(\{M^1\} \cup e)$ er unifiserbar med
en mgu τ, og $M \in p \setminus \{L\}$
else
select *et sett e som er delsett av c slik at*
$\sigma(\{L^1\} \cup e)$ er unifiserbar med
en mgu τ, og L er sist valgte literalen på veien p
fi
comment *Selve utvidelsen.*
SKRITT 6:
$c := c \setminus e$;
$\sigma := \tau_1 \circ \sigma$;
if *$c \neq \emptyset$*
then
go to *3*
fi
comment *Sammentrekning.*
SKRITT 7:
if *WAIT =* ***NIL***
then
Returner("Gyldig")
fi
comment *Gå tilbake til et delmål.*
(WAIT,(c,p,A)) := Pop(WAIT);
$D := F_{.1} \cup \ldots \cup F_{.i} \setminus A$;
go to *3*
comment *SKRITT 8,9,10,11, 12 ,13 og 14 brukes ikke her.*

CP_B^1 SLUTT

Prosedyren CP_C^1 foretar ingen alternative valg. Videre fjerner den visse delmål på samme måte som CP_2^0. Også denne prosedyren må gjøre alle valg

riktig.

Prosedyre A.3 (CP_C^1)

Variabelliste

$F._k$:	*Kopi k av matrisen til utsagnet F.*
i:	*Indeksen til den siste kopi som er laget. Den forteller hvor mange kopier som er laget.*
c:	*Den klausul som til enhver tid behandles av prosedyren. Startklausul eller klausul valgt for utvidelse.*
d:	*En klausul* $d \in D$ *som er kandidat for en utvidelse.*
p:	*Det sett av literaler som til enhver tid utgjør den aktive vei.*
σ:	*Mest generell unifikator for familien av de sett av av literaler som danner de komplementære forbindelsene som (eventuelt) blokkerer veiene gjennom kopiene av matrisen til F.*
τ:	*Unifikator for et sett av literaler som danner en komplementær forbindelse.*
A:	*Settet av klausuler som* *- det går en aktiv vei gjennom, eller* *- er siste klausul valgt for utvidelse, eller* *- tilhører settet av klausuler som er ferdigbehandlet etter separasjon.*
D:	$D = F._1 \cup \ldots \cup F._i \setminus A$ *Det sett av klausuler som kan velges for utvidelser.*
WAIT:	*Stakk med:* *- delmål* *- skillemerker* *- sett av forbindelser*
bdm:	*Boolsk variabel.* ***false*** *dersom e fra c danner en komplementær forbindelse med sist valgte literal L på den aktive veien p.* ***true*** *dersom e fra c danner en komplementær forbindelse med et annet literal M på veien.*
merke:	*Forteller hva slags innførsel som ligger på stakken WAIT.* *- 'sg' = delmål* *- 'sc' = komplementær forbindelse med literal fra aktiv vei som ikke er det sist valgte literal på aktiv vei.* *- 'dm' = Et skillemerke. Blir lagt på stakken når det ikke finnes noen klausul for utvidelse med et komplementært literal til sist valgte literal på aktiv vei.*
L:	*Sist valgte literal på den aktiv vei p.*
M:	*Et literal på den aktive vei p som ikke er det sist valgte literal L på den aktive vei.*
ind1:	*Indeks som peker på det sist valgte element L på den aktive vei.*
ind2:	*Indeks som peker på literalet M på den aktive vei.*
K:	*Literal som er element av klausulen c.*

e:	*Sett av komplementære literaler fra klausulen c, som er valgt for utvidelse.*
L^1, M^1:	*Literal som er element av p og som er komplementært til literaler i e.*
SC:	*Sett av forbindelser lest fra stakken WAIT ved sammentrekning.*

CP_C^1

comment *Fjerner delmål, ellers som* CP_B^1.
Ikke alternative valg, ellers som CP_2^1
SKRITT 0:
Om F ikke inneholder noen eksistenskvantorer
så bruk en prosedyre for utsagnslogikk analog til CP_C^1.
comment *Initialisering.*
SKRITT 1:
$i := 1;$
$D := F_{\cdot 1};$
$\sigma := \varepsilon;$
$A := \emptyset;$
$p := \emptyset;$
$SC := \emptyset;$
$ind1 := 0;$
$bdm :=$ ***true****;*
$WAIT :=$ ***NIL****;*
comment *Valg av startklausul.*
SKRITT 2:
select *en startklausul* $c \in D$
$D := D \backslash c;$
$A := \{c\};$
comment *Valg av et delmål.*
SKRITT 3:
select *et literal L fra klausulen c;*
$c := c \backslash L;$
if $c \neq \emptyset$
then
    $WAIT := Push(WAIT,('sg',ind1,(c,p,A)))$
fi*;*
$ind1 := ind1 + 1;$
$p := p \cup \{(L, ind1)\};$
comment *Valg av klausul for utvidelse.*
SKRITT 4:
if *klausul skal hentes fra ny kopi*
then
    $i := i+1;$
    $D := D \cup F_{\cdot i}$

fi
if *en klausul* $c \in D$ *finnes, slik at* $\sigma(\{L, K\})$
er en komplementær og unifiserbar forbindelse for noen
$K \in c$ *og* L *er sist valgte literal på den aktive veien* p
then
select *en slik klausul* $c \in D$;
$bdm :=$ **false**
else
if *en klausul* $c \in D$ *finnes slik at* $\sigma(\{M, K\})$
er en komplementær og unifiserbar forbindelse for noen
$K \in c$ *og noen* $(M, ind2) \in p \setminus \{(L, ind1)\}$
then
select *en slik klausul* $c \in D$;
$bdm :=$ **true**
else
Returner("Tilbaketog ikke mulig")
fi
fi;
comment *Valg av literalsett fra c for utvidelse.*
SKRITT 5:
$D := D \setminus c$;
$A := A \cup \{c\}$;
if $bdm =$ **true**
then
WAIT := Push(WAIT,('dm',ind1,NIL));
select *et sett e som er delsett av c slik at*
$\sigma(\{M^1\} \cup e)$ *er unifiserbar med*
en mgu τ, *og* $(M, ind2) \in p \setminus \{(L, ind1)\}$
else
select *et sett e som er delsett av c slik at*
$\sigma(\{L^1\} \cup e)$ *er unifiserbar med*
en mgu τ, *og* L *er sist valgte literalen på veien* p
fi
comment *Selve utvidelsen.*
SKRITT 6:
$c := c \setminus e$;
$\sigma := \tau_1 \circ \sigma$;
if $bdm =$ **true**
then
WAIT := Push(WAIT,('sc',ind1,{ind2}))
fi
if $c \neq \emptyset$
then
go to *3*
fi
comment *Sammentrekning.*
SKRITT 7:

```
if WAIT = NIL
then
        Returner("Gyldig")
fi
comment Gå tilbake til et delmål.
SKRITT 8:
if merket på toppen av WAIT er 'sg'
then
        (WAIT,(merke,ind1,(c,p,A))) := Pop(WAIT);
        D := F.1 ∪ ... ∪ F.i \ A;
        go to 3
fi;
comment Lagre en forbindelse på SC.
SKRITT 9:
if merket på toppen av WAIT er 'sc'
then
        (WAIT,(merke,ind1,SC')) := Pop(WAIT);
        SC := SC ∪ SC';
        go to 7
fi;
comment Hoppe over noen delmål.
SKRITT 10:
if merket på toppen av WAIT er 'dm'
then
        SC := SC \ {ind2 | ind2 >indeks på toppen av WAIT} ;
        while indeksen på toppen av WAIT ≥ maksimum av SC
        do
            (WAIT,(merke,indeks,dummy)) := Pop(WAIT)
        od;
        go to 7
fi;
comment SKRITT 11, 12 ,13 og 14 brukes ikke her.

CP^1_C SLUTT
```

Vedlegg B

Robinson's unifikasjonsalgoritme

I dette vedlegget gjengis en implementasjon i Common LISP av Robinson's unifikasjonsalgoritme. Programmet ble kjørt på UiO's DEC-2065 system. Programmet er skrevet av Marit Holden og den er med i denne rapporten med hennes velvillige tillatelse.

```
;;; Output: - om ikke unifiserbare: (())
;;;         - om unifiserbare: mgu for atom 1 og atom 2
;;; Representasjon av atomer, termer og unifikatorer.
;;; P(t1,...,tn) representeres som (P t1 ... tn)
;;; f(t1,...,tn) representeres som (f t1 ... tn)
;;; f()          representeres som (f NIL)  (eventuelt som f)
;;; x            representeres som (x)
;;; (t1/x1,...,tm/xm) representeres som
;;;                     ((t1 (x1)) ... (tm (xm)))
;;;
;;; Forfatter: Marit Holden



(DEFUN finnmgu-robins (atom1 atom2)
      (LET ((sigma '()))
      (DO* ((symbol1 (CAR atom1) (CAR atom1)) ;init
            (symbol2 (CAR atom2) (CAR atom2))
            (atom1 (CDR atom1) (CDR atom1))
            (atom2 (CDR atom2) (CDR atom2)))
       ((OR (NULL symbol1) (EQUAL '(()) sigma)) sigma) ;test
       (IF (NOT (EQUAL symbol1 symbol2)) ;kropp
           (IF (ATOM symbol1)
               (IF (ATOM symbol2)        ;symbol1 = f
                   (SETQ sigma '(()))    ;symbol2 = g
                   (IF (= 1 (LENGTH symbol2))
                       (settinn symbol2 symbol1) ;symbol2 =
(x)
```

```
                    (SETQ sigma '((((((      ;symbol2 =
(g t1 ... tm)
               (IF (= 1 (LENGTH symbol1))
                   (settinn symbol1 symbol2)     ;symbol1 =
(x)
                   (IF (ATOM symbol2)            ;symbol2 =
(f t1 ... tm)
                       (SETQ sigma '((((        ;symbol2 = g
                       (IF (= 1 (LENGTH symbol2))
                           (settinn symbol2 symbol1) ;symbol2
= (x)
                           (PROGN (SETQ nysigma
(finnmgu-robins symbol1 symbol2)) ;symbol2 = (g d1 ... dm)
                                  (IF (EQUAL nysigma '((((
                                      (SETQ sigma '((((
                                      (PROGN (SETQ sigma
(settsammen nysigma sigma))
                                             (SETQ atom1
(anvendsigma nysigma atom1))
                                             (SETQ atom2
(anvendsigma nysigma atom2))))))))))))))

(DEFMACRO settinn (x term)
          `(IF (x-i-t ,x ,term)
               (SETQ sigma '((((
               (PROGN (SETQ atom1 (SUBST ,term ,x atom1
:test #'equal))
                      (SETQ atom2 (SUBST ,term ,x atom2
:test #'equal))
                      (SETQ sigma (SUBST ,term ,x sigma
:test #'equal))
                      (SETQ sigma (CONS (LIST ,term ,x)
sigma)))))

(DEFUN x-i-t (x term)
       (IF (EQUAL x term) T
           (COND ((ATOM term) NIL)
                 ((EQUAL 1 (LENGTH term)) NIL)
                 (T (DO ((elem (CAR term) (CAR term))
                         (term (CDR term) (CDR term)))
                        ((OR (NULL elem) (x-i-t x elem))
                         (NOT (NULL elem))))))))

(DEFUN settsammen (nysigma sigma)
       (DO ((k (CAR nysigma) (CAR restsigma))
            (restsigma (CDR nysigma) (CDR restsigma)))
           ((NULL k) (SETQ sigma (APPEND nysigma sigma)))
           (SETQ sigma (SUBST (CAR k) (CADR k) sigma :test
```

```
#'equal))))

(DEFUN anvendsigma (nysigma atomi)
       (DO ((k (CAR nysigma) (CAR restsigma))
            (restsigma (CDR nysigma) (CDR restsigma)))
           ((NULL k) atomi)
           (SETQ atomi (SUBST (CAR k) (CADR k) atomi :test
#'equal))))
```

Vedlegg C

Robinson's modifiserte unifikasjonsalgoritme

I dette vedlegget gengis en implementasjon i Common LISP av Robinson's unifikasjonsalgoritme, modifisert for en ERAG graf og der alle variable er distinkte (Corbin's og Bidoit's unifikasjonsalgoritme). Programmet ble kjørt på UiO's DEC2065 system.

Arbeidet med dette programmet er formelt *ikke* del av arbeidet med hovedoppgaven, fordi programmet er forfatterens del av en obligatorisk prosjektoppgave til det avanserte emnet "Programmering i LISP".

```
;;;; FINNMGU-CORBID
;;;  Denne versjon returnerer ei liste over substituerte
;;;  variable. Skal brukes i bevisf|ringsprogram som
;;;   skal ha mgu paa ERAG-struktur.
;;;  INN: To atomer.
;;;  RETURVERDI: Om ikke unifiserbar: (())
;;;              Om unifiserbar: mgu for atom1 og atom2.
;;;  REPRESENTASJON av atomer, termer og unifikatorer.
;;;              P(t1,...,tn)       representeres som
;;;                                 (P t1 ... tn)
;;;              f(t1,...,tn)       representeres som
;;;                                 (f t1 ... tn)
;;;              f()                representeres som (f NIL)
;;;                                 (Inndata kan ha konstanter
;;;                                 paa formen f)
;;;              x                  representeres som (x)
;;;              (t1/x1,...,tm/xm)  representeres som ei liste
;;;                                 av subsituerte
;;;                                 variable. Disse har pekere
;;;                                 til den noden
;;;                                 i ERAG-grafen som
;;;                                 representerer termene
;;;                                 som substituerer
;;;                                 variabelen.
;;;
```

```
;;; FORFATTER: Dag Diesen
;;;
;;; INTERN DATASTRUKTUR:
;;;
;;;    1) Atomene danner en Endelig Rettet Asyklisk Graf,
;;;       der alle variablene er distinkte.
;;;    2) Alle distinkte variable er samlet i en
;;;       variabelliste. Etter at termene er unifisert,
;;;       saa er de substituerte variablene merket og
;;;       har peker til den termen som substituerer den.
;;;       De substituerte variablene representerer tilsammen
;;;       den mest generelle unifikator for de unifiserte
;;;       atomene.
;;;
;;;  Hver node i grafen er et symbol, og pekere til andre
;;;  noder finnes i symbolets property-list. Hver node
;;;  som representerer et atom eller funksjon er et
;;;  maskingenerert symbol. Hver node som representerer
;;;  en variabel er det symbol som baerer variabelens navn.
;;;
;;;  TERMENE har dette innhold i sine property-lister:
;;;
;;;          Et ATOM:
;;;             Label:  Navnet paa atomet.
;;;             Barn:   Tabell med pekere til de nodene som
;;;                     representerer termene i atomet.
;;;             Merke:  Ikke negativt heltall.
;;;             Substi: NIL
;;;             Fedre:  NIL
;;;
;;;          En FUNKSJON:
;;;             Label:  Navnet paa funksjonen.
;;;             Barn:   Tabell med pekere til de nodene
;;;                      som representerer subtermene
;;;                      i funksjonen.
;;;             Merke:  Ikke negativt heltall.
;;;             Substi: Liste over de variable som denne
;;;                     funksjonen substituerer.
;;;             Fedre:  Liste av tilbakepekere. Hver peker
;;;                     er et par som bestaar av navnet
;;;                     paa farsnoden, og   d e n n e
;;;                     nodens nummer i s|skenflokken.
;;;
;;;          En VARIABEL:
;;;             Label:  NIL
;;;             Barn:   Tabell med null elementer.
;;;             Merke:  Ikke negativt heltall.
;;;             Substi: 1) Hvis variabelen er substituert skal
```

```
;;;                        det i dette feltet vaere et
;;;                        s y m b o l   som peker til den
;;;                        termen som erstatter variabelen
;;;                        (Feltet oppdateres ved seinere
;;;                        substitusjoner).
;;;                     2) Hvis variabelen ikke er substituert
;;;                        skal det i dette feltet vaere ei
;;;                        l i s t e   av variable som
;;;                        denne noden substituerer
;;;                        (tilbakepekere).
;;;             Fedre:  Liste av tilbakepekere. Hver peker er
;;;                     et par som bestaar av navnet paa
;;;                     farsnoden, og variabelens plass i
;;;                     s|skenflokken.
;;;
;;;
(DEFUN finnmgu-corbid (atom1 atom2)
       (LET ((tdag NIL)
             (node1 NIL)
             (node2 NIL)
             (varliste NIL)
             (sigma NIL)
             (mgu-ind 0))
;;;Danner den Endelig Rettet Asykliske Grafen.
       (SETQ tdag (erag-atom atom1 varliste))
       (SETQ node1 (CAR tdag))
       (SETQ varliste (CADR tdag))
       (SETQ tdag (erag-atom atom2 varliste))
       (SETQ node2 (CAR tdag))
       (SETQ varliste (CADR tdag))
       (SETQ atom1 NIL)
       (SETQ atom2 NIL)
;;;Unifiserer.
       (SETQ mgu-ind (unify node1 node2 mgu-ind))
(IF (= mgu-ind -1)
    (SETQ subst-varliste '(()) )
    (SETQ subst-varliste (substituerte_variable varliste))) ))

;;;; ERAG-ATOM
;;;  Omformer listen av de termene som skal unifiseres til en
;;;  endelig rettet asyklisk graf (ERAG). I denne grafen er
;;;  variablene distinkte og kan derfor ha flere pekere til
;;;   seg. Ei liste over alle distinkte  variable lages ogsaa.
;;;
;;;  INN: Liste som representerer et av atomene som skal
;;;       unifiseres. Variabelliste.
;;;  RETURVERDI:  Liste som bestaar av toppnoden i grafen
;;;               og oppdatert variabelliste.
```

```
;;;  SIDEVIRKNING: Dannelse av en rettet asyklisk graf der
;;;                alle variable er distinkte.

(DEFUN erag-atom (atomet varliste)
    (LET ((pnavn NIL)
          (*node*))
    (DECLARE (SPECIAL *node*))
;;; lager de to f|rste nodene i nettet. Kaller erag-term som
;;; lager resten av nettet.
       (SETQ pnavn (CAR atomet))
       (SETQ *node* (GENTEMP))
       (SETF (GET *node* 'label) pnavn)
       (SETF (GET *node* 'merke) 0)
       (SETF (GET *node* 'substi) NIL)
       (SETF (GET *node* 'fedre) NIL)
       (SETQ atomet (CDR atomet))
       (SETQ varliste (erag-term atomet *node* varliste))
       (LIST *node* varliste)))

;;;; ERAG-TERM.
;;;  INN:            Liste av subtermer fra den funksjonen som
;;;                  danner farsnoden.
;;;                  Farsnoden som subtermen skal knyttes til.
;;;                  Liste med alle distinkte variable funnet
;;;                  saa langt.
;;;  SIDEVIRKNINGER: Bygger opp nettet for alle subtermene og
;;;                  knytter frbindelsene med farsnoden.
;;;  RETURVERDI:     Oppdatert variabelliste.

(DEFUN erag-term (termi fars-node varliste)
      (LET ((*node* NIL)
            (label NIL)
            (barn-nr -1)
            (brorsliste NIL))
       (DECLARE (SPECIAL *node*))
       (DO* ((symbol (CAR termi) (CAR termi))
             (termi (CDR termi) (CDR termi)))
       ((NULL symbol) varliste brorsliste barn-nr)
       (SETQ barn-nr (+ barn-nr 1))
       (IF (ATOM symbol)
           (PROGN (SETQ *node* (GENTEMP))
                  (lag-f-node *node* symbol fars-node barn-nr)
                  (SETQ brorsliste (CONS *node* brorsliste))
                  (ingen-barn *node*))
           (IF (= 1 (LENGTH symbol))
               (PROGN (SETQ label (CAR symbol))
                      (SETQ varliste (oppdat-var label
fars-node varliste barn-nr))
```

```
                    (SETQ brorsliste (CONS label
brorsliste))
                    (ingen-barn label))
               (PROGN (SETQ label (CAR symbol))
                    (SETQ *node* (GENTEMP))
                    (lag-f-node *node* label fars-node
barn-nr)
                    (SETQ brorsliste (CONS *node*
brorsliste))
                    (SETQ symbol (CDR symbol))
                    (SETQ varliste (erag-term symbol *node*
varliste)) ))))
       (SETQ brorsliste (REVERSE brorsliste))
       (SETQ barn-nr (+ barn-nr 1))
       (sett-inn-broedre fars-node brorsliste barn-nr)
       varliste))

;;;; LAG-F-NODE
;;;  INN: node, funksjonsnavn, farsnode, nodens nummer
;;;       i s|skenflokken.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Lager en funksjonsnode ved aa legge inn
;;;                funksjonsnavn og tilbakepeker til
;;;                farsnoden i property-list.

(DEFUN lag-f-node (node fnavn far barn-nr)
      (SETF (GET node 'label) fnavn)
      (SETF (GET node 'fedre) (LIST (LIST far barn-nr)))
      (SETF (GET node 'merke) 0)
      (SETF (GET node 'substi) NIL))

;;;; LAG-V-NODE
;;;  INN: variabelnavn, farsnode, variabelliste, variabelens
;;;        nummer i s|skenflokkken.
;;;  RETURVERDI: Oppdatert variabelliste
;;;  SIDEVIRKNING: Lager en variabelnode ved aa legge inn
;;;                tilbakepeker til farsnoden property-list.

(DEFUN lag-v-node (var-navn far varliste barn-nr)
      (SETF (GET var-navn 'fedre) (LIST (LIST far barn-nr)))
      (SETF (GET var-navn 'merke) 0)
      (SETF (GET var-navn 'substi) NIL)
      (SETF (GET var-navn 'label) NIL)
      (CONS var-navn varliste))

;;;; V-SOEK
;;;  INN: variabelnavn, variabelliste.
;;;  RETURVERDI: T om variabelen finnes, NIL ellers.
```

```
;;;  SIDEVIRKNING: -

(DEFUN v-soek (var-navn varliste)
      (LET ((funnet NIL))
      (DO* ((var (CAR varliste) (CAR varliste))
            (varliste (CDR varliste) (CDR varliste)))
           ((OR (NULL var) funnet) funnet)
           (IF (EQL var var-navn)
               (SETQ funnet 'T))) ))

;;;; NY-FAR-V
;;;  INN: variabelnavn, farsnode, variabelens plass i
;;;       s|skenflokken.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Legger inn ny tilbakepeker til farsnoden.

(DEFUN ny-far-v (varnavn far barne-nr)
      (LET ((f-liste NIL))
      (SETQ f-liste (GET varnavn 'fedre))
      (SETF (GET varnavn 'fedre) (CONS (LIST far barne-nr)
f-liste)) ))

;;;; OPPDAT-VAR
;;;  INN: variabelnavn, farsnode, variabelliste, variabelens
;;;       plass i s|skenflokken.
;;;  RETURVERDI: Oppdatert variabelliste.
;;;  SIDEVIRKNING: Hvis variabelen er ny lages en
;;;                variabelnode og variabellista oppdateres,
;;;                ellers faar variabelen tilbakepeker til
;;;                ny far.

(DEFUN oppdat-var (varnavn far varliste barn-nr)
      (IF (v-soek varnavn varliste)
          (ny-far-v varnavn far barn-nr)
          (SETQ varliste (lag-v-node varnavn far varliste
barn-nr)))
      varliste)

;;;; SETT-INN-BROEDRE
;;;  INN: farsnode, liste over br|dre, antall br|dre.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Lager en tabell og plasserer br|drene i
;;;                den. Setter inn tabellen som  barn i
;;;                farsnodens property-list.

(DEFUN sett-inn-broedre (far brorsliste ant-barn)
      (LET (( brors-tab))
      (SETQ brors-tab (MAKE-ARRAY ant-barn :INITIAL-CONTENTS
```

```
brorsliste))
      (SETF (GET far 'barn) brors-tab) ))

;;;; INGEN-BARN
;;;  INN: node til en variabel eller konstant.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Setter inn tom tabell i property-list
;;;                til noden.

(DEFUN ingen-barn (node)
      (LET ((brors-tab))
      (SETQ brors-tab (MAKE-ARRAY 0))
      (SETF (GET node 'barn) brors-tab) ))

;;;; UNIFY
;;;  Pr|ver aa finne den mest generelle unifikator av to
;;;  termer. Termene maa vaere representert som en endelig
;;;  asyklisk graf der alle variable er distinkte.
;;;  Liste over distinkte variable maa eksistere.
;;;  INN: node1, node2, markeringsteller.
;;;  RETURVERDI: -1 om termene ikke kan unifiseres. Et ikke
;;;              negativt heltall om en mest generell
;;;              unifikator eksisterer.
;;;  SIDEVIRKNING: Grafen vil representere de unifiserte
;;;                termene, hvis termene kan unifiseres.
;;;                De substituerte variablene i variabellista
;;;                vil ha en oppdatert peker til den termen
;;;                som substituerer variabelen. Variable som
;;;                i k k e   er substituert kan skilles fra
;;;                de substituerte variablene.
;;;
;;;  Se artikkelen: Corbin, J & Bidoit, M : A rehabilitation
;;;                 of Robinson's unification algorithm.
;;;                 (Information processing 83) s 909 - 914.

(DEFUN unify (node1 node2 m-teller)
      (LET (( termer NIL)
            (alle-barn1)
            (alle-barn2)
            (ant-barn)
            (ind 0)
            (t-node1)
            (t-node2))
      (SETQ termer (finn-variabel node1 node2))
      (IF (NOT (NULL termer))
          (PROGN (SETQ node1 (CAR termer))
                 (SETQ node2 (CADR termer))
                 (SETQ m-teller (+ m-teller 1))
```

```
                (IF (forekommer node1 node2 m-teller)
                    (SETQ m-teller -1)
                    (erstatt node1 node2)))
          (PROGN (IF (ulike-termer node1 node2)
                     (SETQ m-teller -1)
                     (PROGN (SETQ alle-barn1 (GET node1
'barn))
                            (SETQ alle-barn2 (GET node2
'barn))
                            (SETQ ant-barn (ARRAY-DIMENSION
alle-barn1 0))
                            (DO* ((ind 0 (+ ind 1)))
                                 ((OR (= ant-barn ind)
(= m-teller -1)) m-teller)
                                 (SETQ t-node1 (AREF
alle-barn1 ind))
                                 (SETQ t-node2 (AREF
alle-barn2 ind))
                                 (IF (NOT (EQL t-node1
t-node2))
                                     (SETQ m-teller (unify
t-node1 t-node2 m-teller))))
                     (IF (NOT (= m-teller -1))
                         (erstatt node1 node2)) ))))
      m-teller))

;;;; FOREKOMMER
;;;  Unders|ker om en oppgitt variabel forekommer i en term.
;;;  INN: variabel, term, markeringsteller.
;;;  RETURVERDI: T om variabelen forekommer i termen,
;;;              NIL ellers.
;;;  SIDEVIRKNING: Merker de gjennoms|kte nodene med verdien
;;;                til markerings telleren.

(DEFUN forekommer (var term m-teller)
      (LET ((alle-barn NIL)
            (var-funnet NIL)
            (ant-barn)
            (subterm))
      (IF (EQL var term)
          (SETQ var-funnet 'T)
          (PROGN (SETF (GET term 'merke) m-teller)
                 (SETQ alle-barn (GET term 'barn))
                 (SETQ ant-barn (ARRAY-DIMENSION alle-barn 0))
                 (DO* (( ind 0 (+ ind 1)))
                      ((OR (= ant-barn ind) var-funnet)
var-funnet)
                      (SETQ subterm (AREF alle-barn ind))
```

```
                    (IF (NOT (= m-teller (GET subterm
'merke)))
                        (SETQ var-funnet (forekommer var
subterm m-teller))))))))

;;;; FINN-VARIABEL
;;;  Finner ut om en av nodene representerer en variabel.
;;;  INN: node1, node2
;;;  RETURVERDI: NIL, hvis ingen av nodene representerer
;;;              en variabel. Ei liste av de to nodene der
;;;              f|rste noden representerer en variabel,
;;;              hvis en variabel finnes.
;;;  SIDEVIRKNING: -

(DEFUN finn-variabel (node1 node2)
      (LET ((retur NIL)
            (f-navn NIL))
      (SETQ f-navn (GET node1 'label))
      (IF (EQL f-navn NIL)
          (LIST node1 node2)
          (PROGN (SETQ f-navn (GET node2 'label))
                 (IF (EQL f-navn NIL)
                     (LIST node2 node1)
                     NIL)))))

;;;; ULIKE-TERMER
;;;  Unders|ker om nodene representerer samme funksjon.
;;;  INN: node1, node2
;;;  RETURVERDI: T om de to nodene representerer ulike
;;;              funksjoner, NIL om termene representerer
;;;              samme funksjon.
;;;  SIDEVIRKNING: -

(DEFUN ulike-termer (node1 node2)
      (LET ((f-navn1 NIL)
            (f-navn2 NIL))
      (SETQ f-navn1 (GET node1 'label))
      (setq f-navn2 (GET node2 'label))
      (IF (EQL f-navn1 f-navn2)
          NIL
          T) ))

;;;; ERSTATT
;;;  Erstatter node1 med node2. Hvis node1 representerer
;;;  en variabel, vil den bli markert som substituert.
;;;  Variabelen vil faa en peker til den termen som
;;;  substituerer den.
;;;  INN: node1, node2
```

```
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Node1 sine fedre faar node2 som barn i
;;;                stedet for node1.
;;;                Node2 faar lagt til node1 sine fedre.
;;;                Node1 sine (eventuelle) barn faar fjernet
;;;                node1 som far.
;;;                Variable som node1 substituerer blir satt
;;;                til aa peke paa node2.
;;;                Node2 faar lagt til de substituerte
;;;                variable til node1 paa sin
;;;                substitusjonsliste.
;;;                Hvis node1 er en variabel blir den satt
;;;                til aa peke paa paa node2. Node2 sin
;;;                substitusjonsliste faar lagt til node1.

(DEFUN erstatt (node1 node2)
      (LET ((farsliste1 NIL)
            (farsliste2 NIL)
            (substliste1 NIL)
            (substliste2 NIL)
            (f-navn NIL))
;;; Farsnodene til node1 settes til aa peke paa node2.
      (SETQ farsliste1 (GET node1 'fedre))
      (bytt-ut-barn node2 farsliste1)
;;; Node1 sine tilbakepekere til fedre i grafen flyttes
;;; til node2.
      (SETQ farsliste2 (GET node2 'fedre))
      (SETF (GET node2 'fedre) (APPEND farsliste1 farsliste2))
;;; Barna til node1 skal ikke lenger ha node1 som far.
      (fjern-som-far node1)
;;; De variablene som tidligere er erstattet av node1
;;; faar sine substitusjonspekere til aa peke paa node 2.
      (SETQ substliste1 (GET node1 'substi))
      (bytt-ut-substituttene node2 substliste1)
;;; Node1 sine tilbakepekere til tidligere substituerte
;;; variable flyttes til node2.
      (SETQ substliste2 (GET node2 'substi))
      (SETQ substliste2 (SETF (GET node2 'substi)
(APPEND substliste1 substliste2)))
;;; Hvis node1 er en variabel blir den satt til aa peke
;;; paa den termen (node2) som substituerer den. Node2
;;; sin substitusjonsliste faar node1 som nytt element.
     (SETQ f-navn (GET node1 'label))
     (IF (EQL f-navn NIL)
         (PROGN (REMPROP node1 'fedre)
                (REMPROP node1 'merke)
                (REMPROP node1 'barn)
;;; Variabelen blir markert som substituert ved at
```

```
;;; pekeren i SUBSTI er et symbol.
                (SETF (GET node1 'substi) node2)
                (SETF (GET node2 'substi) (CONS node1
substliste2)) ))))

;;; BYTT-UT-BARN
;;; Node1 sine fedre faar node2 som barn i stedet for node1.

(DEFUN bytt-ut-barn (node2 farsliste)
      (LET (( alle-barn NIL))
      (DO* ((far (CAR farsliste) (CAR farsliste))
            (farsliste (CDR farsliste) (CDR farsliste)))
           ((NULL far) NIL)
           (SETQ alle-barn (GET (CAR far) 'barn))
           (SETF (AREF alle-barn (CADR far)) node2)
           (SETF (GET (CAR far) 'barn) alle-barn))))

;;;; BYTT-UT-SUBSTITUTTENE
;;;  Substituerte variable som peker paa node1 blir satt til
;;;  aa peke paa node2.

(DEFUN bytt-ut-substituttene (node2 substliste)
      (DO* ((var (CAR substliste) (CAR substliste))
            (substliste (CDR substliste) (CDR substliste)))
           ((NULL var) NIL)
           (SETF (GET var 'substi) node2) ))

;;;; FJERN-SOM-FAR
;;;  Fjerner alle tilbakepekere fra barna til den noden som
;;;  skal substitueres vekk. Noden representerer den ene av to
;;;  unifiserte termer hvis den har noen barn.

(DEFUN fjern-som-far (node1)
      (LET ((alle-barn1 NIL)
            (ant-barn)
            (barne-node)
            (far)
            (farsliste))
       (SETQ alle-barn1 (GET node1 'barn))
       (SETQ ant-barn (ARRAY-DIMENSION alle-barn1 0))
       (DO* ((ind 0 (+ ind 1)))
            ((= ind ant-barn) NIL)
            (SETQ barne-node (AREF alle-barn1 ind))
            (SETQ farsliste (GET barne-node 'fedre))
            (SETQ far (LIST node1 ind))
            (SETQ farsliste (REMOVE far farsliste :test
#'equal))
            (SETF (GET barne-node 'fedre) farsliste) )))
```

```
;;;; SUBSTITUERTE_VARIABLE
;;;  Finner de variablene som er substituert og
;;;  returnerer ei liste med disse.
;;;  INN: Liste over distinkte variable.
;;;  RETURVERDI: Liste over alle substiterte variable.
;;;  SIDEVIRKNING: -

(DEFUN substituerte_variable (varliste)
      (LET ((subst-varliste NIL)
            (term NIL))
      (DO* ((var (CAR varliste) (CAR varliste))
            (varliste (CDR varliste) (CDR varliste)))
           ((NULL var) subst-varliste)
           (SETQ term (GET var 'substi))
;;; Tester om variabelen er markert som substituert.
;;; Den skal da ha et symbol forskjellig fra NIL i
;;;  SUBSTI-feltet.
           (IF (AND (ATOM term) (NOT (NULL term)))
               (SETQ subst-varliste (CONS var
subst-varliste))))))
```

Vedlegg D

Paterson's og Wegman's unifikasjonsalgoritme

I dette vedlegget gjengis en implementasjon i Common LISP av Paterson's og Wegman's unifikasjonsalgoritme. Programmet ble kjørt på UiO's DEC-2065 system.

```
;;;; FINNMGU-PATWEG
;;;  INN: To atomer.
;;;  RETURVERDI: Om ikke unifiserbar: (())
;;;              Om unifiserbar: mgu for atom1 og atom2.
;;;  REPRESENTASJON av atomer, termer og unifikatorer.
;;;              P(t1,...,tn)       representeres som
;;;                                 (P t1 ... tn)
;;;              f(t1,...,tn)       representeres som
;;;                                 (f t1 ... tn)
;;;              f()                representeres som (f NIL)
;;;                                   (Inndata kan ha
;;;                                   konstanter paa formen f)
;;;              x                  representeres som (x)
;;;              (t1/x1,...,tm/xm)  representeres som
;;;                                 ((t1 (x1)) ... (tm (xm)))
;;;
;;; FORFATTER: Dag Diesen
;;;
;;; INTERN DATASTRUKTUR:
;;;
;;;    1) Atomene danner en Endelig Rettet Asyklisk Graf,
;;;       der alle variablene er distinkte.
;;;    2) Alle distinkte variable er samlet i
;;;       en variabelliste.
;;;    3) Alle funksjonsnoder er samlet i ei funksjonsliste.
;;;
;;;
;;;
;;;
```

```
;;;  Hver node i grafen er et symbol, og pekere til andre
;;;  noder finnes i symbolets property-list. Hver node som
;;;  representerer et atom eller funksjon er et
;;;  maskingenerert symbol. Hver node som representerer
;;;  en variabel er det symbol som baerer variabelens navn.
;;;
;;;  TERMENE har dette innhold i sine property-lister:
;;;
;;;          Et ATOM:
;;;             Label:  Navnet paa atomet.
;;;             Barn:   Tabell med pekere til de nodene som
;;;                     representerer
;;;                     termene i atomet.
;;;             Fedre:  NIL
;;;
;;;          En FUNKSJON:
;;;             Label:  Navnet paa funksjonen.
;;;             Barn:   Tabell med pekere til de nodene
;;;                     som representerer subtermene
;;;                     i funksjonen.
;;;             Fedre:  Liste av tilbakepekere. Hver peker
;;;                     bestaar av navnet paa farsnoden,
;;;                     og et nummer som gir funksjonens
;;;                     plass i s|skenflokken.
;;;
;;;          En VARIABEL:
;;;             Label:  (VARIABEL)
;;;             Barn:   Tabell med null elementer.
;;;             Fedre:  Liste av tilbakepekere. Hver peker
;;;                     bestaar av navnet paa farsnoden, og
;;;                     et nummer som gir variabelens plass
;;;                     i s|skenflokken.
;;;
;;;          Alle symboler kan i tillegg ha disse pekerene:
;;;             Ekv:    Liste med ekvivalenskanter.
;;;             Peker:  Peker til noden som representerer
;;;                     ekvivalensklassen.
;;;             Stakk:  Ubehandlede noder i ekvivalensklassen.
;;;             Status: Ferdig behandlede noder merkes med
;;;                     FJERNET.
;;;
(DEFUN finnmgu-patweg (atom1 atom2)
       (LET ((tdag NIL)
             (node1 NIL)
             (node2 NIL)
             (varliste NIL)
             (fn-liste NIL)
             (sigma NIL)
```

```
            (mgu-ind 0))
;;;Danner den Endelig Rettet Asykliske Grafen.
       (SETQ tdag (erag-atom atom1 varliste fn-liste))
       (SETQ node1 (CAR tdag))
       (SETQ varliste (CADR tdag))
       (SETQ fn-liste (CADDR tdag))
       (SETQ tdag (erag-atom atom2 varliste fn-liste))
       (SETQ node2 (CAR tdag))
       (SETQ varliste (CADR tdag))
       (SETQ fn-liste (CADDR tdag))
       (SETQ atom1 NIL)
       (SETQ atom2 NIL)
;;;Unifiserer.
       (CATCH 'non-mgu (lin_unify fn-liste varliste node1
node2))
))

;;;; ERAG-ATOM
;;;  Omformer listen av de termene som skal unifiseres
;;;  til en endelig rettet asyklisk graf (ERAG). I denne
;;;  grafen er variablene distinkte og kan derfor ha
;;;  flere pekere til seg.
;;;  Ei liste over alle distinkte  variable lages ogsaa.
;;;
;;;  INN: Liste som representerer et av atomene som skal
;;;        unifiseres. Variabelliste. Funksjonsliste.
;;;  RETURVERDI:  Liste som bestaar av toppnoden i grafen,
;;;               oppdatert variabelliste og funksjonsliste.
;;;  SIDEVIRKNING: Dannelse av en rettet asyklisk graf der
;;;                alle variable er distinkte.

(DEFUN erag-atom (atomet varliste fn-liste)
    (LET ((pnavn NIL)
          (var-fn-liste NIL)
          (*node*))
    (DECLARE (SPECIAL *node*))
;;; lager de to f|rste nodene i nettet. Kaller erag-term som
;;; lager resten av nettet.
       (SETQ pnavn (CAR atomet))
       (SETQ *node* (GENTEMP))
       (SETF (GET *node* 'label) pnavn)
       (SETF (GET *node* 'fedre) NIL)
       (SETQ fn-liste (CONS *node* fn-liste))
       (SETQ atomet (CDR atomet))
       (SETQ var-fn-liste (erag-term atomet *node* varliste
fn-liste))
       (SETQ varliste (CAR var-fn-liste))
       (SETQ fn-liste (CADR var-fn-liste))
```

```
       (LIST *node* varliste fn-liste)))

;;;; ERAG-TERM.
;;;  INN:            Liste av subtermer fra den funksjonen
;;;                  som danner farsnoden.
;;;                  Farsnoden som subtermen skal knyttes til.
;;;                  Liste med alle distinkte variable funnet
;;;                  saa langt.
;;;                  Liste med funksjoner funnet saa langt.
;;;  SIDEVIRKNINGER: Bygger opp nettet for alle subtermene og
;;;                  knytter frbindelsene med farsnoden.
;;;  RETURVERDI:     Oppdatert variabelliste og
;;;                  funksjonsliste.

(DEFUN erag-term (termi fars-node varliste fn-liste)
      (LET ((*node* NIL)
            (label NIL)
            (barn-nr -1)
            (var-fn-liste NIL)
            (brorsliste NIL))
       (DECLARE (SPECIAL *node*))
       (DO* ((symbol (CAR termi) (CAR termi))
             (termi (CDR termi) (CDR termi)))
       ((NULL symbol) varliste fn-liste brorsliste barn-nr)
       (SETQ barn-nr (+ barn-nr 1))
       (IF (ATOM symbol)
           (PROGN (SETQ *node* (GENTEMP))
                  (SETQ fn-liste (lag-f-node *node* symbol
fars-node fn-liste barn-nr))
                  (SETQ brorsliste (CONS *node* brorsliste))
                  (ingen-barn *node*))
           (IF (= 1 (LENGTH symbol))
               (PROGN (SETQ label (CAR symbol))
                      (SETQ varliste (oppdat-var label
fars-node varliste barn-nr))
                      (SETQ brorsliste (CONS label
brorsliste))
                      (ingen-barn label))
               (PROGN (SETQ label (CAR symbol))
                      (SETQ *node* (GENTEMP))
                      (SETQ fn-liste (lag-f-node *node*
label fars-node fn-liste barn-nr))
                      (SETQ brorsliste (CONS *node*
brorsliste))
                      (SETQ symbol (CDR symbol))
                      (SETQ var-fn-liste (erag-term symbol
*node* varliste fn-liste))
                      (SETQ varliste (CAR var-fn-liste))
```

```
                    (SETQ fn-liste (CADR var-fn-liste)) ))))
       (SETQ brorsliste (REVERSE brorsliste))
       (SETQ barn-nr (+ barn-nr 1))
       (sett-inn-broedre fars-node brorsliste barn-nr)
       (LIST varliste fn-liste)))

;;;; LAG-F-NODE
;;;  INN: node, funksjonsnavn, farsnode, funksjonsliste,
;;;       nodens plass i s|skenflokken.
;;;  RETURVERDI: Oppdatert funksjonsliste.
;;;  SIDEVIRKNING: Lager en funksjonsnode ved aa legge
;;;                inn funksjonsnavn og tilbakepeker til
;;;                farsnoden i property-list.

(DEFUN lag-f-node (node fnavn far fn-liste barn-nr)
      (SETF (GET node 'label) fnavn)
      (SETF (GET node 'fedre) (LIST (LIST far barn-nr)))
      (CONS node fn-liste))

;;;; LAG-V-NODE
;;;  INN: variabelnavn, farsnode, variabelliste,
;;;       variabelens plass i s|skenflokken.
;;;  RETURVERDI: Oppdatert variabelliste
;;;  SIDEVIRKNING: Lager en variabelnode ved aa
;;;                legge inn tilbakepeker
;;;                til farsnoden property-list.

(DEFUN lag-v-node (var-navn far varliste barn-nr)
      (SETF (GET var-navn 'fedre) (LIST (LIST far barn-nr)))
      (SETF (GET var-navn 'label) '(variabel))
      (SETF (GET var-navn 'ekv) NIL)
      (SETF (GET var-navn 'peker) NIL)
      (SETF (GET var-navn 'status) NIL)
      (SETF (GET var-navn 'stakk) NIL)
      (CONS var-navn varliste))

;;;; V-SOEK
;;;  INN: variabelnavn, variabelliste.
;;;  RETURVERDI: T om variabelen finnes, NIL ellers.
;;;  SIDEVIRKNING: -

(DEFUN v-soek (var-navn varliste)
      (LET ((funnet NIL))
      (DO* ((var (CAR varliste) (CAR varliste))
            (varliste (CDR varliste) (CDR varliste)))
           ((OR (NULL var) funnet) funnet)
           (IF (EQL var var-navn)
               (SETQ funnet 'T))) ))
```

```
;;;; NY-FAR-V
;;;  INN: variabelnavn, farsnode, variabelens plass
;;;       i s|sskenflokken.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Legger inn ny tilbakepeker til farsnoden.

(DEFUN ny-far-v (varnavn far barn-nr)
      (LET ((f-liste NIL))
      (SETQ f-liste (GET varnavn 'fedre))
      (SETF (GET varnavn 'fedre) (CONS (LIST far barn-nr)
f-liste)) ))

;;;; OPPDAT-VAR
;;;  INN: variabelnavn, farsnode, variabelliste.
;;;  RETURVERDI: Oppdatert variabelliste.
;;;  SIDEVIRKNING: Hvis variabelen er ny lages en
;;;                variabelnode og variabellista
;;;                oppdateres, ellers faar variabelen
;;;                tilbakepeker til ny far.

(DEFUN oppdat-var (varnavn far varliste barn-nr)
      (IF (v-soek varnavn varliste)
          (ny-far-v varnavn far barn-nr)
          (SETQ varliste (lag-v-node varnavn far
varliste barn-nr)))
      varliste)

;;;; SETT-INN-BROEDRE
;;;  INN: farsnode, liste over br|dre, antall br|dre.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Lager en tabell og plasserer br|drene
;;;                i den. Setter inn tabellen som  barn
;;;                i farsnodens property-list.

(DEFUN sett-inn-broedre (far brorsliste ant-barn)
      (LET (( brors-tab))
      (SETQ brors-tab (MAKE-ARRAY ant-barn
:INITIAL-CONTENTS brorsliste))
      (SETF (GET far 'barn) brors-tab) ))

;;;; INGEN-BARN
;;;  INN: node til en variabel eller konstant.
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Setter inn tom tabell i property-list
;;;  til noden.

(DEFUN ingen-barn (node)
```

```
      (LET ((brors-tab))
      (SETQ brors-tab (MAKE-ARRAY 0))
      (SETF (GET node 'barn) brors-tab) ))

;;;; LIN_UNIFY
;;;  Pr|ver aa finne den mest generelle unifikator av
;;;  to termer. Termene maa vaere representert som en
;;;  endelig asyklisk graf der alle variable er distinkte.
;;;  Liste over distinkte variable maa eksistere.
;;;  Funksjons liste med alle funksjonsnoder (og
;;;  konstanter) maa finnes.
;;;  INN: Liste av alle atomer i nettet som representerer
;;;       funksjoner, liste av alle atomer i nettet som
;;;       representerer variable, de to toppnodene i nettet.
;;;  UT: Liste over alle substitusjoner. Substitusjonene
;;;      ikke paa eksplisitt form.
;;;  SIDEVIRKNINGER: Ved vellykket substitusjon vil alle
;;;                  nodene i nettet vaere merket som
;;;                  fjernet. Property-felter for
;;;                  ekvivalenskanter,lokal stakk, peker
;;;                  og status vil vaere opprettet for
;;;                  alle eller noen noder.
;;;
(DEFUN lin_unify (fn-liste varliste node1 node2)
       (LET ((mgu))
       (sett_ekv node1 node2)
       (DO* ((f-node (CAR fn-liste) (CAR fn-liste))
             (fn-liste (CDR fn-liste) (CDR fn-liste)))
            ((NULL f-node) mgu)
            (IF (eksisterer_node_p f-node)
                (SETQ mgu (rot_klasse f-node mgu))))
       (DO* ((v-node (CAR varliste) (CAR varliste))
             (varliste (CDR varliste) (CDR varliste)))
             ((NULL v-node) mgu)
             (IF (eksisterer_node_p v-node)
                 (SETQ mgu (rot_klasse v-node mgu))))
       (REVERSE mgu)))

;;;; ROT_KLASSE
;;;  Behandler en ekvivalensklasse, men f|rst naar den utgj|r
;;;  en rot i nettverket.
;;;  INN:  En startnode og ei liste av substitusjoner.
;;;  UT:   Oppdatert mgu.
;;;  SIDEVIRKNINGER: Den ekvivalensklassen som startnoden
;;;                  tilh|rer faar merket sine noder fjernet.
;;;                  Andre property-felt faar satt verdier.
;;;                  Fedrene til nodene i den
;;;                  ekvivalensklassen som behandles blir
```

```
;;;                  behandlet f|r ekvivalensklassen til
;;;                  startnoden blir behandlet (Gjennom
;;;                  rekursivt kall).

(DEFUN rot_klasse (r mgu)
       (IF (peker_definert_p r)
           (THROW 'non-mgu (SETQ mgu '(())))
           (sett_peker r r))
;;; Lager ny stakk og setter inn f|rste element.
           (push r r)
           (DO (( s NIL))
               ((tom_stakk_p r) mgu)
               (SETQ s (pop r))
               (IF (ulike_funksj_p s r)
                   (THROW 'non-mgu (SETQ mgu '(())))
               (SETQ mgu (se_paa_fedre s mgu))
               (behandle_ekv_noder s r)
               (IF (AND (NOT (EQL s r)) (eksisterer_node_p s))
                   (PROGN (IF (var_node_p s)
                              (SETQ mgu (CONS (lag_subst s r)
mgu))
                              (sett_soenner_ekv s r))
                   (fjern_node s))))
           (fjern_node r)
           mgu)

;;;; ULIKE_FUNKSJ_P
;;;  Unders|ker om nodene representerer to ulike funksjoner.
;;;  INN: node1, node2
;;;  RETURVERDI: T om de to nodene representerer to ulike
;;;              funksjoner, NIL om termene representerer
;;;              samme funksjon eller om en av termene er
;;;              en variabel.
;;;  SIDEVIRKNING: -

(DEFUN ulike_funksj_p (node1 node2)
      (LET ((f-navn1 NIL)
            (f-navn2 NIL))
      (SETQ f-navn1 (GET node1 'label))
      (SETQ f-navn2 (GET node2 'label))
      (COND ((EQL f-navn1 f-navn2) NIL)
            ((EQUAL f-navn1 '(variabel)) NIL)
            ((EQUAL f-navn2 '(variabel)) NIL)
            (T T)) ))

;;;; SE_PAA_FEDRE
;;;  Traverserer farsnodene til inn-noden.
;;;  Kaller (rekursivt) paa rot_klasse.
```

```
;;;  INN: Inn-node og mgu
;;;  RETURVERDI: Oppdatert mgu.
;;;  SIDEVIRKNING: Samme sidevirkning som kall paa
;;;                rot_klasse for alle noder som h|rer
;;;                til ekvivalensklassen til inn-nodens
;;;                farsnoder.

(DEFUN se_paa_fedre (s mgu)
       (LET ((farsliste))
       (SETQ farsliste (GET s 'fedre))
       (DO (( far (CAAR farsliste) (CAAR farsliste))
            (farsliste (CDR farsliste) (CDR farsliste)))
           ((NULL far) mgu)
           (IF (eksisterer_node_p far)
               (SETQ mgu (rot_klasse far mgu)) ))))

;;;; BEHANDLE_EKV_NODER
;;;  Behandler noder som er satt ekvivalente.
;;;  INN: Node med ekvivalenspekere, noden som representerer
;;;       ekvivalens-klassen
;;;  RETURVERDI: -
;;;  SIDEVIRKNING: Ekvivalenskantene til noden med
;;;                ekvivalenspekere blir fjernet.
;;;                Noden med ekvivalenspekere faar de
;;;                ekvivalente nodene paa sin stakk.
;;;                Hvis noen av de ekvivalente nodene
;;;                tilh|rer en annen ekvivalensklasse stopper
;;;                programmet, fordi termene ikke kan
;;;                unifiseres.

(DEFUN behandle_ekv_noder (s r)
       (LET ((ekvliste))
       (SETQ ekvliste (GET s 'ekv))
       (DO* (( trm (CAR ekvliste) (CAR ekvliste))
             (ekvliste (CDR ekvliste) (CDR ekvliste)))
            ((NULL trm))
            (IF (NOT (peker_definert_p trm))
                (sett_peker trm r))
            (IF (NOT (EQL (finn_peker trm) r))
                (THROW 'non-mgu (SETQ mgu '())))
            (fjern_ekv s trm)
            (push trm r))))

;;;; SETT_SOENNER_EKV (r s)
;;;  Setter ekvivalenskanter for s|nnene til to
;;;  ekvivalente noder.
;;;  INN: De to nodene som er ekvivalente.
;;;  RETURVERDI: -
```

```
;;;  SIDEVIRKNING: Setter ekvivalenskanter.

(DEFUN sett_soenner_ekv (r s)
       (LET (( barntab1)
             (barntab2)
             (ant-barn)
             (barn1)
             (barn2))
       (SETQ barntab1 (GET r 'barn))
       (SETQ barntab2 (GET s 'barn))
       (SETQ ant-barn (ARRAY-DIMENSION barntab1 0))
       (DO ((ind 0 (+ ind 1)))
           ((= ant-barn ind))
           (SETQ barn1 (AREF barntab1 ind))
           (SETQ barn2 (AREF barntab2 ind))
           (sett_ekv barn1 barn2))))

;;;; LAG_SUBST
;;;  Lager et substitusjonspar av en variabel og en term.
;;;  INN: Variabel og den substituerte termen.
;;;  RETURVERDI: Variabelen og den substituerte termen
;;;              paa listeform:
;;;              ((variabel) term)
;;;  SIDEVIRKNING: -

(DEFUN lag_subst (var term)
(LIST (LIST var) (liste_av_termer term)))

;;;; LISTE_AV_TERMER
;;;  Bygger opp den substituerte termen til listeform
;;;  ved aa gaa gjennom de aktuelle nodene i grafen.
;;;  INN: Noden som representerer termen.
;;;  RETURVERDI: Termen paa listeform.
;;;  SIDEVIRKNING: -

(DEFUN liste_av_termer (term)
      (LET ((alle-barn NIL)
            (ant-barn)
            (f-navn NIL)
            (sigma NIL)
            (subterm))
      (SETQ f-navn (GET term 'label))
      (IF (EQUAL f-navn '(variabel))
          (SETQ sigma (LIST term))
          (PROGN (SETQ alle-barn (GET term 'barn))
                 (SETQ ant-barn (ARRAY-DIMENSION alle-barn 0))
                 (IF (= ant-barn 0)
                     (SETQ sigma (LIST f-navn NIL))
```

```
                    (PROGN (DO* ((ind 0 (+ ind 1)))
                                ((= ind ant-barn) sigma)
                                (SETQ subterm (AREF alle-barn
ind))
                                (SETQ sigma (CONS
(liste_av_termer subterm) sigma)))
                           (SETQ sigma (CONS f-navn
(REVERSE sigma))) ))))))

;;;; SETT_EKV
;;;  Setter ekvivalenskant mellom to noder.

(DEFUN sett_ekv (node1 node2)
       (LET ((ekvliste))
       (SETQ ekvliste (GET node1 'ekv))
       (SETQ ekvliste (CONS node2 ekvliste))
       (SETF (GET node1 'ekv) ekvliste)
       (SETQ ekvliste (GET node2 'ekv))
       (SETQ ekvliste (CONS node1 ekvliste))
       (SETF (GET node2 'ekv) ekvliste) ))

;;;; FJERN-EKV
;;;  Fjerner ekvivalens-kant mellom to noder.

(DEFUN fjern_ekv (node1 node2)
       (LET ((ekvliste))
       (SETQ ekvliste (GET node1 'ekv))
       (SETQ ekvliste (REMOVE node2 ekvliste))
       (SETF (GET node1 'ekv) ekvliste)
       (SETQ ekvliste (GET node2 'ekv))
       (SETQ ekvliste (REMOVE node1 ekvliste))
       (SETF (GET node2 'ekv) ekvliste) ))

;;;; EKSISTERER_NODE_P
;;;  Tester om noden er fjernet fra nettet.
;;;  RETURVERDI: T hvis noden eksisterer,
;;;              NIL hvis den er fjernet.

(DEFUN eksisterer_node_p (node)
       (IF (EQL (GET node 'status) NIL)
           T
           NIL))

;;;; VAR_NODE_P
;;;  Tester om noden representerer en variabel.
;;;  RETURVERDI: T hvis noden er en variabel,
;;;              NIL hvis den er en funksjon.
```

```
(DEFUN var_node_p (node)
       (IF (EQUAL (GET node 'label) '(variabel))
           T
           NIL))

;;;; FJERN_NODE
;;;  Merker noden som fjernet.

(DEFUN fjern_node (node)
       (SETF (GET node 'status) 'fjernet))

;;;; PEKER_DEFINERT_P
;;;  Tester om det er satt noen peker for noden.
;;;  RETURVERDI: T hvis peker er satt,
;;;              NIL ellers.

(DEFUN peker_definert_p (node)
       (IF (EQL (GET node 'peker) NIL)
           NIL
           T))

;;;; FINN_PEKER
;;;  RETURVERDI: Pekeren til den noden som representerer
;;;              ekvivalensklassen. NIL om ingen slik
;;;              peker finnes.
;;;  INN:        Node som eventuelt har en peker.

(DEFUN finn_peker (node)
       (GET node 'peker))

;;;; SETT_PEKER
;;;  Setter peker for en node til den noden som
;;;  representerer ekvivalensklassen.
;;;  INN: node, noden som representerer
;;;       ekvivalensklassen.

(DEFUN sett_peker (node ekvklasse)
       (SETF (GET node 'peker) ekvklasse))

;;;; PUSH
;;;  Legger peker til en node i stakken til en
;;;  annen node.

(DEFUN push (node stakk-node)
       (LET ((stakkliste))
       (SETQ stakkliste (GET stakk-node 'stakk))
       (SETQ stakkliste (CONS node stakkliste))
       (SETF (GET stakk-node 'stakk) stakkliste) ))
```

```
;;;; POP
;;;  Henter ut f|rste peker til en node fra stakken
;;;  til oppgitt node.
;;;  RETURVERDI: F|rste peker til en node i stakken.

(DEFUN pop (stakk-node)
       (LET ((stakkliste)
             (returnode))
       (SETQ stakkliste (GET stakk-node 'stakk))
       (SETQ returnode (CAR stakkliste))
       (SETQ stakkliste (CDR stakkliste))
       (SETF (GET stakk-node 'stakk) stakkliste)
       returnode ))

;;;; TOM_STAKK_P
;;;  Tester om stakken til oppgitt node er tom.
;;;  RETURVERDI: T om stakken er tom,
;;;              NIL ellers.

(DEFUN tom_stakk_p (stakk-node)
       (IF (EQL (GET stakk-node 'stakk) NIL)
           T
           NIL))
```