Databaser E2001: Kort om deduktive databaser, integritetsbegrænsninger og "simplification", samt på vej mod en metode til automatisk generering af triggers og opdateringsrutiner

Henning Christiansen

6. november 2001

Resumé

Dette notat beskriver et aktuelt forskningsemne indenfor databaser. Det handler om effektiv beregning af og metoder til opretholdelse af integritetsbegrænsninger. Integritetsbegrænsninger er almindeligvis noget tungt stads at skulle beregne for hver ændring og en teknik kaldet "simplification" handler om at udnytte viden om, at integritetsbegrænsninger er kontrolleret allerede for den aktuelle database — så man altså kun behøver at kontrollere effekten af ændringen. Det vil typisk mindske tidsforbruget med en størrelsesorden.

Simplification er velkendt i litteraturen, og vi ønsker her at videreudvikle teknikken, så vi kan udnytte den i forhold til at generere opdateringsrutiner, som f.eks. kan give detaljeret besked om, hvorfor en given opdatering må afvises med henvisning til integritetsbegrænsningerne. Dette kan f.eks. bruges i en dialog med brugeren eller til automatisk genopretning af databasen (som generalisering af SQLs "cascade").

Modellen er i første omgang formuleret i en matematisk-logisk ramme, og næste skridt er så at omsætte den til noget, der kan sættes på som en front-end til et RDBMS. Det er pt. under overvejelse om modellen kan formuleres direkte i relationel algebra, men det har ikke været forsøgt endnu.

1 Indledning

Første-ordens logik kan benyttes som alternativ til relationel algebra til at formulere et grundlag for at forstå og ræsonnere om relationel databaseteknologi. Vi taler om deduktive databaser, hvor "skemaerne" er beskrevet ved et antal basisrelationer plus klausuler som definerer views ("definerede relationer"), forespørgsler og integritetsbegrænsninger. Fordelen ved den logiske formulering er, at man kan trække på et velunderbygget teoretisk grundlag, og at databasebegrebet på naturlig måde udvides med rekursion.

Deduktive databaser benyttes ofte som udtryksmedium i forskning og udvikling indenfor databaser, men der er også udviklet egentlige databasesystemer baseret på deduktive databaser. Deduktive databaser er som teori velegnet til ræsonnering som går ud over direkte forespørgsel-svar-mekanismer, f.eks. opdatering gennem views og beregning af integritetsbegrænsninger.

I dette notat benytter vi logiske databaser som en alternativ teoretisk model, og der ligger ikke heri specielt en afstandtagen fra relationel algebra eller relationelle databaser.

For forudsætninger og historisk baggrundsstof henvises til:

- J. Ullman, J. Widom: A First Course in Database Systems, Prentice-Hall, 1997. Afsnit 4.2–4.4
- H. Gallaire, J. Minker, J.-M. Nicolas: Logic and Databases: A Deductive Approach. ACM Computing Surveys, Volume 16, Issue 2 (June 1984). pp. 153-185.
- ... eller et kendskab til logikprogrammeringssproget Prolog, f.eks. baseret på første halvdel af I. Bratko: Prolog, Programming for Artificial Intelligence, 3rd edition. Addison-Wesley 2001.

Meget af det, som er beskrevet i notatet, bygger naturligvis på andres tidligere arbejde, men vi har ikke inkluderer henvisninger dertil. Læsere som ønsker at studere emnet dybere opfordres til at henvende sig til nærværende forfatter.

Oversigt

I afsnit 2 giver vi en uformel indføring til en logisk databasemodel og beskriver dens forhold til den relationelle model. Afsnit 3 forklarer kort, hvad vi forstå ved en opdatering og afsnit 4 definere begrebet simplification og giver eksempel på, hvordan det kan udnyttes til beregning og opretholdelse af integritetsbegrænsninger. Afsnit 5 er et første bud på en formalisering af en detaljeret procedure til af foretage simplification m.v. — De fleste læsere kan nøjes med den generelle forståelsesniveau, som afsnit 4 forsøger at formidle, og kun sæligt interesserede vil have udbytte af afsnit 5 ophobninger af matematiske symboler.

2 Notation og eksempler

For de, som kender til programmeringssproget Prolog, kan man beskrive en deduktiv database som et Prologprogram uden strukturer, dvs. de eneste data programmet kan udtale sig om er konstantsymboler.

En database har et antal *prædikatsymboler*, eller kort *prædikater*, som svarer til relationsnavne i relationel algebra. Prædikater opdeles i *ekstensionelle* og *intensionelle*. De ekstensionelle svarer til basisrelationer, som er defineret ved en sekvens af fakta, de ekstensionelle ved regler, der svarer til views og sammensatte forespørgsler. Vi viser først et eksempel og forklarer notationen bagefter.

Eksempel: Som gennemgående eksempel benyttes en database over familerelationer. De ekstensionelle prædikater f og m står for relationer om fædre og mødre, så f.eks. $m(a, b)$ betyder at individet (angivet ved) a er mor til individet (angivet ved) b. Vi har ekstensionelle prædikater b for bedsteforældre, s for søskende, p for forældre og ff for forfædre.

Prædikatet e står for "eksisterer som person" og kunne i princippet udvides med forskellig information så som fulde navn, adresse, fødselsår, osv. Følgende formler angiver en en instans af databasen:

$$
\begin{aligned}
&e(john)\\
&e(mary)\\
&e(jane)\\
&e(peter)\\
&e(paul)\\
&f(john, mary)\\
&m(jane, mary)\\
&p(x, y) \leftarrow f(x, y)\\
&p(x, y) \leftarrow m(x, y)\\
&s(x, y) \leftarrow x \neq y \wedge p(z, x) \wedge p(z, y)\\
&b(x, y) \leftarrow p(x, z) \wedge p(z, y)\\
&ff(x, y) \leftarrow p(x, y)\\
&ff(x, y) \leftarrow p(x, z) \wedge ff(z, y)
\end{aligned}
$$

Reglerne består af et hovede (f.eks. $s(x, y)$), en medfører-pil (skrevet "baglæns") og en krop (f.eks. $x \neq y \wedge p(z, x) \wedge p(z, y)$). Der er underforstået en alkvantor uden om en klausul, som dækker de frie variable i den. Så f.eks. reglen for s skal forstås:

$$\forall x, y, z \left(s(x, y) \leftarrow x \neq y \wedge p(z, x) \wedge p(z, y) \right)$$

I kroppen af en klausul kan stå kald af prædikater, herunder til test så som $\neq$. Reglerne forventes at være "range restricted" hvilket vil sige: Variable i hovedet og variable som indgår i et test skal forekomme et kald til et databaseprædikat i kroppen. — Man kan redegøre for, at denne betingelse vil vil gøre, at de intensionelle prædikater faktisk definerer meningsfulde databaserelationer. Intuitivt betyder det, at der er kald i kroppen som knytter værdier til alle variable, når man forsøger at udregne en forespørgsel.

Vi kan også tillade negation i kroppen, og vi begrænser os her til negation på ekstensionelle prædikater. Følgende prædikat o definerer mængden af forældreløse børn:

$$o(x) \leftarrow e(x) \wedge \neg(\exists y\, f(y, x)) \wedge \neg(\exists y\, m(y, x))$$

Vi tillader negerede mål at have lokalt kvantificerede variable. Her udvider vi definitionen af "range restricted" som følger: Variable i et negeret mål skal enten være dækket af en $\exists$ eller forekomme i et ikke negeret databaseprædikat i kroppen.

Semantikken af en database er den mængde relationer som databasen bestemmer, dvs. de ekstensionelle som er angivet eksplicit, og hvad reglerne udfra de ekstensionelle giver for de intensionelle. Dette kan beskrives præcist på flere måder; her nøjes vi med at karakterisere det ved en bottom-up evaluator. M er en variabel som, når nedenstående algoritme terminerer, rummer samtlige fakta for alle relationer.

> M:= mængden af fakta i databasen;
> så længe M vokser, udfør
> M:= $M \cup \{$ethvert faktum H som fås ved at indsætte
>   værdier i en regel $H \leftarrow B$ således at B er
>   tilfredsstillet af $M\}$

Et kald af et prædikat, f.eks. $p(a, b)$, er tilfredsstillet af en mængde M såfremt $p(a, b) \in M$. Et negeret mål, f.eks. $\exists x\, \neg p(a, x)$ er tilfredsstillet af M hvis der ikke findes nogen værdi for x, så $p(a, x) \in M$. Værdien af et test $a \neq b$ er tilfredsstillet såfremt a og b er forskellige og tilsvarende for øvrige tests, vi måtte vælge at inkludere.

Vi betragter den afsluttende værdi af M som en karakterisering af databasens semantik og understreger samtidigt at det ikke tænkt som en praktisk beregningsmetode. Hvis et faktum F ligger i den afsluttende M-mængde konstrueret udfra en databasetilstand DB siger vi, at det er en logisk konsekvens DB hvilket vi skriver $DB \models F$. Sandhedsrelationen generaliseres på naturlig måde til vilkårlige logiske formler bygget vha. logiske operationer så som $\wedge$ og $\vee$.

Man kan overbevise sig om, at $DB \models C$, for enhver regel $C \in DB$.

Databaseregler, som vi har beskrevet dem, er formler af formen $H \leftarrow B$. Vi kan udelade hovedet H og pr. konvention skal det "ingenting" som står til venstre for pilen læses som logisk falskhed; hvis noget medfører falskhed, så må det selv være falsk. Altså skal $\leftarrow B$ læses som "B er falsk", og $DB \models \leftarrow B$ betyder således at B ikke gælder i DB.

Eksempel: Lad os betragte databasen beskrevet ovenfor incl. reglen for o prædikatet, og lad os se på, hvordan M-mængden i definitionen af semantikken ovenfor ændrer sig gennem ved gentagne gennemløb. Vi starter med M:=

$$\{e(john), e(mary), e(jane), e(peter), e(paul), f(john, mary), m(jane, mary)\}$$

Ved første gennemløb får vi tilføjet disse fakta for p:

$$p(john, mary), p(jane, mary)$$

Der tilføjes ikke noget for s og *ff* da der i den værdi for M vi startede med ikke findes noget om p. Endelig giver o-reglen i ét huk, at alle på nær *mary* er forældreløse i forhold til nævnte database, dvs. følgende tilføjes også:

$$o(john), o(jane), o(peter), o(paul)$$

Til næste gennemløb vil tilstedeværelsen af p-fakta gøre, at reglen for s kan komme i sving, men der findes rent faktisk ingen søskendepar i databasen så vi får intet om s (definitionen kan give $x = mary$ og $y = mary$, men $mary \neq mary$ fejler). Den ikke-rekursive regel for *ff* giver os en mængde *ff*-fakta svarende til de nu kendte p-fakta. Nu er processen stabiliseret, et næste gennemløb giver ikke flere nye fakta: Hvis vi havde haft flere f- og m-fakta fra starten kunne vi måske have haft brug for flere iterationer for at få samlet fakta for forfaderrelationen *ff* op.

Der er egenskaben om "range restricted" som gør, at denne semantiske definition altid fungerer.

Lader vi DB_0 stå for databasen ovenfor har vi eksempelvis følgende:

$$DB_0 \models p(john, mary),\ DB_0 \models p(john, mary) \wedge e(paul),\ DB_0 \models \leftarrow \neg e(paul)$$

$$DB_0 \models \leftarrow p(x, y) \wedge p(y, x)$$

Bemærk for den sidste formel, at der er en implicit $\forall x, y$ uden om formlen, så den skal altså læses "for all x, y gælder ikke samtidigt $p(x, y)$ og $p(y, x)$" eller på dansk "ingen er forælder til sin forælder".

I mange tilfælde vil man kunne beregne en databases semantik ved at give det som program til en Prolog-fortolker og bede om samtlige løsninger til forespørgsler $e(x)$, $f(x, y)$, $m(x, y)$, $p(x, y)$ osv. Prologfortolkeren vil dog stå tilbage i forhold til bottom-up evalueringen i visse situationer:

- Variable som indgår i tests og negerede kald (bortset fra dem, som er dækket af $\exists$) skal være bundet af tidligere kald i kroppen for at de respektive tests og negerede kald kan evalueres korrekt af Prolog-fortolkeren.
  (F.eks. vil der gå kludder i reglen for s idet $x \neq y$ vil blive forsøgt udregnet før de efterfølgende kald til p har instantieret x of y.)

- Prolog vil gå i uendelig løkke på en regel som $h(x) \leftarrow h(x)$, hvor bottom-up-evalueringen blot vil konstatere, at denne regel ikke giver noget bidrag til h.

Om forespørgselsevaluering og sammenhæng med relationel algebra

Det er velkendt at et vilkårligt udtryk i relationel algebra kan oversættes til en klausul. Deduktive databaser har (i vores præsentation) ikke attributnavne, så ved f.eks. en naturlig join skal "oversætteren" holde styr på dem. Det er beskrevet i de dele af Ullman og Widoms bog, vi omtalte i indledningen, så det bruger vi ikke tid på her.

Vi vil antage, der findes en eller anden mekanisme, som er i stand til at evaluere forespørgsler, således at vi givet et kald $p(x, y)$ vil vi kunne få udregnet mængden af tupler $\langle x, y\rangle$, hvor $DB \models p(x, y)$. Noget sådant kunne f.eks. implementeres vha. en oversættelse til SQL[1] som man overlod til et RDBMS at udregne.

Vi antager denne evalueringsmekaniske kan evaluere forespøgsler svarende til vilkårlige regel-kroppe. Vi kan skrive en sådan forespørgsel som

[1] Udvidet på passende vis med rekursion.

følger:

$$\langle x_1, \ldots, x_n \rangle \colon B$$

Resultatet skal svare til resultatet af kaldet $\langle x_1, \ldots, x_n \rangle$ hvis vi leger, at reglen $\langle x_1, \ldots, x_n \rangle \leftarrow B$ var tilføjet databasen mens vi evaluerede. Forespørgsler af denne art skal være “range restricted” (defineret som man kan forvente).

Vi siger en forespørgsel fejler, såfremt den giver et tomt svar, og vi kan se, at $\langle x_1, \ldots, x_n \rangle \colon B$ fejler i forhold til DB hvis og kun hvis $DB \models \leftarrow B$.

Integritetsbegrænsninger

Integritetsbegrænsninger er udsagn, som til enhver tid skal gælde for en databases tilstand.

Vi begrænser os til udsagn af formen $\leftarrow B$, da det er en beregningsmæssig bekvem form, og man kan iøvrigt redegøre for, at alle de integritetsbegrænsninger, vi måtte være interesseret i, kan skrives på denne form. En database har tilknyttet nul eller flere sådanne integritetsbegrænsninger, og den kaldes *inkonsistent*, hvis en af dens integritetsbegrænsninger ikke er overholdt; ellers siges den at være *konsistent.* At $DB \models \leftarrow B$ ikke er overholdt er det samme som at B som forespørgsel giver succes. Dvs. når vi siger at en database DB har en integritetsbegrænsning $\leftarrow B$ betyder “DB er konsistent” altså at B skal fejle i forhold til databasen. Lidt bagvendt, men sådan er terminologien nu en gang.

Følgende begrænsninger er relevante i forhold til vores familiedatabase:

$$\begin{array}{l} \leftarrow f(x,z) \wedge f(y,z) \wedge x \neq y \\ \leftarrow m(x,z) \wedge m(y,z) \wedge x \neq y \\ \leftarrow f(x,y) \wedge m(x,z) \end{array}$$

De kan læses, at inkonsistens gælder, såfremt

- Der er en z med to forskellige fædre x og y,
- eller tilsvarende for mødre,
- eller der findes en x som optræder både som mor og far.

Følgende begrænsning siger noget så fornuftigt som at forfaderrelationen ikke må indeholde cirkler:[2]

$$\leftarrow \mathit{ff}(x,y), \mathit{ff}(y,x)$$

[2] Dvs. vor database egner sig ikke til verdener hvor tidsrejser forekommer.

Vi kan også beskrive "referential integrity constraints" så som at personer nævnt i f og m skal findes i e-relationen.

$$\leftarrow f(x,y) \wedge (\neg e(x) \vee \neg e(y))$$
$$\leftarrow m(x,y) \wedge (\neg e(x) \vee \neg e(y))$$

Bemærk: På denne måde kan vi kun beskrive såkaldt statiske integritetsbegrænsninger, dvs. som går på den specifikke databasetilstand. Såkaldt dynamiske begrænsninger, som f.eks. at en opdatering aldrig på sætte en medarbejder ned i løn, men kun op, kan ikke beskrives.

Kontrol af overholdelsen af integritetsbegrænsninger kan i princippet foretages ved, for hver opdatering af databasen, at evaluere samtlige integritetsbegrænsninger. Hvis de hver især fejler, er alt godt, hvis ikke må vi afvise opdateringen. Men en sådan beregning må forventes at være temmelig tidskrævende, fordi databasen skal gennemtrævles på kryds op tværs (jvf. "Join-effekten").

Det forekommer heller ikke særlig optimalt at gøre det på den måde. Antag, at vi på et givet tidspunkt ved at databasen er konsistent. Det svarer som sagt til, at vi ved, at integritetsbegrænsningerne betragtet som forespørgsler vil fejle i den aktuelle databasetilstand. Lad os nu tænke frem til en ny databasetilstand, som er ligesom den aktuelle, men med tilføjelsen af $f(a,b)$, og overveje evalueringen af integritetsbegrænsningerne på ny: Det synes indlysende at de fleste af de delberegninger, som skal udføres, allerede har været udført for at checke konsistens af den aktuelle tilstand. Så hvorfor gøre dem igen? Det vi vil beskæftige os med i det følgende er netop hvordan man kan holde styr på, hvad der der har været beregnet allerede, og hvad som vi ikke kan undgå at beregne.

3 Hvad forstår vi ved en opdatering?

For nemheds skyld nøjes vi med at se på opdatering ved tilføjelse af nye fakta til en database.

Vi definerer nu en *opdatering* eller *forslag til opdatering* (eng.: *"update (request)"*) som en mængde af ekstensionelle fakta; vi antager, at en opdatering ikke overlapper med den aktuelle database, dvs. man må ikke forsøge at tilføje noget, som allerede er i databasen. — Det, som her kaldes en opdatering, svarer til det som i Ullman og Widoms databasebog kaldes en transaktion.

Lader vi U stå for en eller anden opdatering til en database DB, betyder $DB \models \phi$ altså logisk sandhed af ϕ i den aktuelle database, og $DB \cup U \models \phi$ logisk sandhed i den opdaterede database.

Bemærk: Vi har ikke skrevet detaljerne ud for opdatering ved sletning, men vi vil dog hævde, at vor model kan håndtere sletning, iden sletning kan *simuleres* som en speciel slags tilføjelse. Hvis p (med ét argument) er et prædikat vi ønsker at slette fra, kan vi opfinde et prædikat slet-p, og alle steder, hvor der står $p(x)$ tilføjer vi "$\wedge\neg$slet-$p(x)$". Sletning af $p(a)$ kan nu simuleres ved tilføjelse af slet-$p(a)$. Det såkaldte "update" operation, som ændrer ved en enkelt attribut i en enkelt tupel, kan simuleres ved en sletning plus en tilføjelse. — Denne "simulering" er selvfølgelig ikke måden at gøre tingene på i praksis, men det viser at det, vi kan redegøre for om tilføjelser, også kan generaliseres til at gælde de andre former for opdatering.

4 Simplification: Definition og eksempler

Vi betragter (for nemheds skyld) en fast samling prædikater, regler og integritetsbegrænsninger, og når vi taler om en "vilkårlig database (tilstand)" er det altså med denne del fastholdt, men de ekstensionelle fakta kan variere.

Definition: Lad ϕ være en formel, DB en vilkårlig database med $DB \models \phi$ og U en opdatering i forhold til DB. En formel ψ kaldes en *simplification* af ϕ såfremt

$$DB \cup U \models \phi \text{ hvis og kun hvis } DB \models \psi$$

Simplificationen af ϕ er altså en formel, vi kan checke i den aktuelle databasetilstand DB for at se, om ϕ stadig vil gælde i den opdaterede tilstand uden vi rent faktisk behøver at foretage opdateringen.[3]

Betragt en vilkårlig databasetilstand DB med de regler og integritetsbegrænsninger, vi har set, og antag, at sidstnævnte er overholdt. Betragt nu opdateringen $\{f(\mathit{børge}, \mathit{peter})\}$ og følgende af integritetsbegrænsningerne, som vi kalder ϕ_1:

$$\leftarrow f(x,z) \wedge f(y,z) \wedge x \neq y$$

At den er overholdt betyder, at for samtlige mulige par af f-fakta i databasen med fælles anden-komponent, må første-komponenterne ikke være forskellige. Kunne vi finde $f(a,c)$ og $f(b,c)$ er vores database inkonsistent.

Når det angår den opdaterede tilstand $DB \cup U$ behøver vi jo kun kontrollere de par af f-fakta, som er *nye*, dvs. de som indeholde den nye tupel. Kombinationer af gamle fakta (dvs. udelukkende fra DB) behøver vi ikke checke, fordi vi har antaget, at ϕ gælder i DB. Altså kan vi nøjes med at

[3] Det er en svaghed ved vor definition, at den ikke indeholder et eksplicit krav om at ψ i en eller anden forstand skal være simplere eller nemmere at beregne end ϕ.

undersøge følgende forespørgsel (hvor vi ydermere har udnyttet symmetrien i ϕ_1), som vi vil kalde $\phi_1^{f(\mathit{børge},\mathit{peter})}$:

$$\leftarrow f(x, \mathit{peter}) \wedge x \neq \mathit{børge}$$

Dette er vel og mærket en formel, vi kan kontrollere i DB, så vi har altså, at $\phi_1^{f(\mathit{børge},\mathit{peter})}$ er en simplification af ϕ_1 med hensyn til updateringen $\{f(\mathit{børge}, \mathit{peter})\}$.

Lad os uformelt analysere tidsforbruget på evalueringen af de to formler. Hvis der ikke er en ualmindelig genial indeksering i databasen, kræver kontrol af ϕ_1 en størrelsesorden $\mathcal{O}(n^2)$ sammenligninger, hvorimod $\phi_1^{f(\mathit{børge},\mathit{peter})}$ kun indebærer $\mathcal{O}(n)$ sammenligninger.

Lad nu ϕ_2 være en anden af vores integritetsbegrænsninger:

$$\leftarrow f(x, y) \wedge (\neg e(x) \vee \neg e(y))$$

Her vi vi den simplificerede formel $\phi_2^{f(\mathit{børge},\mathit{peter})}$:

$$\leftarrow \neg e(\mathit{børge}) \vee \neg e(\mathit{peter})$$

Når vi gennemtrænger den logiske symbolik, kan vi se, at man skal kontrollere at $e(\mathit{børge})$ *og* $e(\mathit{peter})$ skal gælde i DB for at opdateringen er legal. Her ser vi at simplification medfører et fald i tidskompleksiteten fra $\mathcal{O}(n)$ til $\mathcal{O}(1)$, dvs. konstant tid.

Og hvad har det så med opdatering i relationelle databaser at gøre?

Vi diskuterer i forhold til eksemplet ovenfor.

Integritetsbegrænsninger, som vi har defineret dem her, er overordnede udsagn om databasens tilstand, og ville i SQL kunne implementeres som en såkaldt assertion. Den slags evalueres helt og totalt hver gang en af de relationer, den afhænger af, ændrer sig. Derfor bruger man meget sjældent assertions, da de er for ineffektive, og derfor må databaseprogrammøren bruge meget af sin kreativitet til at omsætte integritetsbegrænsningernes overordnede logiske udtryk til en række afledte betingelser, som skal kontrolleres, og disse skal så sættes på databaserelationerne som constraints eller tilføjes databasen som triggers.

Det store problem for programmøren er så at sikre at alle disse constraints og triggers tilsammen rent faktisk sørger for at integritetsbegrænsningerne altid er overholdt.

Et mål i vor forskning er at gøre det automatisk: Ideelt set angiver programmøren følgende:

- integritetsbegrænsninger, der kan referere vilkårligt til relationerne i databasen, som logiske udtryk (eller "relationelt-udtryk = $\emptyset$")
- angivelser af, hvilke relationer som skal kunne opdateres og på hvilken måde (indsætte, slette, ...)

Databasen skal så automatiske finde simplificerede formler og sørge for, at de bliver kaldt inden databasen bliver opdateret. Disse simplifications kan så enten lægges ind som triggers eller eksplicitte constraints i skemaet, eller bedre indkoorpereres i procedurer for opdatering, hvor man kan tage resultatet af integritetschecket med ind i brugerdialogen.

Tag som eksempel et forsøg på indsættelse af $f(børge, peter)$. Hvis nu f.eks. $\phi_2^{f(børge,peter)}$ opdagede et problem kunne man, hvis det f.eks. var $e(børge)$ som manglede i databasen skrive noget a la følgende ud til brugeren:

> "Du kan ikke indsætte $f(børge, peter)$, da der ikke gælder $e(børge)$.
> Skal jeg indsætte $e(børge)$ eller vil du prøve en anden $f(X, peter)$?"

Som vi har beskrevet det, kan man få det indtryk, at man er nødt til at konstruere simplication-formler hver gang der kommer et forslag til en opdatering, f,eks. $f(børge, peter)$, $f(børge, jane)$ eller .... Men bemærk, at vor symbolmanipulation var uafhængig af hvilke konstanter, der faktisk optrådte i den foreslåede opdatering. Vi kan med andre ord betragte *børge* og *peter* som *parametre*[4] i et generelt mønster for opdatering, dvs. som en slags variable, man kan udskifte med de aktuelt interessante værdier.

For at kunne realisere denne vision synes vi at mangel to ting:

- En generel procedure til, givet integritetsbegrænsninger og mønster for ønskede opdateringsmuligheder, at generere simplication-formler.
- En mekanisme til at håndtere rutiner eller forløb af afledte opdateringer, som bruger(dialog)en kan gå igennem for at få opnå konsistens (svarende til en vej gennem et beslutningstræ).

[4]Visse personer ville omtale *børge* og *peter* i denne kontekst som Skolem-konstanter

5 Et forslag til, hvordan man kan konstruere simplification-formler med nogle interessante transformationer, som også kan bruges til andre ting

Kære læser, jeg håber du kan tilgive mig at sproget skifter til engelsk, da jeg har sakset i et arbejdsnotat skrevet på dette sprog.

Først lidt ekstra notation; for de såkaldte "parameters" se diskussionen om *børge* og *peter* ovenfor:

The notation $\bar{a}$ indicates a sequence of terms and $p(\bar{a})$ an atom whose predicate has arity corresponding to the length of $\bar{a}$, and whose arguments are given by $\bar{a}$. In general, italic font is used for predicates $p, q, \ldots$, constants $a, b, \ldots$, and variables $x, y, \ldots$, however a special kind of constants called *parameters* are written in bold face, $\mathbf{a}, \mathbf{b}, \ldots$

We need the following notation for describing generic patterns for classes of updates.

Definition 1 *A* parameter *is a constant* $\mathbf{a}$ *not used in any database state. A* generic *update (request) is one containing zero or more parameters. For any expression E with parameters $\bar{\mathbf{a}}$ and $\bar{c}$ a sequence of constants of same length as $\bar{\mathbf{a}}$, the notation $E_{\bar{\mathbf{a}}/\bar{c}}$ refers to the expression that arise from E when each element of $\bar{\mathbf{a}}$ is replaced consistently by the matching element of $\bar{c}$; with an overloading of usage, $E_{\bar{\mathbf{a}}/\bar{c}}$ is called* an instance *of E. Whenever E is an expression with parameters $\bar{\mathbf{a}}$, we may indicate this by writing E as $E(\bar{\mathbf{a}})$ in which case $E(\bar{c})$ is used as an alternative notation for $E_{\bar{\mathbf{a}}/\bar{c}}$. An update (request) is called* ground *if it contains no parameters.*[5]

Two formulas α and β are equivalent up to instantiation of parameters, *written $\alpha \cong \beta$ whenever $\alpha' \equiv \beta'$ for all instances (α', β') of (α, β).*

Example 1 *Let $\mathbf{a}_1$, $\mathbf{a}_2$ be two distinct parameters and c_1, c_2 two distinct constants that are not parameters. Then we have $c_1 \neq c_2 \equiv true$ as well as $c_1 \neq c_2 \cong true$ but not $\mathbf{a}_1 \neq \mathbf{a}_2 \cong true$.*

In order to describe procedures for simplification and evaluation of integrity constraints, and update routines, we define a series of transformations on integrity constraints. In the following definitions, the letter ϕ intuitively refers to the fixed set of ICs associated with a given database whereas Γ is an arbitrary set of ICs (typically resulting from sequences of transformations applied to ϕ).

[5] It depends on the contexts whether a given ground update is considered generic (as generic updates occationally can be ground by definition). It is not so elegant when we below refer to "a (perhaps generic) update"...

5.1 Translation of ICs back and forth between different states

To consider whether or not a given property holds after a prospective update means to reason about the possible truth of a formula in a future state, but arguing in the present state. This resembles the so-called future studies.

We also need to consider the reverse. Reasoning about the truth of "past" formulas but arguing in the present state. This resembles historical studies.

One way to do such reasoning is to define a common language with modal operators referring to the different states. However, such a language is far more general what we need. Our methods are ment to be applied in contexts in which there is always one particular *current* state: We want to avoid testing formulas referring to other states, and in this way we ensure that the truth of a given formula always can be tested as a straightforward query formulated as, e.g., an SQL statement or a Prolog query.

Our way to handle the reasoning about future and past is by the introduction of operators to translate formulas between different states. More specifically, we define operators "after" and "before" below that satisfy the following properties:[6] The formula resulting from the transformation $\mathrm{after}^U(\Gamma)$ is defined in such a way that it holds in present state iff Γ holds in the state updated with U. We use the following straightforward definitions for the "after" and "before" translations by means of syntactic replacements.

Definition 2 *For given (perhaps generic) update request*

$$\begin{array}{rll} U & = \{ & p_1(\bar{a}_{1,1}), p_1(\bar{a}_{1,2}), \ldots, p_1(\bar{a}_{1,n1}), \\ & & p_2(\bar{a}_{2,1}), p_2(\bar{a}_{2,2}), \ldots, p_2(\bar{a}_{2,n2}), \\ & & \ldots \\ & & p_k(\bar{a}_{k,1}), p_k(\bar{a}_{k,2}), \ldots, p_k(\bar{a}_{k,nk})\}, \end{array}$$

where the p_i's are distinct, and set of ICs Γ, the notation $\mathrm{after}^U(\Gamma)$ *refers to the set of clauses Γ' found as follows.*

1. *Let Γ' consist of a copy of Γ in which all occurences of an atom of form $p_i(\bar{t})$ have been simultaneously replaced by $p_i(\bar{t}) \vee \bar{t} = \bar{a}_{i,1} \vee \cdots \vee \bar{t} = \bar{a}_{i,ni}$*
2. *Do the following as long as possible:*
   - *Replace any formula in Γ' of the form $\leftarrow A \wedge (B_1 \vee B_2) \wedge C$ by the two formulas $\leftarrow A \wedge B_1 \wedge C$ and $\leftarrow A \wedge B_2 \wedge C$.*

[6] Notice the difference to the definition of simplification. The identities of proposition 1 hold independently of whether the given formulas holds in the present state.

- *Replace any formula in* Γ' *of the form* $\leftarrow A \wedge \neg(B_1 \vee B_2) \wedge C$ *by the formula* $\leftarrow A \wedge \neg B_1 \wedge \neg B_2 \wedge C$.

The notation $\text{before}^U(\Gamma)$ *refers to the set of clauses* Γ' *found as follows.*

1. *Let* Γ' *consist of a copy of* Γ *in which all occurences of an atom of form* $p_i(\bar{t})$ *have been simultaneously replaced by* $p_i(\bar{t}) \wedge \bar{t} \neq \bar{a}_{i,1} \wedge \cdots \wedge \bar{t} \neq \bar{a}_{i,ni}$
2. *Do the following as long as possible:*
   - *Replace any formula in* Γ' *of the form* $\leftarrow A \wedge \neg(B_1 \wedge B_2) \wedge C$ *by the formula* $\leftarrow A \wedge (\neg B_1 \vee \neg B_2) \wedge C$.
   - *Replace any formula in* Γ' *of the form* $\leftarrow A \wedge (B_1 \vee B_2) \wedge C$ *by the two formulas* $\leftarrow A \wedge B_1 \wedge C$ *and* $\leftarrow A \wedge B_2 \wedge C$.

The intermediary formulas with disjunction operators resulting from step 1 are not clauses, but step 2 removes those in such a way that the remaining formulas are in the format for clauses that we use.

The essential properties of these transformations are given by the following proposition.

Proposition 1 *Assume a database state DB and a (perhaps generic) update U. For arbitrary set of clauses* Γ *we have that*

$$\begin{array}{rcl} DB \models \text{after}^U(\Gamma) & \cong & DB \cup U \models \Gamma \\ DB \cup U \models \text{before}^U(\Gamma) & \cong & DB \models \Gamma \end{array}$$

$$\Gamma \quad \equiv \quad \text{after}^U(\text{before}^U(\Gamma)) \quad \equiv \quad \text{before}^U(\text{after}^U(\Gamma))$$

In addition, we have the following trivial properties:

$$\begin{array}{rcl} \text{after}(\Gamma_1 \cup \Gamma_2) & = & \text{after}(\Gamma_1) \cup \text{after}(\Gamma_2) \\ \text{before}(\Gamma_1 \cup \Gamma_2) & = & \text{before}(\Gamma_1) \cup \text{before}(\Gamma_2) \end{array}$$

Finally, For any set of ICs Γ *and (perhaps generic) update* $U = U_1 \cup U_2$, *we have*

$$\begin{array}{rcccl} \text{after}^U(\Gamma) & = & \text{after}^{U_2}(\text{after}^{U_1}(\Gamma)) & = & \text{after}^{U_1}(\text{after}^{U_2}(\Gamma)) \\ \text{before}^U(\Gamma) & = & \text{before}^{U_2}(\text{before}^{U_1}(\Gamma)) & = & \text{before}^{U_1}(\text{before}^{U_2}(\Gamma)). \end{array}$$

Proof 1 *More or less trivial!*

Important notice: It seems possible to generalize these operators also to handle update by deletion: "after" for an update by removal $p(a)$ can be defined identical to "before" for update by adding $p(a)$. We leave this out for the moment.

Example 2 *For the IC* $\phi :\leftarrow p(x) \wedge q(x) \wedge \neg r(x)$ *we give the following examples of "after"-formulas;* $\mathbf{a}$ *and* $\mathbf{b}$ *are parameters.*

$$\begin{array}{rcl}
\text{after}^{p(\mathbf{a})}(\phi) & = & \phi \cup \leftarrow x = \mathbf{a} \wedge q(x) \wedge \neg r(x) \\
\text{after}^{q(\mathbf{b})}(\phi) & = & \phi \cup \leftarrow p(x) \wedge x = \mathbf{b} \wedge \neg r(x) \\
\text{after}^{r(\mathbf{a})}(\phi) & = & \leftarrow p(x) \wedge q(x) \wedge \neg r(x) \wedge \neg(x = \mathbf{a}) \\
\text{after}^{\{p(\mathbf{a}),r(\mathbf{b})\}}(\phi) & = & \quad\ \leftarrow p(x) \wedge q(x) \wedge \neg r(x) \wedge \neg(x = \mathbf{b}) \\
 & & \cup \ \leftarrow x = \mathbf{a} \wedge q(x) \wedge \neg r(x) \wedge \neg(x = \mathbf{b}) \\
\text{after}^{\{p(\mathbf{a}),r(\mathbf{a})\}}(\phi) & = & \quad\ \leftarrow p(x) \wedge q(x) \wedge \neg r(x) \wedge \neg(x = \mathbf{a}) \\
 & & \cup \ \leftarrow x = \mathbf{a} \wedge q(x) \wedge \neg r(x) \wedge \neg(x = \mathbf{a}) \\
\text{after}^{\{p(\mathbf{a}),q(\mathbf{b})\}}(\phi) & = & \quad\ \phi \\
 & & \cup \ \leftarrow x = \mathbf{a} \wedge q(x) \wedge \neg r(x) \\
 & & \cup \ \leftarrow p(x) \wedge x = \mathbf{b} \wedge \neg r(x) \\
 & & \cup \ \leftarrow x = \mathbf{a} \wedge x = \mathbf{b} \wedge \neg r(x)
\end{array}$$

Notice that when a predicate that appears in a positive literal is updated, after(ϕ) *contains a copy of* ϕ *and for a predicate that appears in a negative literal,* after(ϕ) *contain a formula that is obviously a specialization of (i.e., subsumed by)* ϕ *occurs.*

Examples of applications of "before" will appear later; typically "before" is applied to ICs that are results of other transformations — so repeating the example above, just replacing "after" by "before" would be a bit misleading.

5.2 Normalization of formulas

Sets of ICs produced by the "after" and "before" transformations are obviously not in any "reduced" or "normalized" form. For this purpose, we postulate two normalization procedures, one for reducing parameter-free ICs in a given database state, and another for processing generic ICs so to speak statically.

Definition 3 (Sketch) *For given set of parameter-free ICs and database state* DB*, the notation* $\text{normalize}^{DB}(\Gamma)$ *refers to a set of ICs with* $\text{normalize}(\Gamma)^{DB} \equiv \Gamma$ *in such a way that*

- $\text{normalize}^{DB}(\Gamma) =$ *true (or* $\emptyset$*) whenever* $DB \models \Gamma$ *,*
- $\text{normalize}^{DB}(\Gamma) =$ *false whenever* $DB \not\models \Gamma$*, and*
- $\text{normalize}^{DB}(\text{normalize}^{DB}(\Psi)) = \text{normalize}^{DB}(\Psi)$ *for any set of ICs* Ψ *and database state* DB*.*

For given set of ICs Γ *(perhaps with parameters), the notation* $\text{normalize}(\Gamma)$ *refers to a set of ICs with* $\text{normalize}(\Gamma) \cong \Gamma$ *in such a way that*

- $\text{normalize}(\Gamma) = $ *true (or* $\emptyset$*) whenever* $\Gamma \cong$ *true,*
- $\text{normalize}(\Gamma) = $ *false whenever* $\Gamma \cong$ *false, and*
- $\text{normalize}(\text{normalize}(\Psi)) = \text{normalize}(\Psi)$ *for any set of ICs* Ψ*.*

Notice that "normalize^{DB}" embeds a decision procedure for testing truth and falsity of ICs; this is a decidable property and there exists "normalize^{DB}" functions with the desired property. We are in doubt about the generic/static "normalize" procedure. Basically it embeds a decision procedure for satisfiability, which may imply som decidability problems. THIS MUST BE CHECKED, and if decidability is a problem, an occurrence of "whenever" should be replaced by "only when" above. In the following, we pretend for simplicity that everything is decidable.

We expect to describe concrete procedures at a later stage. However, for the examples we assume the following procedures being part of normalize.

Equation removal: If a clause contains an equation $x = t$, x a variable in the clause, remove the equation and replace all occurrences of x in the clause by t. Any equation of form $t = t$ is removed.

Nonquation removal: Any nonequation $A \neq B$, where A and B are two distinct non-parameters, is removed.

Normalize negations: Remove any occurrences of $\neg\neg$, replace any literal $\neg(A = B)$ by $A \neq B$ and $\neg(A \neq B)$ by $A = B$.

Detection of failed equations and nonequations: An IC is replaced by *true* if it contains

- $A = B$ and $A \neq B$ at the same time for arbitrary A and B,
- $A = B$ where A and B are nonunifiable and none of them parameters.
- $A \neq A$ for arbitrary A.

Folding by resolution: Two ICs $\leftarrow A \wedge B$ and $(\leftarrow \neg A \wedge B)\rho$, ρ a renaming substitution, are replaced by $\leftarrow B$.

It should be noticed, however, that it is not sufficient for a concrete normalize procedure to consider each IC in isolation as unsatisfiability can be caused by dependencies "across" the different ICs, e.g., as in $\{\leftarrow p, \leftarrow \neg p\}$.

We have the following properties.

Proposition 2 *Let DB be a database state, U a (perhaps generic) update and Γ a set of ICs. Then we have:*

$$\begin{array}{rcl} \text{normalize}^{DB\cup U}(\Gamma) & \cong & \text{normalize}^{DB}(\text{after}^{U}(\Gamma)) \\ \text{normalize}^{DB}(\Gamma) & \cong & \text{normalize}^{DB\cup U}(\text{before}^{U}(\Gamma)). \end{array}$$

Notice also:

Proposition 3

$$\text{normalize}^{DB}(\Gamma_1 \cup \Gamma_2) \quad \cong \quad \text{normalize}^{DB}(\Gamma_1) \cup \text{normalize}^{DB}(\Gamma_2)$$

5.3 Subsumption checks

Simplification of ICs is based on an assumption that the ICs hold in the present state. Thus, when testing the truth of formulas that are "obviously" implied by these ICs are obviously true and need not be considered by a procedure. For that purpose, we define the following.

Definition 4 *A clause ψ* subsumes *another ϕ (or, alternatively, ϕ is* subsumed by *ψ) whenever there is a substitution σ so that each literal of $\psi\sigma$ is contained in ϕ.*

Example 3 *The clause*

$$\leftarrow p(x,b) \land q(x,b) \land \neg r(x,b) \land x \neq a$$

is subsumed by

$$\leftarrow p(x,y) \land q(x,y) \land \neg r(x,y).$$

Proposition 4 *Assume IC ϕ is subsumed by IC ψ. If ψ is satisfied in a given database, so is ϕ.*

Proof 2 *Trivial.*

Proposition 5 *Assume (perhaps generic) IC ϕ is subsumed by (perhaps generic) IC ψ. If $\psi \cong$ true, we have also $\phi \cong$ true.*

Proof 3 *Needs to be checked.*

Transformation "Remove-subsumed" defined below corresponds to its name and is used to remove those derived ICs produced by other transformations, that are anyhow subsumed by the original ICs.

Definition 5 *For sets of ICs ϕ and Γ, the notation* $\text{Remove-subsumed}^{\phi}(\Gamma)$ *refers to the subset of Γ in which any IC subsumed by an IC in ϕ is removed.*

Proposition 6

$$\text{Remove-subsumed}^{\phi}(\Gamma) \cup \phi \equiv \Gamma \cup \phi$$

5.4 Putting together a simplification procedure

The following composition of transformations defines a procedure for simplification of ICs corresponding to previous works on simplification where updates always take place from consistent states.

Definition 6 *For set of ICs ϕ and generic update request U, we define*

$$\text{Simp}^{U}(\phi) = \text{Remove-subsumed}^{\phi}(\text{normalize}(\text{after}^{U}(\phi))).$$

The following property is central:

Proposition 7 *For any set of ICs ϕ and generic update request U,* $\text{Simp}^{U}(\phi)$ *is a simplification of ϕ wrt. U.*

Proof 4 *Seems trivial but should be written down.*

Example 4 *For the IC $\phi :\leftarrow p(x) \wedge q(x) \wedge \neg r(x)$ we have the following examples of simplified ICs;* $\mathbf{a}$ *and* $\mathbf{b}$ *are parameters.*

$$\begin{array}{rcl}
\text{Simp}^{p(\mathbf{a})}(\phi) & = & \leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}) \\
\text{Simp}^{q(\mathbf{b})}(\phi) & = & \leftarrow p(\mathbf{b}) \wedge \neg r(\mathbf{b}) \\
\text{Simp}^{r(\mathbf{a})}(\phi) & = & \mathit{true} \\
\text{Simp}^{\{p(\mathbf{a}),r(\mathbf{b})\}}(\phi) & = & \leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}) \wedge \mathbf{a} \neq \mathbf{b} \\
\text{Simp}^{\{p(\mathbf{a}),r(\mathbf{a})\}}(\phi) & = & \mathit{true} \\
\text{Simp}^{\{p(\mathbf{a}),q(\mathbf{b})\}}(\phi) & = & \begin{array}{l} \leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}) \\ \leftarrow p(\mathbf{b}) \wedge \neg r(\mathbf{b}) \\ \leftarrow \mathbf{a} = \mathbf{b} \wedge \neg r(\mathbf{a}) \end{array}
\end{array}$$

A detailed trace for the evaluation of these simplified formulas will show that Remove-subsumed *removes ϕ from the resulting set of formulas when a predicate with only positive occurrences are updated, and a nontrivial specialization of ϕ when a predicate with a negative occurrence is updated; compare with example 2.*

The last simplified formula in the example illustrates the following proposition that describes how simplification for sequences of updates can be split up into a combination of simplifications for simpler (e.g., singleton) updates. However, this is not a general property, cf. the conditions in the following property.

Proposition 8 *For set of ICs ϕ and (perhaps generic) update request $U = U_1 \cup U_2$, where the predicates of U_2 do not occur negated in ϕ, we have:*

$$\begin{array}{lcl} \mathrm{Simp}^U(\phi) & \cong & \quad \mathrm{Simp}^{U_1}(\phi) \\ & & \cup \ \mathrm{Simp}^{U_2}(\phi) \\ & & \cup \ \mathrm{Simp}^{U_2}(\mathrm{Simp}^{U_1}(\phi)) \end{array}$$

Proof 5 *Looks trivial but should be checked!!!*

The property can easily be generalized for arbitrarily long sequences of updates. It is interesting to notice that the proposition can be applied even in the case where the intermediate state $DB \cup U_1$ is inconsistent (but DB must of course be consistent).

Unfortunately, we do not have a straightforward compositionality property for the general case, but we can supply the following which, together with proposition 8 will cover a large portion of simplifications for a single IC.

Proposition 9 *For an IC ϕ and update request $U = U_1 \cup U_2$, where $U_2 = p(\bar{\mathbf{a}})$ and*

$$\mathrm{Simp}^{U_1}(\phi) = (\leftarrow B \wedge \neg p(\bar{t}))$$

with no occurrences of p in B, we have:

$$\mathrm{Simp}^U(\phi) \quad \cong \quad \leftarrow B \wedge \neg p(\bar{t}) \wedge \bar{t} \neq \bar{\mathbf{a}}$$

In other words, the effect of extending a given update (U_1) by adding a tuple for a predicate that occurs negatively in an IC is to specialize the simplification for U_1. Thus adding such tuples in may tend to make inconsistent intermediate states consistent, whereas adding tuples for "positive" predicates can lead to inconsistency.

Proof 6 *Looks trivial but should be checked!!!*

To the list of standard properties we may add this one:

Proposition 10

$$\mathrm{Simp}^U(\phi_1 \cup \phi_2) \quad \cong \quad \mathrm{Simp}^U(\phi_1) \cup \mathrm{Simp}^U(\phi_2)$$

Simplification can used in two ways that we have not distinguished clearly until now:

- **Dynamically:** At an arbitrary point in the lifetime of a database, the simplification procedures is called for the original set of ICs, and the simpler formulas can be executed as a query that is expected to run much faster than the original ones (in the updated state).

  It may also be interesting to simplify generic ICs dynamically in case new ICs arise during the database history, e.g., derived from the original ones taking the actual updates into account (this is shown in the following).

- **Statically:** The database designer defines what generic updates that are relevant. The ICs are simplified according to those generic updates (perhaps optimized further by other mean), and when a particular update arises, an instantiation of the relevant simplified IC is tested.

The following proposition captures the fact that the dynamic and static simplifications yield the same results.

Proposition 11 *Let $U(\bar{\mathbf{a}})$ be a generic update request with parameters $\bar{\mathbf{a}}$ and $\bar{c}$ any matching sequence of non-parameter constants, and Γ a set of ICs. Then*

$$\mathrm{Simp}^{U(\bar{c})}(\Gamma) \quad \equiv \quad [\mathrm{Simp}^{U(\bar{\mathbf{a}})}(\Gamma)]_{\bar{\mathbf{a}}/\bar{c}}.$$

With this, we can consider most results given for simplified ICs applicable to both static and dynamic contexts. As an example, we can rewrite proposition 8 using the indicated principle as follows — and we get a form showing how "precompiled" simplifications for generic updates apply in sequence for specific updates.

Proposition 12 *For set of ICs ϕ and generic update request $U(\bar{\mathbf{a}}, \bar{\mathbf{b}}) = U_1(\bar{\mathbf{a}}) \cup U_2(\bar{\mathbf{b}})$ where the predicates of U_2 do not occur negated in ϕ, we have, for any instances $U_1(\bar{a})$, $U_2(\bar{b})$ of $U_1(\bar{\mathbf{a}})$, $U_2(\bar{\mathbf{b}})$, that*

$$\begin{array}{rcl} [\mathrm{Simp}^{U(\bar{\mathbf{a}},\bar{\mathbf{b}})}(\phi)]_{\bar{\mathbf{a}},\bar{\mathbf{b}}/\bar{a},\bar{b}} & \equiv & [\mathrm{Simp}^{U_1(\bar{\mathbf{a}})}(\phi)]_{\bar{\mathbf{a}}/\bar{a}} \\ & \cup & [\mathrm{Simp}^{U_2(\bar{\mathbf{b}})}(\phi)]_{\bar{\mathbf{b}}/\bar{b}} \\ & \cup & [\mathrm{Simp}^{U_2(\bar{\mathbf{b}})}([\mathrm{Simp}^{U_1(\bar{\mathbf{a}})}(\phi)]_{\bar{\mathbf{a}}/\bar{a}})]_{\bar{\mathbf{b}}/\bar{b}} \end{array}$$

Notice that the last component of the right-hand side also could have been written $[\mathrm{Simp}^{U_2(\bar{\mathbf{b}})}(\mathrm{Simp}^{U_1(\bar{\mathbf{a}})}(\phi)]_{\bar{\mathbf{a}},\bar{\mathbf{b}}/\bar{a},\bar{b}}$; this form of the proposition is be relevant when a "precompiled" simplification for the compound generic update is given. However, in general we allow ourselves to think of all previous propositions (for which it is relevant) generalized in this way.

We end this section by showing how simplified formula to be applied after the update can be uptained.

Proposition 13 *Let DB be a consistent stated with ICs ϕ. Then the state $DB \cup U$ is consistent iff*

$$DB \cup U \quad \rightarrow \quad \text{normalize}(\text{before}^U(\text{Simp}^U(\phi))).$$

Example 5 *In example 4, we considered the IC $\phi :\leftarrow p(x) \wedge q(x) \wedge \neg r(x)$ and got the following;* $\mathbf{a}$ *and* $\mathbf{b}$ *are parameters.*

$$\begin{array}{rcl} \text{Simp}^{\{p(\mathbf{a}),q(\mathbf{b})\}}(\phi) & = & \leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}) \\ & & \leftarrow p(\mathbf{b}) \wedge \neg r(\mathbf{b}) \\ & & \leftarrow \mathbf{a} = \mathbf{b} \wedge \neg r(\mathbf{a}) \end{array}$$

Assuming the update $\{p(\mathbf{a}), r(\mathbf{b})\}$ performed to a consistent state, the updated state is consistent iff the following holds in it:

$$\begin{array}{l} \leftarrow q(\mathbf{a}) \wedge \mathbf{a} \neq \mathbf{b} \wedge \neg r(\mathbf{a}) \\ \leftarrow p(\mathbf{b}) \wedge \mathbf{a} \neq \mathbf{b} \wedge \neg r(\mathbf{b}) \\ \leftarrow \mathbf{a} = \mathbf{b} \wedge \neg r(\mathbf{a}) \end{array}$$

5.5 Incremental evaluation of ICs based on simplification

In case we only consider updates that lead from a consistent state to another consistent state it is straightforward to apply simplified formulas for validation of suggested updates; no notion of incrementality is relevant.

Here we consider the case where an update U_1 brings the database into an inconsistent state and we consider the problem of whether a subsequent update U_2 can restore consistency. As a first example, let us consider proposition 8, i.e., U_2 does not affect predicates that occur negated in the ICs ϕ. Here we have consistency of the updated database $DB \cup U_1 \cup U_2$ equivalent with the following condition:

$$\begin{array}{rcl} DB \models & & \text{Simp}^{U_1}(\phi) \\ & \cup & \text{Simp}^{U_2}(\phi) \\ & \cup & \text{Simp}^{U_2}(\text{Simp}^{U_1}(\phi)) \end{array}$$

This is in fact interesting: We intended that U_1 leads to inconsistency, and thus $DB \not\models \text{Simp}^{U_1}(\phi)$. The condition above implies that a U_2 with the indicated property cannot repare the consistency (since $\text{Simp}^{U_1}(\phi)$ appears as a conjunct above).

Let us now consider the conditions in proposition 9: $U_2 = q(\bar{\mathbf{b}})$ and

$$\text{Simp}^{U_1}(\phi) = (\leftarrow B \wedge \neg q(\bar{t}))$$

with no occurrences of q in B. Here we have consistency of $DB \cup U_1 \cup U_2$ expressed by:

$$DB \models \leftarrow B \wedge \neg q(\bar{t}) \wedge \bar{t} \neq \bar{\mathbf{b}}.$$

There are two ways to use this formula: Knowing that $DB \not\models \text{Simp}^{U_1}(\phi)$ we can avoid "physically" performing U_1 and reason about which $\bar{b}$ that makes the condition above hold. The "smart" way to do this is to formulate the checking of $DB \models \text{Simp}^{U_1}(\phi)$ as an SQL query for the following set $S = \{\bar{t} \mid \leftarrow B \wedge \neg q(\bar{t}) \equiv \mathit{true}\}$. Then consistency of $DB \cup U_1 \cup U_2$ for prospective $U_2 = q(\bar{\mathbf{b}})$ can be formulated as $S = \{\bar{\mathbf{b}}\}$.

Another way would be to perform the update (so that $DB \cup U_1$ becomes the current database) and reason about the following condition:

$$DB \cup U_1 \models \text{before}^{U_1}(\leftarrow B \wedge \neg q(\bar{t}) \wedge \bar{t} \neq \bar{\mathbf{b}})$$

What happends is that we translate an expression of consistency wrt. the semantics DB into another that expresses the same condition when evaluated wrt. semantics $DB \cup U_1$. The translated expression can be evaluated in "current state" $DB \cup U_1$ in order to test whether a prospective update will bring the database into a consistent state.

Let us now assume that $U_1 = p(\bar{\mathbf{a}})$ and $B = (B' \wedge \neg p(\bar{s}))$ with p not occurring in B'. Then we can evaluate "before" replacing $p(\bar{s})$ by $p(\bar{s}) \wedge \bar{s} \neq \bar{\mathbf{a}}$ and thus $\neg p(\bar{s})$ by $\neg p(\bar{s}) \vee \bar{s} = \bar{\mathbf{a}}$. Thus the condition splits up into the following two:[7]

$$DB \cup U_1 \models (\leftarrow B \wedge \neg q(\bar{t}) \wedge \bar{t} \neq \bar{\mathbf{b}}) \wedge (\leftarrow B' \wedge \bar{s} = \bar{\mathbf{a}} \wedge \bar{t} \neq \bar{\mathbf{b}})$$

Conclusion of this section: We have a nice way to describe incremental evaluation of ICs. The operators "Simp" and "before" are straightforward to implement and the resulting formulas are easy to test as SQL or Prolog queries. However, we still need more *general* statements characterizing how these operators work together — how optimal static optimization can be made in ICs for compound updates.

[7]The literal $\bar{s} = \bar{\mathbf{a}}$ in the second IC is outputted from the procedure as $\neg(\bar{s} \neq \bar{\mathbf{a}})$.