Abstract

Dit werk gaat over het chase algoritme. Dit is een belangrijk algoritme met veel toepassingen in verband met databases. Het idee achter dit algoritme is het nemen van een database instantie en het hierop toepassen van een verzameling voorwaarden voor deze database. We gaan in detail op de definitie, terminatie en toepassingen van dit algoritme.

We introduceren voldoende theorie om tot de definitie te komen. De nadruk van het werk ligt op de terminatie van het chase algoritme. In het algemene geval is deze terminatie onbeslisbaar. We delen deze terminatie op in verschillende gevallen en introduceren terminatie condities voor elk van deze gevallen.

Verder geven we ook een overzicht van verschillende toepassingen van de chase. Dit illustreert de belangrijkheid van het algoritme en van de terminatie condities. Het werk dient dus als een domein studie die een overzicht geeft van de terminatie condities en toepassingen van de chase.

Dankwoord

Graag zou ik de mensen bedanken die geholpen hebben om dit werk tot stand te brengen. Een thesis is een lang en zwaar werk en alleen zou ik het niet hebben kunnen doen.

In de eerste plaats wil ik mijn promotor prof. dr. Jan Van Den Bussche bedanken om mij de kans te geven deze masterproef te maken. Verder was hij ook altijd beschikbaar voor vragen of andere problemen. Ook voor de immense hoeveelheid feedback is hij zeer bedankt.

Dan wil ik ook mijn vrienden en familie bedanken voor de constante steun. Zij zorgden voor de nodige ontspanning (en het nodige eten) om dit werk tot een goed einde te brengen.

Inhoudsopgave

Hoofdstuk 1

Inleiding

Deze thesis gaat over het chase algoritme. Dit algoritme is een bekend algoritme in de databasetheorie met verschillende toepassingen. Wat mij vooral interesseerde aan dit algoritme is zijn verscheidenheid. Alhoewel het algemeen bekend is, wordt het algoritme toch telkens op een andere manier gedefinieerd en in verschillende gebieden toegepast.

1.1 De Chase

Het chase algoritme werkt op een database en een verzameling beperkingen voor deze database. Dit wil zeggen dat deze beperkingen bepaalde voorwaarden opleggen op de structuur en inhoud van de database. Het chase algoritme gaat dan de constraints uit de verzameling toepassen op de database totdat dit niet meer mogelijk is. Dit geeft dan als resultaat een database die voldoet aan de regels.

Een belangrijk aspect van dit algoritme is dat de terminatie niet verzekerd is.

1.2 Nut van de chase

De oorspronkelijke introductie van de chase was als een oplossing voor het implicatie-probleem. Later bleek echter dat de chase in zeer veel gebieden nut heeft. Zo vormt de chase de basis voor data exchange en data integration. De chase wordt ook gebruikt voor het optimaliseren van queries of om ervoor te zorgen dat een database voldoet aan bepaalde vereisten. Het wordt elke dag direct of indirect gebruikt door mensen die gebruik maken van databases.

1.3 Onderzoeksvraag?

Wat is de chase, wanneer eindigt dit algoritme en waarvoor wordt het gebruikt?

1.4 Doel van dit werk

Het doel van deze masterproef is om het verschillende werk in verband met de chase te verenigen. De literatuur in verband met de chase maakt immers gebruik van verschillende definities en spreekt over verschillende gebieden. In dit werk brengen we dit samen vertrekkende vanuit één algemene definitie.

Verder leggen we vooral de nadruk op de terminatie van de chase. Dit is immers zeer belangrijk als we gebruik willen maken van het chase algoritme. Als het algoritme niet eindigt geeft het uiteraard ook geen resultaat.

We kijken ook naar de verschillende toepassingen van de chase. Hiermee zien we het belang van de chase. De chase is immers belangrijk bij het oplossen van verschillende problemen in verband met databases.

Dit werkt toont het belang van de chase en het belang van de terminatie van deze chase aan.

1.5 Indeling van dit werk

We beginnen in Hoofdstuk 2 met de introductie van de nodige theorie. Dit hoofdstuk bevat de nodige vereisten zodat we in Hoofdstuk 3 de definitie van de chase kunnen geven.

Na de definitie van de chase bekijken we de terminatie. Het bespreken van deze terminatie is verdeeld in verschillende hoofdstukken. We beginnen met het opdelen van de terminatie in verschillende klassen. Hierna bespreken we deze klassen afzonderlijk. Zowel Hoofdstuk 5 als Hoofdstuk 6 bespreken de verschillende terminatiecondities voor één van deze klassen. In Hoofdstuk 7 bespreken we de laatste twee van deze klassen.

We gaan dan over tot het bespreken van de toepassingen van de chase in Hoofdstuk 8. We beginnen met het bespreken van meer academische toepassingen zoals het oplossen van het implicatieprobleem. Hierna bekijken we meer praktische toepassingen zoals data exchange, data integration en query optimization. We bekijken ook twee systemen voor data exchange en data integration en bestuderen hoe zij gebruik maken van de chase.

Na de toepassingen bekijken we kort de implementatie van de chase. We bespreken onze eigen implementatie en mogelijke verbeteringen hiervoor. Om af te sluiten geven we nog een conclusie.

Hoofdstuk 2

Databasetheorie

In dit hoofdstuk bestuderen we de begrippen die nodig zijn om het chase algoritme te definiëren en te bespreken.

2.1 Formele definitie

We werken met het standaard relationele model voor databases. De formele definitie bepaalt een schema dat aangeeft hoe de data georganiseerd is binnen de database, dit noemen we het *databaseschema*. We moeten echter ook voor iedere relatie weten hoe deze eruit ziet, hiervoor maken we gebruik van een *relatieschema*. Vermits een database een verzameling relaties is, zal een databaseschema dan ook een verzameling van relatieschema's bevatten.

We beschouwen nu een eerste mogelijke formele definitie.

2.1.1 Named perspective

We beginnen met het definiëren van een relatieschema. Een relatieschema S is een eindige verzameling attributen.

We definiëren dan een databaseschema S als een eindige verzameling koppels van de vorm (R, S) met de volgende onderdelen:

- R: relatienaam
- S: relatieschema

Verder eisen we dat S een functie is, dit heeft als gevolg dat er geen twee koppels een verschillend relatieschema kunnen hebben voor eenzelfde relatienaam.

Als we S bekijken als een functie, komt het domein $dom(\mathsf{S})$ overeen met de verzameling relatienamen. In de context van het relationele model kunnen we dit duidelijker noteren als $rel(\mathsf{S})$. We kunnen dan een relatie $R \in rel(\mathsf{S})$ afkorten als $R \in \mathsf{S}$. We geven een voorbeeld van een dergelijke databaseschema in Voorbeeld 2.1.1.

Merk op dat we bij deze definitie gebruik maken van de namen van de attributen, deze zitten namelijk in het relatieschema. Als we werken met attribuutnamen zeggen we dat we gebruik maken van het *named perspective*, later in dit hoofdstuk volgt ook nog een definitie van het *unnmaned perspective* waarbij de attributen geen naam krijgen.

Voorbeeld 2.1.1. *We bekijken een databaseschema* S *met drie relaties* R_1, R_2, R_3. $\mathsf{S} = \{(R_1, \{A, B\}), (R_2, \{A, B, C\}), (R_3, \{C, D, \})\}$

We kunnen dan zien dat $\mathsf{S}(R_1) = \{A, B\}$ *aangezien* S *een functie is.*

2.1.2 Instance van named perspective

We willen uiteraard deze database gevuld zien met informatie, daarom introduceren we een *instance* van de database. Dit is een instantie van de database waarbij de relaties gevuld worden met waarden in overeenstemming met hun relatieschema. We zullen dit begrip nu meer formeel definiëren.

Veronderstel dat we beschikken over een universum $\mathbb{U}$ en een databaseschema S, dit universum kan bepaalde predicaten of functies hebben. Bijvoorbeeld $\mathbb{Q}, \leq, +$. Dit geeft als universum de verzameling van de rationale getallen $\mathbb{Q}$ met als predicaat $\leq$ en als functie de optelling. We kunnen de elementen van dit universum dan vergelijken via het $\leq$ predicaat en we kunnen de optelling erop toepassen. Merk op dat we meestal geen gebruik gaan maken van deze predicaten of functies.

Voor een gegeven relatieschema S uit een databaseschema S en over een universum $\mathbb{U}$ definiëren we een tupel van type S. Dit is een functie van S naar $\mathbb{U}$ die aan elke attribuut uit relatieschema S een waarde uit universum $\mathbb{U}$ toekent.

Voorbeeld 2.1.2. *We geven een voorbeeld van een tupel van type* S. *We hergebruiken het databaseschema* $\mathsf{S} = \{(R_1, \{A, B\}), (R_2, \{A, B, C\}), (R_3, \{C, D, \})\}$ *Hieruit kiezen we het relatieschema bepaald door* $\mathsf{S}(R_1)$, *dus* $\{A, B\}$.

Een voorbeeld van een tupel over dit relatieschema ziet er dan als volgt uit:
$\{(A, 8), (B, 40), (C, 22)\}$

We kunnen tupels van deze vorm op verschillende manieren noteren. Een eerste manier is het volledig noteren van de functie zoals in Voorbeeld 2.1.2, dit geeft ons $\{(A, 8), (B, 40), (C, 22)\}$. We kunnen dit echter ook noteren op een manier die analoog is aan de manier waarop dergelijke datastructuren genoteerd worden in populaire programmeertalen: $(A : 8, B : 40, C : 22)$. In deze tekst gaan we meestal de afgekorte vorm gebruiken waarbij de attributen weggelaten worden maar verondersteld wordt dat de volgorde van de tupel aangeeft met welk attribuut een element van de tupel overeenkomt. Dit ziet er dan als volgt uit: $(8, 40, 22)$.

Met behulp van de definitie van een tupel van type S kunnen we nu overgaan tot de definitie van een relatie-instance. Voor dit gegeven universum en een relatieschema S uit het databaseschema S definiëren we een relatie-instance s van S over $\mathbb{U}$ als een eindige verzameling tupels van type S over $\mathbb{U}$

We spreken dan over een database-instance I van S over $\mathbb{U}$. I is een functie op $dom(\mathsf{S})$ oftewel $rel(\mathsf{S})$ (de relaties uit S) die aan elke $R \in \mathsf{S}$ een relatie-instance van $\mathsf{S}(R)$ over $\mathbb{U}$ toekent.

Als het duidelijk is over welk universum er gesproken wordt en dit universum verandert niet dan kunnen we deze definities afkorten door de qualificatie " over $\mathbb{U}$ " telkens weg te laten.

We geven een voorbeeld van een instance:

Voorbeeld 2.1.3. *Neem* $\mathsf{S} = \{(R_1, \{A, B\})(R_2, \{C, D, E\})\}$ *en* $\mathbb{U} = \mathbb{N}$.
Een voorbeeld van een database-instance I *is als volgt:* $I(R_1) = \{(A : 1, B : 2), (A : 2, B : 3)\}$ *en* $I(R_2) = \{(C : 1, D : 2, E : 3), (C : 4, D : 5, E : 6), (C : 7, D : 8, E : 9)\}$

2.1.3 Unnamed perspective

In het unnamed perspective gaan we uit van eenzelfde structuur, we hechten echter geen belang meer aan de namen van de attributen. Een relatieschema geeft dan enkel de ariteit van de relatie aan, we weten dus wel hoeveel attributen er zijn maar niet hoe deze benoemd worden.

We definiëren weerom een databaseschema S als een eindige verzameling koppels van de vorm (R, ar) met de volgende onderdelen:

- R: relatienaam
- ar: een natuurlijk getal dat de ariteit van de bijhorende relatie aangeeft.

Aangezien S een functie is, komt $dom(\mathsf{S})$ overeen met de verzameling relatienamen. We kunnen dit dus ook noteren als $rel(\mathsf{S})$. In het unnamed persepective is $\mathsf{S}(R)$ de ariteit van relatie R in databaseschema S.

Merk op dat het unnamed perspective een speciaal geval is van het named perspective, namelijk waar elk relatieschema van de vorm $\{1, 2, 3, \ldots, ar\}$ is. We gebruiken dus natuurlijke getallen als attributen. We noemen de getallen $\{1, \ldots, ar\}$ dan ook de attributen van een relatie R of de kolomnummers van R.

Een voorbeeld van een databaseschema in het unnamed perspective vinden we in Voorbeeld 2.1.4.

Voorbeeld 2.1.4. *We hernemen het databaseschema* S *uit Voorbeeld 2.1.1. In het unnamed perspective gaat dit er dan als volgt uitzien:*

$\mathsf{S} = \{(R_1, 2), (R_2, 3), (R_3, 2)\}$

2.1.4 Instance van unnamed perspective

Ook van een databaschema S in het unnamed perspective willen we een instance kunnen aanmaken. Dit is analoog aan de definitie van een instance in het named perspective. We spreken dus opnieuw over een database-instance I, een functie op de relaties uit S die aan elke relatie $R \in \mathsf{S}$ een relatie-instance s van $\mathsf{S}(R)$ over $\mathbb{U}$ toekent.

Het verschil in de definitie schuilt in de relatie-instance s, dit wordt weerom een eindige verzameling tupels over $\mathbb{U}$, echter niet van type S maar van ariteit $\mathsf{S}(R)$ aangezien in het unnamed perspective het relatieschema vervangen is door een ariteit.

Een dergelijk tupel van bijvoorbeeld ariteit drie ziet er dan als volgt uit:
$\{(1,8),(2,40),(3,22)\}$. Afgekort krijgt dit dus de volgende vorm $(8,40,22)$.

We geven ook een voorbeeld van een instance in het unnamed perspective namelijk Voorbeeld 2.1.5

Voorbeeld 2.1.5. *We geven een analoog voorbeeld aan Voorbeeld 2.1.3, merk op dat er een verschil is in de definitie van het databaseschema maar dat eenzelfde instantiatie wel mogelijk is. We gebruiken hier ook de afgekorte notatie van de tupels.*

Neem $\mathsf{S} = \{(R_1,2)(R_2,3)\}$ *en* $\mathbb{U} = \mathbb{N}$.
Een voorbeeld van een database-instance I *is als volgt:* $I(R_1) = \{(1,2),(2,3)\}$ *en* $I(R_2) = \{(1,2,3),(4,5,6),(7,8,9)\}$

2.1.5 Keuze van perspectief

Beide perspectieven zijn equivalent en alle onderdelen uit de studie van het relationeel model kunnen uitgevoerd worden op eender welk perspectief. Er zullen echter kleine verschillen zijn bij bepaalde definities ten gevolge van het verschil in perspectief, dit zal echter niet leiden tot een verschil in betekenis.

We kiezen er voor om gebruik te maken van het het unnamed perspective aangezien dit het meeste aansluit bij de standaarden van de theoretische informatica.

2.2 Afkortingen

Als de ariteiten van de verschillende relatienamen niet van belang zijn kunnen we een databaseschema afkorten door deze weg te laten.

Voorbeeld 2.2.1. $\mathsf{S} = \{(R_1,2),(R_2,3)\}$ *kan afgekort worden naar* $\mathsf{S} = \{R_1,R_2\}$

Verder introduceren we nog de notatie $t[A]$ in verband met tupels. Voor een bepaalde tupel t en een verzameling attributen A is $t[A]$ de beperking van t op A. We geven een voorbeeld van een dergelijke beperking.

Voorbeeld 2.2.2. *Bekijk de tupel* $t = (8,40,22)$. *We weten dat dit de afkorting is voor* $\{(1,8),(2,40),(3,22)\}$. *Neem* $A = \{1,3\}$ *dan is* $t[A] = \{(1,8),(3,40)\}$.

2.3 Dependencies

Dependencies zijn eerste orde logica formules van de vorm:

$$\forall x_1 \ldots \forall x_n[\phi(x_1,\ldots,x_n) \rightarrow \exists z_1 \ldots z_k \psi(y_1,\ldots,y_m)]$$

met :

- $\{z_1,\ldots,z_k\} = \{y1,\ldots,y_m\} - \{x_1,\ldots,x_n\}$
- ϕ een conjunctie van atomaire formules (mogelijk leeg). Dit wordt ook wel de Head genoemd.

- ψ een niet-lege conjunctie van atomaire formules . Dit wordt ook wel de Body genoemd.

Deze atomaire formules zijn ofwel *relatie-atomen* van de vorm $R(w_1, \ldots, w_n)$ met R de naam van een relatie met ariteit n of *gelijkheid atomen* van de vorm $w = w'$

We maken een onderscheid tussen full dependencies en embedded dependencies.

Een dependency is *full* als deze geen existentiële kwantoren bevat. En dus is een dependency *embedded* als dit wel het geval is.

Verder zijn er twee soorten dependencies *tuple generating dependencies* (TGD's) en *equality generating dependencies* (EGD's). Deze worden verder uitgelegd in de volgende paragrafen.

Ook is het belangrijk een onderscheid te maken tussen typed en untyped dependencies: Een dependency is *typed* als er een assignement van variabelen is aan de kolomposities zodat:

1. variabelen in relatie atomen alleen voorkomen op hun aangewezen positie
2. elk gelijkheidsatoom een paar van variabelen beschrijft die toegewezen zijn aan eenzelfde positie

Intuïtief betekent dit dat elke variabele een bepaald type heeft en dus niet op posities gebruikt mag worden die niet van eenzelfde type zijn of in gelijkheidsoperaties met andere variabelen die niet van hetzelfde type zijn.

De *untyped* dependencies zijn dan de dependencies waarin dit wel mag. Een voorbeeld van een typed en een untyped dependency geven we in Voorbeeld 2.3.1

Voorbeeld 2.3.1. *We geven een voorbeeld dat het verschil tussen typed en untyped illustreert: Stel dat we willen uitdrukken dat een relatie R transitief is, dan zouden we dit kunnen doen via de volgende dependency*

$$\forall x \forall y \forall z [R(x, y) \wedge R(y, z) \rightarrow R(x, z)]$$

Dit is echter geen typed dependency aangezien y zowel op de eerste als tweede positie van relatie R gebruikt wordt.

2.3.1 Tuple generating dependencies

Dit is een speciaal geval van de algemene vorm van dependencies gedefinieerd in de vorige paragraaf. Een *tuple genereating dependency* (TGD) is een dependency waarbij zowel de Head als de Body geen gelijkheidsatomen bevatten.

Voorbeeld 2.3.2. $\forall x, y, z(R(x, x, y) \wedge R(y, y, z) \rightarrow T(x, z))$

Speciale gevallen: join dependency

Een speciaal geval van een TGD is een join dependency(JD). Intuïtief geeft deze dependency aan dat voor bepaalde tupels uit een relatie R de join van deze tupels ook in R moet zitten.

Beschouw een databaseschema S dat de relatie R bevat. Neem $A_1, \ldots, A_n$ verzamelingen van attributen van R. We noteren een JD als volgt:

$$R : A_1 \bowtie \ldots \bowtie A_n$$

Een instantie I voldoet aan deze JD als voor elk stel tupels $t_1 \ldots t_n$ van $I(R)$ met $t_i[A_i \cap A_j] = t_j[A_i \cap A_j]$ voor $1 \leq i, j \leq n$ er een tupel t in $I(R)$ bestaat zodat $t[A_i] = t_i[A_i]$ voor elke $i \in \{1, \ldots, n\}$.

We geven een voorbeeld van een JD en een equivalente TGD:

Voorbeeld 2.3.3. *Beschouw een relatie R met ariteit 3. We bekijken de JD $\{1,2\} \bowtie \{2,3\}, \bowtie \{1,3\}$. De TGD die dezelfde dependency uitdrukt, ziet er dan als volgt uit :*

$$\forall x_1, x_2, y_1, y_2, z_1, z_2 (R(x_1, y_1, z_2) \wedge R(x_2, y_1, z_1) \wedge R(x_1, y_2, z_1) \rightarrow R(x_1, y_1, z_1))$$

Merk op dat de tupels niet volledig bepaald hoeven te zijn door de join dependency. We kunnen ook een embedded JD maken. Hiervan geven we een voorbeeld.

Voorbeeld 2.3.4. *Beschouw een relatie R met ariteit 4. We kunnen dan de JD $\{1,2\} \bowtie \{2,3\}$ beschouwen. Het valt meteen op dat kolom 4 niet voorkomt in deze JD, het is dus duidelijk een embedded JD. Als we deze JD omzetten naar een TGD krijgen we het volgende resultaat:*

$$\forall x, y, z, w, x_1, z_1, w_1 (R(x, y, z, w) \wedge R(x_1, y, z_1, w_1) \rightarrow \exists w_2 R(x, y, z_1, w_2))$$

We zien dat dit inderdaad een embedded TGD is aangezien er een existentiële kwantor aanwezig is.

Speciale gevallen: inclusion dependency

Inclusion dependencies(IND) zijn dependencies waarbij een deel van de ene relatie vervat zit in de andere relatie. Deze zijn van de vorm:

$$R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$$

Hierbij zijn:

- R, S (mogelijk identieke) relatienamen uit een databaseschema S
- $A_1, \ldots, A_n$ een geordende lijst attributen van R
- $B_1, \ldots, B_n$ een geordende lijst attributen van S

Merk op dat de volgorde van de attributen in zowel $A_1 \ldots A_n$ als $B_1 \ldots B_n$ van belang is en dat ze eenzelfde aantal attributen hebben.

Een database-instance I voldoet aan een inclusion dependency

$$R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$$

als voor elk tupel t_R van $I(R)$ er een tupel t_S in $I(S)$ bestaat zodat $t_S[B_1 \ldots B_n] = t_R[A_1 \ldots A_n]$

2.3.2 Equality generating dependencies

Equality generating dependencies (EGD's) hebben als vorm een body die voldoet aan dezelfde voorwaarden als TGD's maar met een head dat bestaat uit een enkele gelijkheid.

Voorbeeld 2.3.5. $\forall x, y, z(R(x, x, y) \wedge R(y, y, z) \rightarrow x = z)$

Speciale gevallen: functional dependency

Een speciaal geval van een EGD is een Functional Dependency(FD), deze is van de vorm

$$R : X \rightarrow Y$$

hierbij is $R \in \mathsf{S}$ met S een databaseschema en $X, Y \subseteq \{1, \ldots, \mathsf{S}(R)\}$ oftewel X, Y zijn verzameling van attributen van relaties R. Zowel X als Y kan dus uit meerdere attributen bestaan.

We geven een voorbeeld van een FD in het unnamed perspective en een equivalente EGD:

Voorbeeld 2.3.6. *Beschouw een relatieschema R met ariteit 3. We bekijken de FD $R : 1 \rightarrow 2$. Een EGD die dezelfde dependency uitdrukt, ziet er dan als volgt uit:*

$$\forall x_1, y_1, y_2, z_1, z_2(R(x_1, y_1, z_1) \wedge R(x_1, y_2, z_2) \rightarrow y_1 = y_2)$$

2.3.3 Afkortingen en begrippen

We kunnen de notatie van een dependency afkorten door de universele kwantoren weg te laten. We veronderstellen dan dat alle variabelen universeel gekwantificeerd zijn tenzij ze expliciet existentieel gekwantificeerd zijn. Een TGD $\sigma = \forall x_1, x_2, x_3(R(x_1, x_2) \wedge R(x_3, x_2) \rightarrow \exists y\ R(x_2, y))$. kan dan afgekort worden naar $\sigma' = R(x_1, x_2) \wedge R(x_3, x_2) \rightarrow \exists y\ R(x_2, y)$.

Verder introduceren we $\overline{x}$ als afkorting voor een tupel $(x_1, \ldots, x_n)$ van willekeurige lengte. We kunnen dan een dependency σ beperken, genoteerd als $\sigma(\overline{x})$. Voor een $\sigma = \forall \overline{x}(Body \rightarrow Head)$ kunnen we deze beperken tot $\overline{b}$. Het resultaat hiervan is $\sigma(\overline{b}) = \forall \overline{x}(Body(\overline{x} \setminus \overline{b} \rightarrow Head(\overline{x} \setminus \overline{b}))$.

Een ander begrip is een *positie* in een relatie-atoom. Bijvoorbeeld een relatie-atoom $R(a, b)$ heeft twee posities a genoteerd met $R.1$ en b genoteerd met $R.2$.

We kunnen dit begrip dan uitbreiden naar een dependency. De verzameling posities van een dependency is dan de unie van de posities van de relatie-atomen die voorkomen in de dependency. Voor een TGD $\sigma = R(x_1, x_2) \wedge S(x_1) \rightarrow R(x_2, x_1)$ geeft dit dus posities $\{R.1, R.2, S.1\}$.

2.4 Tableau

Een tableau is een instance over een uitgebreid universum. We introduceren zowel variabelen als constanten. Hiervoor gebruiken we het gekende universum $\mathbb{U}$ van constanten

1	2	3
4	5	6
7	8	9

Figuur 2.1: *Een voorbeeld van een tableau zonder variabelen*

1	x_2	x_0
x_2	5	6

Figuur 2.2: *Een voorbeeld van een tableau met variabelen, de variabelen worden voorgesteld door middel van x-en.*

en een nieuwe universum $\mathbb{V}$ van variabelen. Een tableau wordt dan een instance over het universum bepaald door $\mathbb{U} \cup \mathbb{V}$. Belangrijk hierbij is dat $\mathbb{U} \cap \mathbb{V} = \emptyset$, er bestaat dus geen twijfel of een bepaald element van het tableau een variabele of een constante is.

We geven een voorbeeld van een tableau dat overeenkomt met relatie R_2 uit Voorbeeld 2.1.5, dit is te zien in Figuur 2.1. Verder geven we ook een voorbeeld waarbij er variabelen toegevoegd zijn aan het tableau. Dit is te zien in Figuur 2.2. We geven ook een voorbeeld van een instance van een database met meerdere relaties, dit is te zien in Figuur 2.3.

Voor een tableau T met dergelijke variabelen uit $\mathbb{V}$ introduceren we de notatie $var(T)$: de verzameling variabelen uit T.

Verder introduceren we het begrip *valuation*. Een valuation van een tableau T is een functie $v : var(T) \to \mathbb{U}$. Deze functie genereert dus een mapping van variabelen uit T op constanten uit universum $\mathbb{U}$. We noemen $v(T)$ de instance die je krijgt door elke variabele x in T te vervangen door zijn beeld $v(x)$.

Als we een valuation v van T hebben voor een database-instance I dan noemen we v een *matching* van T in I als $v(T) \subseteq I$.

Een valuation kan dus gebruikt worden om van een tableau met variabelen uit $\mathbb{V}$ te gaan naar een tableau met enkel elementen uit $\mathbb{U}$. Dit resultaat komt dus overeen met onze originele definitie van een database-instance, deze bevat immers enkel elementen uit $\mathbb{U}$.

Voor het vinden van verbanden tussen gelijkvormige tableaus introduceren we het begrip *homomorfisme*. Voor twee tableaus T_1 en T_2 noemen we een functie $h : var(T_1) \to var(T_2)$ een homomorfisme van T_1 naar T_2 als $h(T_1) \subseteq T_2$. Hierbij is het resultaat van het toepassen van een homomorfisme h op een tableau T, genoteerd als $h(T)$ het toepassen van de functie h op elke tupel uit T. De constanten in beide tableaus blijven dus ongewijzigd. h zal enkel variabelen uit T_1 afbeelden op variabelen uit T_2.

Het idee achter een homomorfisme is het vinden van een patroon. We willen controleren

R		
1	2	3
x_1	x_2	x_3

S	
1	2
x_1	x_2

Figuur 2.3: *Een voorbeeld van een tableau over een relatie R met ariteit 3 en S met ariteit 2.*

of we het patroon gegeven door T_1 kunnen vinden in T_2. Als we een homomorfisme h vinden dan weten we dat er variabelen zijn in T_1 die we kunnen afbeelden op variabelen uit T_2 zodat $h(T_1) \subseteq T_2$. In dit geval komt het patroon dat overeenkomt met T_1 dus voor in in T_2. We geven hiervan een voorbeeld in Voorbeeld 2.4.1.

Om plaats te besparen zullen we een tableau soms ook noteren als de verzameling van alle tupels in dit tableau.

Voorbeeld 2.4.1. *We bekijken twee tableaus T_1, T_2 over een relatie R met ariteit 2. We bekijken een tableau T_1 van de volgende vorm:*

x_1	x_2
x_2	x_1

In onze verkorte notatie zouden we zeggen dat $T_1 = \{R(x_1, x_2), R(x_2, x_1)\}$. Verder bekijken we een tableau T_2 van de volgende vorm:

y_1	y_2
y_2	y_1
y	y

We gaan dus kijken of we het symmetrische patroon gegeven door T_1 kunnen vinden in T_2. Een homomorfisme h dat meteen duidelijk zichtbaar is, is $h = \{(x_1, y_1), (x_2, y_2)\}$. Door het toepassen van h op T_1 krijgen we $h(T_1)$:

y_1	y_2
y_2	y_1

We zien dat $h(T_1) \subseteq T_2$ en h is dus een homorfisme van T_1 naar T_2. Dit tableau kan ook verkregen worden door gebruik te maken van $h' = \{(y_1, x_2), (y_2, x_1)\}$. h' is dus ook een homomorfisme van T_1 naar T_2.

We hoeven echter niet elke variabele uit T_1 om te zetten naar een variabele uit T_2. We kunnen ook verschillende variabelen uit T_1 afbeelden op eenzelfde variabele uit T_2. Een voorbeeld hiervan is $h'' = \{(x_1, y), (x_2, y)\}$. Als we dit toepassen krijgen we $h''(T_1)$:

y	y

Ook in dit geval zien we dat $h''(T_1) \subseteq T_2$. h'' is dus een homorfisme van T_1 naar T_2.

2.5 Dependencies in tableauvorm

Merk op dat we de body van een dependency ook altijd kunnen voorstellen als een tableau. We kunnen de conjunctie van relatie atomen immers bekijken als een opsomming van tupels. Deze tupels bevatten constanten en variabelen en kunnen dus perfect gebruikt worden voor de opbouw van een tableau. Voor elke relatie R zal de relatie-instance de verzameling zijn van

Aangezien de head echter een gelijkheid kan zijn, kunnen we deze niet voorstellen door middel van een tableau. Het tableau is dus de verzameling van alle relatie-atomen uit de body van de dependency voor elke relatie.

Voorbeeld 2.5.1. *We kunnen bijvoorbeeld kijken naar een EGD σ over een relatie R*

met ariteit 3.

$$\sigma = \forall x, y, z, z_2(R(x, y, z) \wedge R(x, y, z_2) \rightarrow z = z_2)$$

De tableau die overeenkomt met de body van σ ziet er dan als volgt uit:

x	y	z
x	y	z_2

We geven ook een voorbeeld van een TGD σ_2 over R, deze TGD spreekt ook over S een relatie met ariteit 2.

$$\sigma_2 = R(x_1, x_2, x3) \wedge S(x_1, x_2) \rightarrow S(x_2, x_3)$$

R		
x_1	x_2	x_3

S	
x_1	x_2

2.6 Logische implicatie

We willen nu voor onze tableaus kunnen controleren of deze voldoen aan een bepaalde dependency of niet. We willen dus weten of een dependency waar of onwaar is voor een bepaald tableau. Om dit te doen introduceren we de logische implicatie $\models$ voor tableaus en dependencies. Voor een dependency σ en een tableau T zeggen we dan dat $T \models \sigma$ of T voldoet aan σ als :

- Als σ een TGD is dan $T \models \sigma$ als en slechts als voor elk homomorfisme h van de body van σ naar T geldt dat er een homomorfisme h' is zodat $h \subseteq h'$ en het resultaat van de toepassing van h' op de head van σ zit in T.
- Als σ een EGD is met als head $x_i = x_j$ dan $T \models \sigma$ als en slechts als voor elk homomorfisme h van de body van σ naar T geldt dat $h(x_i) = h(x_j)$.

We kunnen de $\models$ operator ook gebruiken voor een verzameling dependencies Σ en een tableau T. We zeggen dan dat $T \models \Sigma$ als en slechts als voor elke $\sigma \in \Sigma$ geldt dat $T \models \sigma$.

We kunnen ook het verband bekijken tussen een verzameling dependencies Σ en een dependency $\sigma \notin \Sigma$. Dit geeft dan aan of σ een logisch gevolg is van de dependencies uit Σ. De logische implicatie $\Sigma \models \sigma$ geldt als voor alle tableaus T : $T \models \Sigma \implies T \models \sigma$. Intuïtief wil dit zeggen dat de logische implicatie geldt als alle mogelijke tableaus die voldoen aan Σ ook voldoen aan σ.

We kunnen een onderscheid maken tussen het eindige en het oneindige geval. In het geval van eindige tableaus gaat het ook over eindige implicatie, dit noteren we als $\models_{fin}$. Als we echter ook oneindige tableaus toelaten dan kunnen we ook de oneindige logische implicatie bekijken, genoteerd als $\models_{unr}$.

De operator $\models$ geeft aan of er een semantisch verband is. We kijken immers naar de betekenis van de regels en testen of deze voldaan zijn. We kunnen echter ook kijken naar een syntactisch verband. Hiervoor gebruiken we de operator $\vdash$. We noemen dit een syntactisch verband omdat dit geldt zodra er een bewijs is, er hoeft geen interpretatie van de formules gedaan te worden.

Formeel zeggen we dat een formule σ bewijsbaar is vanuit een verzameling formules Σ, genoteerd als $\Sigma \vdash \sigma$ als er een formeel bewijs bestaat van σ vanuit Σ. Merk op dat Σ een verzameling is van eerste orde logica formules en dat σ ook een eerste orde logica formule is. We kunnen $\vdash$ dus overnemen uit de eerste orde logica.

Hoofdstuk 3

De Chase

3.1 Inleiding

De chase is een algoritme dat gebruikt kan worden om te controleren of een bepaalde verzameling dependencies Σ een dependency σ als logisch gevolg heeft. De chase is geïntroduceerd in 1979 als een oplossing voor het implicatieprobleem door zowel Aho, Beeri en Ullman als Maier, Mendelzon en Sagiv [2, 30]. In 1984 werd het algoritme in meer detail bekeken door Beeri en Vardi[3, 4].

De intuïtieve werking van een chase over een verzameling dependencies Σ verloopt als volgt: We vertrekken vanuit een tableau T. Op dit tableau T passen we telkens een dependency uit Σ toe totdat dit geen veranderingen meer geeft. Voordat we op dit tableau beginnen werken passen we eerst een hernoeming toe op de variabelen in dit tableau. We vervangen ze in volgorde door $x_1, \ldots, x_n$. Dit is belangrijk zodat we een lexicografische volgorde op de variabelen hebben. Het is ook belangrijk dat we weten dat de volgende nieuwe variabele die we introduceren x_{n+1} zal zijn.

3.2 Definitie

Voor de Chase van een tableau T over een verzameling dependencies Σ (die beide voldoen aan een bepaald databaseschema S) vertrekken we vanuit dit tableau T.

Op dit tableau T gaan we nu *chasen* door dependencies uit Σ toe te passen. De chase is een inductief algoritme. Dit wil zeggen dat we vertrekken vanuit T en stap per stap verder gaan. Elke toepassing van een dependency uit Σ zal tot een nieuw tableau T_i leiden. We krijgen dus een sequentie van tableaus $T, T_1, \ldots, T_i, \ldots$.

We geven nu aan hoe deze toepassing van dependencies verloopt voor zowel TGD's als EGD's.

- EGD: We kiezen een EGD σ_1 uit Σ. Voor het tableau T_e gegeven door de body van deze σ_1 zoeken we een homomorfisme h van T_e naar T. Als er geen homomorfisme bestaat van T_e naar T dan kunnen we de dependency σ_1 niet toepassen.
  Als dit wel bestaat dan kunnen we de head van σ_1 toepassen op T. Dit doen we

door de gelijkheid $a_1 = a_2$ uit σ_1 toe te passen op T. Hierbij is het van belang of a_1 en a_2 constanten of variabelen zijn.

  - Als a_1 en a_2 beide variabelen zijn dan weten we door het hernoemen van de originele variabelen naar $x_1 \ldots, x_n$ dat er een orde is op $h(a_1)$ en $h(a_2)$. We krijgen dus $h(a_1) \leq h(a_2)$ of $h(a_2) \leq h(a_1)$. We passen σ dan toe door de voorkomens van het grotere element in T telkens te vervangen door het kleinere element.
  - Als een van de twee een constante is dan beschouwen we deze constante als het kleinste element. We gaan dan alle voorkomens van de variabele in T vervangen door de constante.
  - Als a_1 als a_2 beide constanten zijn kan dit tot een probleem leiden. a_1 en a_2 mogen immers geen verschillende constanten zijn. Dan krijgen we een situatie die analoog is aan $2 = 4$. Dit kan uiteraard niet voorvallen. In dit geval faalt de volledige chase dan ook. We stoppen dan met de uitvoering van het algoritme en zeggen dat dit geen resultaat geeft.

- TGD: We kiezen een TGD σ_2 uit Σ. Voor het tableau T_t gegeven door de body van σ_2 zoeken een homomorfisme h van T_t naar T. Hierbij is het belangrijk dat er geen uitbreiding h' is zodat $h \subseteq h'$ en de toepassing van h' op de head van σ_2 vervat zit in T (dus $T \not\models \sigma_2$). De toepassing van σ_2 op T is dan het toevoegen van de head van σ_2 aan T. Dit doen we door voor elk tupel t uit deze head $h(t)$ toe te voegen aan T.
  Merk op dat in het geval van embedded TGD's niet elke variabele uit de head ook voor zal komen in de body. Dit wil zeggen dat er variabelen zijn uit σ_2 die niet voorkomen in het domein van h, we kunnen dus $h(t)$ niet berekenen voor deze variabelen. Dit zijn de variabelen met een existentiële kwantor in de head van σ_2 die niet voorkomen in de body van σ_2 en dus ook niet in het homomorfisme van deze body. Voor het toepassen van een embedded TGD breiden we h dus uit zodat de existentieel gekwantificeerde variabelen uit de head van σ_2 telkens afgebeeld worden op een nieuwe x_{n+1} als x_n op dat moment de variabele is in T met de grootste index. We gebruiken dan de uitgebreide vorm van h om voor elke tupel t uit de head van σ_2 $h(t)$ toe te voegen aan T.

Merk op dat de toevoeging van nieuwe variabelen in de TGD regel ervoor kan zorgen dat de chase oneindig blijft doorgaan, dit is echter niet het geval als we gebruik maken van full dependencies. Dan worden er immers geen nieuwe variabelen geïntroduceerd. We blijven deze EGD en TGD regels toepassen tot er geen verandering meer optreedt.

De *chase sequence* is een mogelijk oneindige opeenvolging van tableaus $= T_1 \ldots T_n \ldots$ Waarbij T_{i+1} telkens verkregen kan worden door het toepassen van een dependency uit Σ op T_i.) De *chase step* is een stap uit de chase sequence.

Een eindige chase sequence is *terminaal* als op het laatste tableau in de chase sequence geen enkele verandering meer verkregen kan worden via een chase step.

Het resultaat van een chase is het laatste tableau uit de terminale chase sequence. Als we vertrekken vanuit een tableau T en een verzameling dependencie Σ dan noteren we dit eindresultaat als $chase_\Sigma(T)$. Merk op dat de chase niet altijd dient te eindigen dus dat er niet altijd een dergelijke $chase_\Sigma(T)$ bestaat.

We kunnen de chase ook gebruiken voor een bepaalde dependency σ en een verzameling dependencies Σ. We spreken dan van een chase van σ over Σ. Deze vorm van de chase

wordt gebruikt om het implicatieprobleem op te lossen. De definitie is volledig analoog aan de eerdere definitie. Het enige verschil is dat het startableau T nu gegeven zal worden door de body van dependency σ. Het toepassen van de dependencies uit Σ blijft ongewijzigd. We weten dan dat $\Sigma \models \sigma$ als $chase_\Sigma(T) \models \sigma$. We geven een voorbeeld van een dergelijke chase in Voorbeeld 3.2.1. Meer informatie over deze toepassing van de chase vinden we in Paragraaf 8.1

De volgorde van toepassing van de dependencies in de chase is niet van belang, een verschillende volgorde in de toepassing zal nog steeds eenzelfde resultaat geven. Dit wordt de Church-Rosser eigenschap genoemd [1]. Het bewijs van deze eigenschap vermelden we niet in detail. Dit steunt op het veronderstellen van twee verschillende resultaten van een chase afhankelijk van de gevolgde volgorde van toepassing, dus met een verschillende chase sequence. We tonen dat er dan een homomorfisme bestaat tussen deze resultaten in beide richtingen.

Wel zullen we in de volgende hoofdstukken het al dan niet eindigen van de chase ofwel de *terminatie* van de chase in meer detail bekijken.

Voorbeeld 3.2.1. *We geven een voorbeeld van chase over een enkele relatie R met ariteit drie. We maken gebruik van full dependencies zodat de chase zeker eindigt. De verzameling dependencies $\Sigma = \{\sigma_1, \sigma_2\}$ met :*

- $\sigma_1 = R(y_1, y_2, y_3) \wedge R(y_1, y_4, y_5) \rightarrow R(y_1, y_2, y_5)$
- $\sigma_2 = R(y_1, y_2, y_3) \wedge R(y_4, y_2, y_5) \rightarrow y_3 = y_5$

We willen nu bepalen of $\Sigma \models \sigma$ waarbij:

$$\sigma = R(y_1, y_2, y_3) \wedge R(y_1, y_4, y_5) \rightarrow y3 = y5$$

We vertrekken vanuit het tableau gegeven door de body van σ. Hierop hernoemen we de variabelen naar $x_1, x_2, \ldots$. Dit geeft het eerste tableau van onze chase T_1:

x_1	x_2	x_3
x_1	x_4	x_5

Hierna beginnen we met het toepassen van dependencies uit Σ. We beginnen met het toepassen van σ_1. We zoeken een homomorfisme van de tableau T_{σ_1} gegeven door de body van σ_1 naar T_1. Een eerste homomorfisme $h = \{(y_1, x_1), (y_2, x_2), (y_3, x_3), (y_4, x_4)$ $,(y_5, x_5)\}$. De tupel $t = R(y_1, y_2, y_5)$ is de head van σ_1. We zien dat $h(t) = R(x_1, x_2, x_5)$ niet voorkomt in T_1. Er is ook geen uitbreiding van h mogelijk aangezien alle variabelen uit de head al een mapping hebben in h. We kunnen dus $h(t)$ toevoegen aan T_1 om T_2 te bekomen. T_2 ziet er dus als volgt uit:

x_1	x_2	x_3
x_1	x_4	x_5
x_1	x_2	x_5

Er zijn echter nog andere homomorfismen van T_{σ_1} naar T_1 (en dus ook naar T_2). We kiezen een homomorfisme $h = \{y_1, x_1), (y_2, x_4), (y_3, x_5), (y_4, x_2), (y_5, x_3)\}$. Als we dit

toepassen op de head t van σ_1 krijgen we $h(t) = R(x_1, x_4, x_5)$. We zien echter dat $h(t)$ reeds voorkomt in T_2. We kunnen dus σ_1 niet toepassen voor dit homomorfisme.

We besluiten dan om te kijken naar σ_2. We zoeken dus een homomorfisme van T_{σ_2} het tableau dat overeenkomt met de body van σ_2 naar T_2. Een eerste homomorfisme dat we vinden is $h = \{(y_1, x_1), (y_2, x_2), (y_3, x_3), (y_4, x_1), (y_5, x_3)\}$. Voor de head van σ_2 namelijk $y_3 = y_5$ geeft h ons $x_3 = x_3$. De toepassing van σ_2 volgens dit homomorfisme heeft dus geen effect.

We proberen dus een ander homomorfisme. We kiezen een nieuw homomorfisme $h = \{(y_1, x_1), (y_2, x_2), (y_3, x_3), (y_4, x_2), (y_5, x_5)\}$. Hier geeft het toepassen van h op de head van σ_2 het resultaat $x_3 = x_5$. Dit kunnen we toepassen op T_2. Het resultaat van deze toepassing geeft ons T_3 en ziet er als volgt uit:

x_1	x_2	x_3
x_1	x_4	x_3

Het toepassen van de gelijkheid gaf ons immers twee gelijke relatieatomen $R(x_1, x_2, x_3)$ in T_3, we kunnen dus één van deze dubbels weglaten.

We proberen nu opnieuw om σ_1 toe te passen. Hiervoor moeten we een homomorfisme vinden van T_{σ_1} naar T_3. Mogelijke homomorfismen zullen altijd y_1 afbeelden op x_1 aangezien dit de enige variabele is die voorkomt in de eerste positie. Verder zal y_2 afgebeeld worden op x_2 of x_4 aangezien dit de enige varaibelen zijn die voorkomen in de tweede positie van R in T_3. Voor y_5 zien we dat deze altijd zal afgebeeld worden op x_3 aangezien dit de enige variabele is die voorkomt in de laaste positie. Onder alle mogelijke homomorfismen zijn er slechts twee mogelijkheden voor de toepassing van het homormorfisme op de head $t = R(y_1, y_2, y_5)$ van σ_1. Deze resultaten zijn $R(x_1, x_2, x_3)$ en $R(x_1, x_4, x_3)$. Beide van deze mogelijkheden komen echter al voor in T_3. We kunnen dus geen verandering teweeg brengen door de toepassing van σ_1 op T_3.

We bekijken dus σ_2. We zien dat deze EGD telkens de variabele in de laatste positie gelijk zal stellen. In T_3 komt echter alleen x_3 voor op de laatste positie van R. De head van σ_2 zal dus altijd $x_3 = x_3$ geven voor eender welke homomorfisme van T_{σ_2} naar T_3. Er kan dus ook geen verandering verkregen worden door de toepassing van σ_2.

We hebben nu dus de dependencies uit Σ toegepast totdat er geen verandering meer is. Het resultaat van de chase is dus $chase_\Sigma(T_1) = T_3$. We controleren nu of σ waar is in T_3 oftewel $T_3 \models \sigma$. We zien dat dit inderdaad het geval is. Elke homomorfisme van T_σ zal immers zowel y_3 als y_5 op x_3 moeten afbeelden aangezien dit de enige variabele is die voorkomt in de laatste positie van R in $chase_\Sigma(T_1)$. We zien dan dat de gelijkheid $y_3 = y_5$ altijd zal afgebeeld worden op $x_3 = x_3$ en dus zien we dat $chase_\Sigma(T_1) \models \sigma$. Aangezien het resultaat van de chase voldoet aan σ kunnen we concluderen dat $\Sigma \models \sigma$. We hebben dus gebruik gemaakt van de chase om het implicatieprobleem voor Σ en σ op te lossen.

Hoofdstuk 4

Soorten terminatie van de chase

Dit hoofdstuk geeft een inleiding tot de terminatie van de chase. Het is gebaseerd op het werk gedaan door Meier in zijn doctoraatsthesis [32].

We beginnen met een voorbeeld dat aantoont dat de terminatie van de chase niet vanzelfsprekend is.

Voorbeeld 4.0.2. *Veronderstel de volgende gegevens:*

- $T = \{verbinding(Brussel, Parijs)\}$
- $\Sigma = \{\sigma\}$
- $\sigma = verbinding(x_1, x_2) \rightarrow \exists y\ verbinding(x_2, y)$

Het toepassen van σ zal dus altijd leiden tot de introductie van een nieuwe gelabelde null. Als we nu de chase achtereenvolgens toepassen vertrekkende vanuit $T_0 = T$ krijgen we de volgende resultaten:
$T_1 = \{verbinding(Brussel, Parijs), verbinding(Parijs, n_1\}$
$T_2 = \{verbinding(Brussel, Parijs), verbinding(Parijs, n_1), verbinding(n_1, n_2)\}$
$T_3 = T_2 \cup \{verbinding(n_2, n_3)\}$
En zo verder tot in het oneindige. Deze chase zal dus zeker nooit eindigen want we kunnen σ blijven toepassen.

De voortdurende introductie van nieuwe variabelen leidt dus duidelijk tot een probleem. We vragen ons dan ook af of de terminatie van de chase een beslisbaar probleem is.

Er zijn echter bepaalde voorwaarden die we kunnen gebruiken om het eindigen van de chase af te dwingen. We geven nu een voorbeeld van een conditie die de terminatie kan afdwingen.

4.1 Zwakke acycliciteit

Een methode om het opbouwen van nieuwe variabelen te stoppen werd geintroduceerd door Fagin in 2005 [12]. Deze methode is zwakke acycliciteit. De methode voorkomt de generatie van nieuwe variabelen op een syntactische manier.

We introduceren nu de noodzakelijke begrippen voor de definitie van zwakke acycliciteit. De *dependency graph* van een verzameling dependencies Σ, genoteerd als $dep(\Sigma)$ definiëren we als volgt: De verzameling knopen is de verzameling van de posities die voorkomen in een TGD in Σ. Er zijn twee soorten bogen. Voor elke TGD: $\forall \overline{x}(\varphi(\overline{x}) \rightarrow \exists \overline{y}\ \psi(\overline{x}, \overline{y})) \in \Sigma$ en voor elke x in $\overline{x}$ die voorkomt in ψ en voor elk voorkomen van x in φ op positie π_1:

- Voor elk voorkomen van x in ψ op positie π_2, voeg een boog $\pi_1 \rightarrow \pi_2$ toe als deze nog niet bestaat.
- Voor elke existentieel gekwantificeerde variabele y en voor elk voorkomen van y op positie π_2, voeg een speciale boog $\pi_1 \overset{*}{\rightarrow} \pi_2$ toe als deze nog niet bestaat.

Een verzameling Σ van dependencies wordt *zwak acyclisch* genoemd als de dependency graph $dep(\Sigma)$ geen cyclus bevat via een speciale boog.

Intuïtief kunnen we deze definitie als volgt bekijken: De gewone bogen geven de stroom van de data binnen de database posities aan. De speciale bogen geven de gevallen aan er waar nieuwe gelabelde nullen gegenereerd worden. Deze speciale bogen zijn dus de oorzaak van de opbouw van gelabelde nullen.

Fagin toonde aan dat de chase eindigt voor zwak acyclische verzamelingen van dependencies[12]. Intuitief is dit resultaat duidelijk omdat we op een syntactische manier ervoor zorgen dat het onmogelijk is om nieuwe gelabelde nullen te genereren. De bogen die gelabeld zijn met een ster zijn immers de oorzaak van nieuwe variabelen. Door ervoor te zorgen dat deze niet voorkomen in een cyclus zorgen we er voor dat er geen ophoping van nieuwe variabelen voorkomt. Fagin toont verder aan dat dit kan in polynomiale tijd ten opzichte van de grootte van het domein van de input instance.

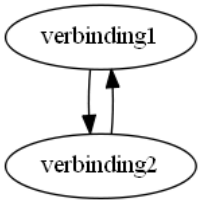
We hebben dus een manier gevonden om de terminatie van de chase af te dwingen.

We illustreren het gebruik van zwakke acycliciteit aan de hand van Voorbeeld 4.1.1. Dit voorbeeld geeft ook aan waarom het zwakke acycliciteit genoemd wordt. De graaf mag immers wel over cycli beschikken zolang deze geen speciale boog bevatten.

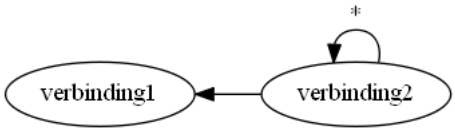
Voorbeeld 4.1.1. *Veronderstel dat we vertrekken vanuit:*

- $T = \{verbinding(Brussel, Parijs)\}$
- $\Sigma = \{\sigma\}$
- $\sigma = verbinding(x_1, x_2) \rightarrow verbinding(x_2, x_1)$

We kunnen de dependency graph opstellen voor Σ. De knopen van deze graaf komen overeen met de posities uit onze TGD σ en zijn dus $\{verbinding.1, verbinding.2\}$. Er zijn geen speciale bogen aangezien de TGD niet over een existentiële kwantor beschikt. De gewone bogen zijn tussen verbinding.1 en verbinding.2 en weer terug. Deze graaf is te zien in Figuur 4.1.



Figuur 4.1: *De dependency graph Voorbeeld 4.1.1*



Figuur 4.2: *De dependency graph uit Voorbeeld 4.1.2*

We zien dat de graaf wel over een cyclus beschikt maar niet over speciale boog($\xrightarrow{*}$). We zien dan ook bij de toepassing van de chase dat deze na een enkele iteratie stopt. Door het toepassen van de TGD σ krijgen we immers:

$$T' = \{verbinding(Brussel, Parijs), verbinding(Parijs, Brussel)\}$$

Hierna is σ niet meer toepasbaar en stopt de chase.

We kunnen nu Voorbeeld 4.0.2 uit het begin van de paragraaf verder uitwerken aan de hand van de dependency graph.

Voorbeeld 4.1.2. *We werken nog steeds met dezelfde informatie:*

- $T = verbinding(Brussel, Parijs)$
- $\Sigma = \{\sigma\}$
- $\sigma = verbinding(x_1, x_2) \rightarrow \exists y\ verbinding(x_2, y)$

We weten dat de chase van Σ over T niet zal eindigen.

*Wel kunnen we de dependency graph opstellen voor Σ. We zien dat de knopen zullen bestaan uit verbinding.*1 *en verbinding.*2*. Verder zal er een pijl zijn tussen verbinding.*2 *en verbinding.*1 *en een speciale pijl die verbinding.*2 *verbindt met zichzelf. Deze graaf is te zien in Figuur 4.2.*

Deze speciale pijl vormt een cyclus op zichzelf. We zien dus dat Σ niet zwak acyclisch is en dus zal de chase van Σ over I zeker niet eindigen. Het komt dus overeen met de resultaten van ons eerdere voorbeeld.

We weten nu dat als Σ zwak acyclisch is dat de chase van Σ zal eindigen. In de omgekeerde richting is dit echter niet het geval. Er zijn verzamelingen van dependencies die niet zwak acyclisch zijn maar waarvoor de chase toch eindigt. Een voorbeeld hiervan komt later aan bod in Voorbeeld 6.1.7. We kunnen ons dus afvragen of er een terminatie conditie is die wel alle gevallen omvat.

4.2 Algemene terminatie

In het algemene geval is de terminatie van de chase onbeslisbaar. Het is onbeslisbaar of alle chase sequences van de chase van T over Σ zullen eindigen. Het is zelfs onbeslisbaar of er een chase sequence is zodat de chase van T over Σ volgens deze chase sequence eindigt. Aangezien dit een zeer belangrijke eigenschap is bekijken we het bewijs hiervoor in meer detail.

We bespreken het bewijs voor TGD's van Deutsch uit 2008[9]. De opzet van het bewijs is om de onbeslisbaarheid van de terminatie te bewijzen door het haltingprobleem voor Turing Machines(TM's) te reduceren naar het probleem van de terminatie van een chase van een tableau T over een verzameling TGD's Σ volgens een zekere chase sequence.

Om dit te doen encoderen we een berekening als een graaf met horizontale en verticale bogen. De graaf krijgt dus de vorm van een raster. Elke rij in dit raster encodeert één configuatie van de TM. Dit geeft dus aan wat er op de tape staat, wat er op de head van de TM staat en wat de positie van de head is. Opeenvolgende rijen geven dan opeenvolgende configuraties van de TM aan. Deze rijen zijn verbonden door twee soorten verticale bogen, een soort links en een soort rechts van de head. We maken gebruik van de volgende relaties:

- $T(x, a, y)$ De tape, een horizontale boog van x naar y met symbool a.
- $H(x, s, y)$ De head,een horizontale boog van x naar y met toestand s.
- $L(x, y)$ Een links verticale boog.
- $R(x, y)$ Een rechts verticale boog.

Hierbij maken we gebruik van constanten voor alle symbolen die voorkomen op de tape van de TM. Verder is er ook een constante voor elke toestand van de head en twee constanten B en E die het begin en het einde van de tape aangeven.

We maken gebruik van deze relaties en constanten om een verzameling Σ van TGD's op te bouwen. Σ ziet er als volgt uit:

1. We stellen eerst de initiële configuratie in. Dit doen we door middel van de volgende dependency:

   $$\exists w, x, y, z T(w, B, x), T(x, \#, y), H(x, s_0, y), T(y, E, z)$$

   Hierbij is # een leeg symbool op de tape en s_0 is de begintoestand. Samen met B en E zijn dit de constanten die voorkomen in de TGD. De tape bevat dus enkel het lege symbool # en de head bevindt zich in toestand s_0.

2. Voor elke transitie tussen twee toestand waarbij de head naar rechts beweegt, symbool a vervangen wordt door a' en de toestand overgaat van s naar s' voegen we de volgende dependency toe:

   $$T(x, a, y) \wedge H(x, s, y) \wedge T(y, b, z) \rightarrow$$
   $$\exists x', y', z' L(x, x') \wedge R(y, y') \wedge R(z, z') \wedge T(x', a', y') \wedge T(y', b, z') \wedge H(y', s', z')$$

   Hierbij zijn a, a', s, s' constanten aangezien ze overeenkomen met symbolen van de tape en toestanden van de TM.

3. Voor elke overgang van de head naar rechts voorbij het einde van de tape. De overgang gaat van toestand s naar s' en vervangt symbool a door a'. Hiervoor gebruiken we de volgende TGD:

$$\begin{aligned} &T(x,a,y) \wedge H(x,s,y) \wedge T(y,E,z) \rightarrow \\ \exists w', x', y', z' L(x,x') \wedge R(y,y') \wedge R(z,z') \wedge \\ &T(x',a',y') \wedge T(y',\#,z') \wedge H(y',s',z') \wedge T(z',E,w') \end{aligned}$$

   Opnieuw zijn a, a', s, s' constanten.

4. Voor de transities naar de linkerkant maken we analoge TGD's.

5. Voor de transities waarbij de head niet opschuift maken we ook een analoge TGD.

6. We dwingen nu af dat de linkse kopie altijd bestaat. Hiervoor stellen we de volgende TGD op:

$$T(x,a,y) \wedge L(y,y') \rightarrow \exists x' L(x,x') \wedge T(x',a,y')$$

7. We maken een analoge TGD voor de rechtse kopie:

$$T(x,a,y) \wedge R(x,x') \rightarrow \exists y' T(x',a,y') \wedge R(y,y')$$

We kunnen nu de chase van een leeg tableau T over Σ bekijken. De eerste TGD uit Σ zal altijd toegepast kunnen worden. We vertrekken dus vanuit een tape met enkel het lege symbool # op. De andere dependencies zullen dan toegepast worden wanneer de overeenkomstige transitie voorkomt in de TM. Hierbij wordt telkens eerst een van de transities toegepast waarna de nodige TGD's voor de linkse en of rechtse kopieën toegepast kunnen worden. We zien dus dat we de uitvoering van de TM simuleren via de TGD's. Als er geen enkele transitie regel meer toegepast kan worden en alle mogelijke kopie regels zijn toegepast dan stopt de uitvoer van de TM. We zien dus dat de uitvoering van de TM stopt als en slechts als de chase een terminale chase sequence heeft. We hebben dus een reductie gevonden van het halting probleem naar het probleem van de terminatie van de chase. Hiermee hebben we dus aangetoond dat de terminatie van de chase onbeslisbaar is.

Merk op dat dit zowel voor een enkele chase sequence als voor alle mogelijke chase sequences geldt. Als we één terminale chase sequence vinden weten we immers dat de TM zal eindigen, dit wil zeggen dat de chase voor alle mogelijke chase sequences zal eindigen. Aangezien de TM zal stoppen komen we immers op een punt waarop het niet meer mogelijk is om nog naar een nieuwe toestand over te gaan. Dit zal er dus voor zorgen dat er ook geen transitieregel toegepast zal kunnen worden en dus zal de chase ook eindigen.

4.3 Opdeling van de terminatie

We weten nu dat de terminatie van de chase onbeslisbaar in het algemeen. Het is dus interessanter om te kijken of er andere gevallen zijn waarin het eindigen van de chase wel beslisbaar is. Met dit als doel introduceren we de belangrijkste vernieuwing uit de doctoraatsthesis van Meier, namelijk de opdeling van de terminatie van de chase in verschillende gevallen [32]. We introduceren eerst de verschillende gevallen en zullen daarna in aparte hoofdstukken de terminatie per geval bekijken.

We kunnen terminatie opdelen in vier verschillende gevallen:

- voor alle tableaus en voor alle chase sequences,
- voor alle tableaus en minstens één chase sequence,
- voor een gegeven tableau en voor alle chase sequences,
- voor een gegeven tableau en minstens één chase sequence.

Formeel introduceren we de volgende notatie en definitie voor de verschillende gevallen van chase terminatie (CT):

- $CT_{\forall\forall} = \{\Sigma \mid$ voor elk tableau U zijn alle mogelijke chase sequences terminaal voor de chase van U over $\Sigma\}$
- $CT_{\forall\exists} = \{\Sigma \mid$ voor elk tableau U bestaat er een terminale chase sequence voor de chase van U over $\Sigma\}$
- $CT_{T,\forall} = \{\Sigma \mid$ elke chase sequence voor de chase van T over Σ is terminaal$\}$
- $CT_{T,\exists} = \{\Sigma \mid$ er bestaat een chase sequence die terminaal is voor de chase van T over $\Sigma\}$

We zeggen dus dat de chase eindigt als er een terminale chase sequence is.

We kunnen nu bekijken hoe deze vier gevallen met elkaar in verband staan. Er zijn duidelijk relaties tussen de verschillende gevallen. We weten dat $CT_{\forall\forall}$ altijd de kleinste verzameling zal zijn. Er zal immers de meeste restrictie moeten toegepast worden om de chase te doen eindigen in het algemene geval. Verder weten we ook dat $CT_{T,\forall}$ en $CT\forall, \exists$ meer restrictief zullen zijn dan $CT_{T,\exists}$.

We zien ook dat we door het nemen van de doorsnede voor alle eindige tableaus T kunnen gaan van $CT_{T,\forall}$ naar $CT_{\forall,\forall}$. Dit omdat we door het nemen van de doorsnede alle mogelijke tableaus T beschouwen. Als het geldt voor de doorsnede geldt het immers voor alle T uit deze doorsnede en dus gaan we van T naar $\forall$. Volledig analoog kunnen we dit ook gebruiken om van $CT_{T,\exists}$ te gaan naar $CT_{\forall,\exists}$.

Dit geeft ons de volgende relaties:

1. $CT_{\forall\forall} \subseteq CT_{T,\forall} \subseteq CT_{T,\exists}$
2. $CT_{\forall\forall} \subseteq CT_{\forall,\exists} \subseteq CT_{T,\exists}$
3. $\bigcap_{T \text{ eindige chase instance}} CT_{T,\forall} = CT_{\forall,\forall}$
4. $\bigcap_{T \text{ eindige chase instance}} CT_{T,\exists} = CT\forall, \exists$

We zullen nu aan de hand van enkele voorbeelden de relaties tussen deze gevallen verder bestuderen.

Voorbeeld 4.3.1. *Neem twee tableaus $T = \{E(a,b), E(b,a)\}$ en $U = \{E(a,b)\}$ met $\Sigma = \{E(x_1,x_2) \rightarrow \exists y\ E(x_2,y)\}$. In dit geval zal de chase van T over Σ eindigen aangezien de dependency uit Σ nooit toegepast zal worden. De chase van U over Σ zal echter nooit eindigen aangezien de dependency uit Σ telkens opnieuw toegepast kan worden.*

Aan de hand van dit voorbeeld zien we dat er een T bestaat zodat $\boxed{CT_{T,\forall} \setminus CT_{\forall\forall} \neq \emptyset}$. We zien immers dat de chase van T over Σ eindigt voor alle chase sequences (er kan immers geen dependency uit Σ worden toegepast op T), dus $\Sigma \in CT_{T,\forall}$. We zien echter ook dat de chase van U over Σ niet eindigt en dus $\Sigma \notin CT_{\forall\forall}$.

Als we weten dat $CT_{T,\forall} \setminus CT_{\forall\forall} \neq \emptyset$ volgt hier uiteraard uit dat $\boxed{CT_{T,\exists} \setminus CT_{\forall\forall} \neq \emptyset}$ aangezien $CT_{T,\forall} \subseteq CT_{T,\exists}$.

Nog voor dit voorbeeld kunnen we aantonen dat er geen chase sequence bestaat voor U over Σ zodat deze eindigt. We zullen immers altijd de dependency uit Σ moeten blijven toepassen wat steeds tot de introductie van nieuwe variabelen zal leiden. Dit wil zeggen dat $\Sigma \notin CT_{\forall,\exists}$. U geeft hier immers een tegenvoorbeeld voor. Als Σ wel deel zou zijn van $CT_{\forall,\exists}$ dan zouden we voor elke tableau, dus ook voor U een chase sequence moeten vinden die terminaal is. We herinneren ook dat $\Sigma \in CT_{T,\forall}$. Dit wil zeggen dat $\boxed{CT_{T,\forall} \setminus CT_{\forall\exists} \neq \emptyset}$.

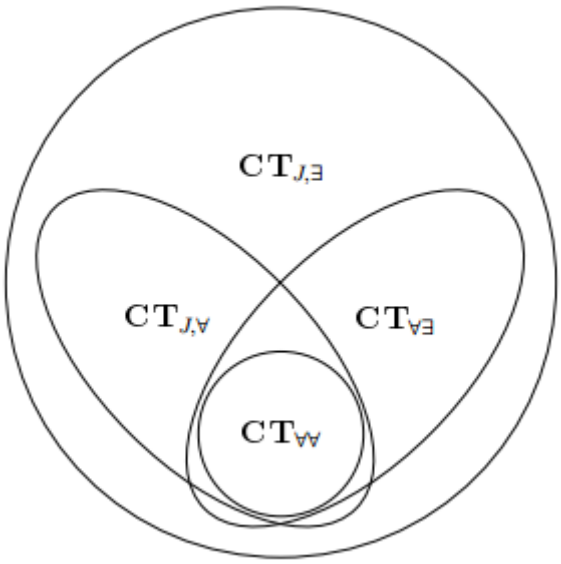
Voorbeeld 4.3.2. *We nemen een tableau $T = \{E(a,b)\}$ en $\Sigma = \{\sigma_1, \sigma_2\}$ met:*

- $\sigma_1 = E(x_1,x_2) \rightarrow \exists y\ E(x_2,y)$
- $\sigma_2 = E(x_1,x_2) \rightarrow E(x_2,x_1)$

We zien dat het hier mogelijk is om een oneindige chase sequence te bouwen. Dit doen we door eerst σ_1 toe te passen. Dit zal leiden tot de introductie van een nieuwe variabele. Voor het relatieatoom dat deze nieuwe variabele bevat zullen we σ_1 opnieuw kunnen toepassen. We krijgen dus een ophoping van nieuwe variabelen wat leidt tot een oneindige chase sequence en dus een oneindige chase. Als we echter eerst σ_2 toepassen in onze chase sequence dan zal deze chase sequence wel terminaal zijn. Door het toepassen krijgen we immers tableau $T' = \{E(a,b), E(b,a)\}$. Hierop kunnen we σ_1 niet toepassen aangezien er een duidelijk homomorfisme is van de head van deze dependency naar T'. Hetzelfde geldt voor σ_2 en dus is er geen verandering meer mogelijk. We zien dus dat het terminaal zijn van de chase sequence afhangt van de dependency die als eerste wordt toegepast.

We zien dus dat we voor éénzelfde tableau T en verzameling depedencies Σ een verschillend resultaat krijgen voor de terminatie voor verschillende chase sequences. We hebben dus een T en een Σ gevonden die aantonen dat $\boxed{CT_{T,\exists} \setminus CT_{T,\forall} \neq \emptyset}$. Er is immers terminatie voor een bepaalde chase sequence maar niet in alle gevallen.

Als we verdergaan op dit voorbeeld zien we ook dat $\Sigma \notin CT_{\forall\forall}$. We hebben immers een chase sequence en een instance T gegeven waarvoor de chase niet eindigt. We kunnen echter wel aantonen dat $\Sigma \in CT_{\forall\exists}$. Dit doen we door als chase sequence telkens eerst gebruik te maken van σ_2. Dit zal gelden voor elke input instance. We hebben immers slechts twee mogelijk gevallen voor een input tableau T'. Ofwel zijn zowel σ_1 en σ_2 niet toepasbaar op T' en eindigt de chase dus meteen, ofwel zijn σ_1 en σ_2 wel toepasbaar.



Figuur 4.3: *Een visualisatie van de inclusie relatie tussen de verschillende deelverzamelingen van chase terminatie*

In dit geval volgen we de chase sequence uit het voorbeeld en passen we σ_2 toe. Merk op dat er geen geval is waarin we σ_1 kunnen toepassen maar niet σ_2, ze hebben immers dezelfde body en een analoge head. Na toepassing van σ_2 zal het resultaat ook voldoen aan σ_1. De tupel toegevoegd door σ_2 maakt immers de conditie opgelegd door σ_1 waar. Als we E beschouwen als een boog in een graaf proberen beide dependencies immers om vanuit het eindpunt van een gegeven boog een nieuwe boog te doen vertrekken. Het verschil is dat σ_1 deze nieuwe boog kan maken naar een nieuwe knoop terwijl σ_2 dit doet door terug te linken naar een reeds bestaande knoop. We kunnen dus in onze sequence telkens eerst σ_2 toepassen zodat er geen nieuwe variabelen geïntroduceerd worden.
We zien dus dat $\Sigma \notin CT_{\forall\forall}$ en $\Sigma \in CT_{\forall\exists}$.
Dit leidt dus tot het resultaat dat $\boxed{CT_{\forall\exists} \setminus CT_{\forall\forall} \neq \emptyset}$.

Voor dit voorbeeld kunnen we ook aantonen dat $\Sigma \notin CT_{T,\forall}$. We kiezen er dan in onze chase sequence voor om eerst σ_1 toe te passen. Dit introduceert telkens nieuwe variabelen en zal dus nooit eindigen. We weten dan ook dat $\boxed{CT_{\forall\exists} \setminus CT_{J,\forall} \neq \emptyset}$.

Een overzicht van de inclusie relatie tussen deze verzamelingen is te vinden in Figuur 4.3 gemaakt door Meier[32].

Het is dus duidelijke dat de nieuwe gevallen verzamelingen vertegenwoordigen die minder strikt zijn dan de originele $CT_{\forall\forall}$. In de komende paragrafen zullen we elk van deze gevallen in meer detail bekijken.

Hoofdstuk 5

Studie van $CT_{\forall\exists}$

We beginnen met het bespreken van $CT_{\forall\exists}$. Dit is een eenvoudiger geval dan $CT_{\forall\forall}$ aangezien we slechts één chase sequence moeten vinden die terminaal is.

5.1 Stratificatie

Stratificatie is geïntroduceerd door Deutsch als verbetering van zwakke acycliciteit [9]. Het idee achter stratificatie is het opdelen van de verzameling in deelverzamelingen en deze dan testen voor zwakke acycliciteit. De terminatie van de volledige verzameling zou dan het resultaat moeten zijn van de test voor zwakke acycliciteit van de deelverzamelingen. Het blijk echter dat stratificatie een terminatieconditie is voor $CT_{\forall\exists}$ en niet voor $CT_{\forall\forall}$.

Om de decompositie in deelverzamelingen mogelijk te maken introduceren we de binaire relatie $\prec$ op een verzameling dependencies. Gegeven twee dependencies σ_1 en σ_2 uit een verzameling dependencies Σ zeggen we dat $\sigma_1 \prec \sigma_2$ als er een tableau T en twee tupels $\overline{a}, \overline{b}$ zodat:

- $T \not\models \sigma_1(\overline{a})$.
- $T \models \sigma_2(\overline{b})$.
- Er is een chase stap in de gewone chase die σ_1 toepast op T voor tupel $\overline{a}$ zodat T' het resultaat is.
- $T' \not\models \sigma_2(\overline{b})$.

We zien dat de eerste voorwaarde verzekert dat er inderdaad een chase stap is op T volgens σ_1.De intuïtieve betekenis van $\sigma_1 \prec \sigma_2$ is dat de toepassing van σ_1 kan leiden tot de toepassing van σ_2. σ_1 kan dus σ_2 afvuren.

Met behulp van deze relatie komen we dan tot de definitie van de chase graph $G(\Sigma)$ voor een verzameling dependencies Σ. Dit is een gerichte graaf $G(\Sigma) = (\Sigma, E)$. De knopen zijn dus de dependencies uit Σ. Voor elke twee dependencies $\sigma_1, \sigma_2 \in \Sigma$ is er een gerichte boog (σ_1, σ_2) in E als en slechts als $\sigma_1 \prec \sigma_2$.

We zeggen dan dat Σ voldoet aan stratificatie als en slechts als de dependencies in elke cyclus van $G(\Sigma)$ zwak acyclisch zijn.

Uit de definitie volgt dat stratificatie een generalisatie is van zwakke acycliciteit want:

- Als Σ zwak acyclisch is dan voldoet Σ aan stratificatie.
- Er zijn verzamelingen van dependencies die voldoen aan stratificatie maar niet zwak acyclisch zijn. (zie Voorbeeld 5.1.2)

We zien dan dat als Σ voldoet aan stratificatie dan $\Sigma \in CT_{\forall\exists}$[32].

De reden dat dit zo is, is dat we de chase graph kunnen gebruiken om een terminale chase sequence op te stellen. We doen dit door eerst te chasen met de dependencies uit de cycli en daarna eventueel met de rest van de dependencies. Als we deze strategie volgen zal dit altijd leiden tot een eindige chase sequence, onafhankelijk van het tableau waarop de chase gebeurt.

We geven een schets van het bewijs van deze stelling. We bewijzen dat als voor elke cyclus Σ' van de chase graph $G(\Sigma)$ geldt dat $\Sigma' \in CT_{\forall\exists}$ dan $\Sigma in CT_{\forall\exists}$.

We doen dit door een equivalentierelatie $\sim$ op te stellen. We zeggen voor twee dependencies Σ_1 en Σ_2 dat $\Sigma_1 \sim \Sigma_2$ als σ_1 en σ_2 in een gemeenschappelijke cyclus zitten in $G(\Sigma)$ of als $\sigma_1 = \sigma_2$.

Met $\Sigma/\sim$ bedoelen we de dependencies uit Σ die voorkomen in de $\sim$ relatie. We noemen deze dependencies $\{W_1, \ldots, W_n\}$. We construeren $E' = \{(W_i, W_j) \mid i, j \in [1, n], i \neq j, \exists \sigma_i \in W_i, \sigma_j \in W_j \text{ zodat } \sigma_i \prec \sigma_j\}$. Deze E' geeft dan de volgorde aan waarin we $W_1, \ldots, W_n$ gaan toepassen tijdens de chase.

We kunnen dan de chase over een willekeurige tableau T bekijken. We passen dan eerst de chase toe voor alle dependencies die voorkomen in cycli, dus $\{W_1, \ldots, W_n\}$. We weten dat we hiervoor een eindige chase sequence zullen vinden aangezien al deze cycli deel zijn van $CT_{\forall\exists}$. Dit geeft ons T'

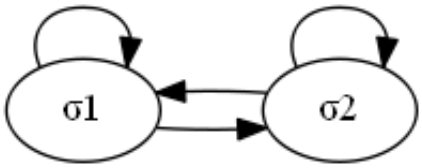
Als de chase van deze dependencies gedaan is moeten we nog de dependencies beschouwen die geen deel zijn van $\sim$. We kunnen deze dan nog toepassen op T' als dit mogelijk is. Dit zal ook eindig zijn aangezien geen van deze dependencies in een cyclus zit. Er is dus een eindig aantal keren dat deze uitgevoerd kunnen worden. Het resultaat van deze toepassingen T'' zal dan voldoen aan Σ. We hebben dus een eindige chase sequence gevonden. Dus $\Sigma \in CT_{\forall\exists}$.

Voorbeeld 5.1.1. *We beschouwen een verzameling dependencies $\Sigma = \{\sigma_1, \sigma_2\}$ met :*

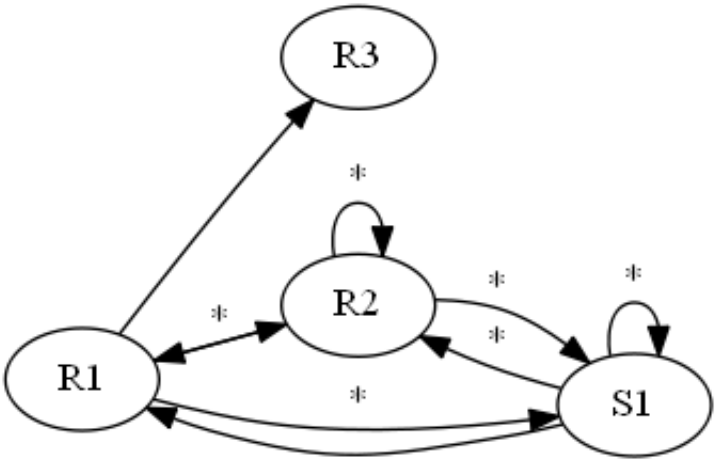
- $\sigma_1 = R(x_1, x_2, x_3) \wedge S(x_2) \rightarrow \exists y\ R(x_2, y, x_1) \wedge S(y)$
- $\sigma_2 = R(x_1, x_2, x_3) \wedge S(x_2) \rightarrow R(x_2, x_3, x_1) \wedge S(x_3)$

We zien meteen dat $\Sigma \notin CT_{\forall\forall}$. We kunnen immers een chase sequence opstellen waarbij we telkens eerst σ_1 toepassen en dus herhaaldelijk nieuwe variabelen introduceren.

Verder zien we ook dat we altijd σ_2 eerst zouden kunnen toepassen tijdens de chase over een willekeurig tableau. Aangezien σ_2 geen nieuwe gelabelde nullen introduceert zou dit leiden tot een eindige chase. Het resultaat van deze toepassingen zal ook voldoen aan



Figuur 5.1: *De chase graph voor Voorbeeld 5.1.1.*



Figuur 5.2: *De dependency graph voor Voorbeeld 5.1.1.*

σ_1. *We zien immers dat* x_3 *uit* σ_2 *zal dienen als waarde voor* y *uit* σ_1. *Er is dus een chase sequence waarvoor* Σ *eindigt en dus* $\Sigma \in CT_{\forall\exists}$.

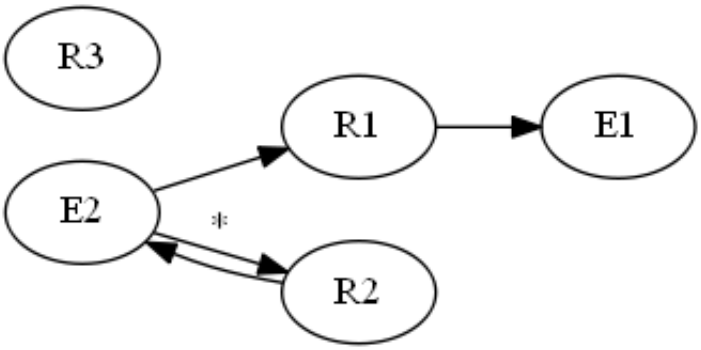
We kunnen nu controleren of Σ *gestratifieerd is. We stellen de chase graph* $G(\Sigma)$ *op voor* Σ. *De knopen van deze graaf zijn* $\{\sigma_1, \sigma_2\}$. *We weten dat er een boog is tussen twee dependencies als de* $\prec$ *relatie van toepassing is op deze dependencies. We zien dat alle dependencies elkaar kunnen afvuren aangezien ze bijna identiek zijn. Dit leidt ons tot de chase graph zichtbaar in Figuur 5.1. We moeten nu voor alle cycli uit de chase graph controleren of deze zwak acyclisch zijn. We kiezen ervoor om eerst* $\{\sigma_1\}$ *te bekijken. We zouden ook* $\{\sigma_1, \sigma_2\}$ *of* $\{\sigma_2\}$ *kunnen bekijken aangezien dit allemaal cycli zijn uit de chase graph. We testen dus of* $\{\sigma_1\}$ *zwak acyclisch is. Om dit te doen stellen we de dependency graph op. Deze is zichtbaar in Figuur 5.2. We zien dat deze dependency graph niet zwak acyclisch is.* Σ *is dus niet gestratifieerd.*

We hebben dus een voorbeeld gevonden van een Σ *die wel in* $CT_{\forall\exists}$ *zit maar niet gestratifieerd is.*

Voorbeeld 5.1.2. *Beschouw* $\Sigma = \{\sigma_1, \sigma_2\}$ *met*

- $\sigma_1 = E(x_1, x_2) \rightarrow \exists y R(x_2, y, c)$
- $\sigma_2 = R(x_1, x_2, d) \rightarrow E(x_1, x_2)$

Hierbij zijn c *en* d *constanten. Omdat* c *en* d *constanten zijn zien we dat* $\sigma_1 \not\prec \sigma_2$. σ_1 *kan immers enkel relatieatomen maken met* c *op de derde positie.* σ_2 *heeft een atoom nodig met* d *op de derde positie, dus het uitvoeren van* σ_1 *zal niet leiden tot het*



Figuur 5.3: *De dependency graph voor Voorbeeld 5.1.2.*

uitvoeren van σ_2. Verder zien we ook gemakkelijk in dat $\sigma_1 \not\prec \sigma_1$ en $\sigma_2 \not\prec \sigma 2$ aangezien ze verschillende relaties hebben in hun body en head.

De relatie die wel geldt is $\sigma_2 \prec \sigma_1$. Om dit aan te tonen maken we gebruik van tableau $T = R(a, b, d)$ en we kiezen $\overline{a} = \overline{b} = a, b$. We zien dat $T \not\models \sigma_2$ aangezien er een homomorfisme is van de body van σ_2 naar T maar niet voor de head van σ_2. We zien ook dat $T \models \sigma_1$. T bevat immers gaan atomen voor de relatie E dus T voldoet triviaal aan σ_1. We kunnen dan een chase stap toepassen op T door σ_2 toe te passen. Dit geeft ons $T' = \{R(a, b, d), E(a, b)\}$. Voor deze T' zien we dat er een homomorfisme van de body van σ_1 naar T' is. Dit is echter niet uitbreidbaar naar de constante c in de head van σ_1 dus $T' \not\models \sigma_1$. De voorwaarden zijn dus voldaan zodat $\sigma_2 \prec \sigma_1$.

De enige boog in onze chase graph wordt dus een boog van σ_2 naar σ_1. Aangezien er geen cycli in de chase graph zitten voldoet Σ triviaal aan stratificatie.

We zien echter dat Σ niet zwak acyclisch is. Dit doen we daar de dependency graph op te stellen voor Σ. Deze graaf is zichtbaar in Figuur 5.3. Deze graaf bevat een cyclus met een speciale boog en dus is Σ niet zwak acyclisch. We hebben dus een voorbeeld gevonden van een verzameling dependencies die niet zwak acyclisch is maar wel gestratifieerd.

Hoofdstuk 6

Studie van $CT_{\forall\forall}$

In dit hoofdstuk bestuderen we de verschillende technieken die de terminatie kunnen afdwingen voor de elementen van $CT_{\forall\forall}$.

We hebben reeds één voorwaarde voor de terminatie van de chase gezien in zwakke acycliciteit (Paragraaf 4.1). We bouwen nu verder op deze voorwaarde om meer en algemenere voorwaarden te vinden voor de terminatie.

6.1 Super zwakke acycliciteit

Super zwakke acycliciteit (SwA) is een verbetering van zwakke acycliciteit. Deze techniek werkt echter enkel met TGD's, de verzameling dependencies Σ mag dus geen EGD's bevatten. De techniek werd geïntroduceerd door Marnette in 2009[31].

6.1.1 Vereisten

Een eerste begrip dat we nodig hebben om tot de definitie van SwA te komen is *skolemizatie*. Dit is een techniek die de existentieel gekwantificeerde variabelen vervangt door een functie gebaseerd op de universeel gekwantificeerde variabelen. Voor een zekere existentieel gekwantificeerde variabele y en een dependency σ noteren we deze functie als f_y^σ. Het resultaat van de skolemizatie van een verzameling van dependencies Σ noteren we als $P(\Sigma)$. Dit is de verzameling van alle dependencies uit Σ met skolemizatie toegepast op deze dependencies.

Voorbeeld 6.1.1. *We bekijken de skolemizatie van een TGD* $\sigma = \forall x_1, x_2(R(x_1, x_2) \rightarrow \exists y\ R(x_2, y))$. *We kunnen dan* y *vervangen door* $f_y^\sigma(x_1, x_2)$. *We krijgen dan als resultaat de TGD* $\sigma' = \forall x_1, x_2(R(x_1, x_2) \rightarrow R(x_2, f_y^\sigma(x_1, x_2)))$. *We zien dus dat dit resultaat inderdaad geen existentieel gekwantificeerde variabelen meer bevat.*

Een volgende begrip waarvan SwA gebruik maakt is een *plaats*. Neem een verzameling Σ van TGD's. Neem $P(\Sigma)$ de skolemization van Σ. Een plaats (a, i) is een paar waarbij a een atoom is van $P(\Sigma)$ en $1 \leq i \leq ar(a).i$ is de index van een element in a.

We introduceren nu een manier om deze plaatsen met elkaar te vergelijken. Algemeen is *unification* het probleem waarbij we voor twee beschrijvingen x en y een z zoeken die aan beide beschrijvingen voldoet[27].

In deze context is een term een variabele, een constante, een functiesymbool of een relatie symbool gevolgd door een reeks van termen gescheiden door komma's en omringd door haakjes. We zien dat de atomen die voorkomen in een plaats dus ook een term zijn.

Voorbeeld 6.1.2. *Een voorbeeld van een term is* $R(x, f(x))$ *of* $f(R(a, b), S(u, v, w))$.

Een substitutie is een functie die variabelen afbeeldt op termen. We gaan deze noteren met s of s'. We noteren een substitutie s die x afbeeldt op $R(a, a)$ en y op $(R(b, f(b))$ als $s = \{(x, R(a, a)), (y, R(b, f(b)))\}$.

We zeggen dan voor twee termen t en t' dat ze *unifiable* zijn als er een substitutie s bestaat zodat $s(t) = s(t')$.

We definiëren dit begrip nu in de context van plaatsen. Hierbij gebruiken we de atomen van de plaatsen als de termen die unifiable moeten zijn en eisen we dat de index van de twee plaatsen overeenkomt. Twee plaatsen (a, i) en (a', i') zijn dan *unifiable*, genoteerd als $(a, i) \sim (a', i')$ als en slechts als $i = i'$ en er substituties s en s' zijn zodat $s(a) = s'(a')$.

Voorbeeld 6.1.3. *) Veronderstel twee plaatsen* $(R(x, x), 1)$ *en* $(R(x, y), 1)$. *We zien dan dat* $(R(x, x), 1) \sim (R(x, y), 1)$. *We kiezen* $s = \{(x, x)\}$ *en* $s' = \{(x, x), (y, x)\}$. *We zien dan dat* $s(R(x, x)) = s'(R(x, y)) = R(x, x)$. *De voorwaarden voor* $\sim$ *zijn dus voldaan.*

Veronderstel een plaats die het resultaat is van een skolemizatie: $(R(x, f_x^\sigma(x)), 1)$. *We kunnen deze plaats dan vergelijken met onze eerste plaats* $(R(x, x), 1)$. *Voor deze plaatsen geldt dan dat* $(R(x, x), 1) \not\sim (R(x, f_x^\sigma(x)), 1)$. *Onze substitutie zal immers de variabelen wel kunnen vervangen maar het functiesymbool zal niet veranderd kunnen worden. Bijvoorbeeld voor* $s = s' = \{(x, t)\}$ *krijgen we* $R(t, t) \neq R(t, f_x^\sigma(t))$.

Voor een dependency $\sigma \in \Sigma$ en een existentiële variabele y in de head van σ noteren we met $Out(\sigma, y)$ de verzameling plaatsen uit de head van $P(\sigma)$ waar een term van de vorm $f_y^\sigma(\ldots)$ voorkomt.

Voorbeeld 6.1.4. *We kiezen een TGD* $\sigma = S(x) \to \exists y\ R(x, y) \wedge R(y, x)$. *Als we skolemizatie toepassen op deze TGD krijgen we* $P(\sigma) = S(x) \to R(x, f_y^\sigma(x)) \wedge R(f_y^\sigma(x), x)$. *We zien dan dat* $Out(\sigma, y) = \{(R(x, f_y^\sigma(x)), 2), (R(f_y^\sigma(x), x), 1)\}$.

Voor een universeel gekwantificeerde variabele x in de body van σ definiëren we $In(\sigma, x)$, dit is de verzameling plaatsen uit de body van σ waarin x voorkomt.

Voorbeeld 6.1.5. *We hernemen de gegevens uit Voorbeeld 6.1.4. We zien dan dat voor deze* σ *de verzameling* $In(\sigma, x) = \{(S(x), 1)\}$.

We kunnen ook twee verzamelingen van plaatsen Q, Q' met elkaar vergelijken. We zeggen dat $Q \triangleleft Q'$ als en slechts als voor alle $q \in Q$ er een $q' \in Q'$ bestaat zodat $q \sim q'$. Intuïtief wil dat zeggen dat voor alle plaatsen uit Q er een plaats is in Q' die unifiable is.

We introduceren verder ook nog de verzameling $Move(\Sigma, Q)$ voor een verzameling dependencies Σ en een verzameling plaatsen Q. Dit is de kleinste verzameling van plaatsen Q' zodat $Q \subseteq Q'$ en voor alle regels $r : B_r \to H_r \in P(\Sigma)$ en voor alle variabelen x geldt dat als $\gamma_x(B_r) \lhd Q'$ dan $\gamma_x(H_r) \subseteq Q'$. Hierbij staat B_r voor de body van r, H_r voor de head van r, $\gamma_x(B_r)$ en $\gamma_x(H_r)$ voor de verzameling plaatsen in respectievelijk B_r en H_r waar x voorkomt.

De opbouw van deze $Move(\Sigma, Q)$ verloopt dus als volgt: we vertrekken vanuit de gegeven verzameling Q. We weten dat $Q \subseteq Q'$ dus om te beginnen is $Q' = Q$. Voor de dependencies uit Σ zoeken we dan de dependency zodat $\gamma_x(B_r) \lhd Q'$. Als de body van deze dependency unifiable is naar Q' dan kunnen we de head toevoegen aan Q' zodat $\gamma_x(H_r) \subseteq Q'$. Op deze manier zoeken we dan de kleinste verzameling die voldoet aan alle voorwaarden.

Voorbeeld 6.1.6. *We hernemen opnieuw de data uit Voorbeeld 6.1.4. We kunnen dan* $Move(\Sigma, \{(S(a), 1)\})$ *bepalen. We vertrekken vanuit* $Q' = Q$ *dus vanuit* $\{(S(a), 1)\}$. *Van hieruit zien we dat* $\gamma_x(B_\sigma) \lhd Q'$. *We kunnen immers van* $(S(a), 1)$ *naar* $(S(x), 1)$ *gaan via een substitutie* s *die* a *op* x *afbeeldt. Onze* $Move$ *verzameling moet dan dus voldoen aan* $\gamma_x(H_\sigma) \subseteq Q'$. *We voegen dus alle plaatsen uit* $\gamma_x(H_\sigma)$ *toe aan* Q'.
Deze plaatsen zijn: $\{(R(x, f_y^\sigma(x)), 1), (R(x, f_y^\sigma(x)), 2), (R(f_y^\sigma(x), x), 1),$
$(R(f_y^\sigma(x), x), 2)\}$.
Na deze toevoeging voldoet onze Q' *aan alle voorwaarden. We krijgen dus als resultaat dat:*

$$Move(\Sigma, \{(S(a), 1)\} = \{(S(a), 1), (R(x, f_y^\sigma(x)), 1), (R(x, f_y^\sigma(x)), 2),$$
$$(R(f_y^\sigma(x), x), 1), (R(f_y^\sigma(x), x), 2)\}$$

We maken nu gebruik van de begrippen In, Out, en $Move$ om tot de volgende definitie te komen: Voor een gegeven verzameling van TGD's Σ met $\sigma_1, \sigma_2 \in \Sigma$ zeggen we dat σ_1 *vuurt* σ_2, genoteerd als $\sigma_1 \hookrightarrow_\Sigma \sigma_2$, als er een existentieel gekwantificeerde variabele y bestaat in de head van σ_1 en een universeel gekwantificeerde variabele x in de head en de body van σ_2 zodat $In(\sigma_2, x) \lhd Move(\Sigma, Out(\sigma_1, y))$.

We zeggen in dit geval dat σ_1 vuurt σ_2 omdat als er voldaan is aan deze vereisten de uitvoering van σ_1 kan leiden tot een uitvoering van σ_2. Verder merken we nog op dat σ_1 minstens één existentiële variabele moet beschikken om aan de definitie te voldoen. Hier zijn we dus het verband met de creatie van variabelen. Deze definitie is dan ook waarop SwA bouwt. We gaan de chase op een symbolische manier benaderen met behulp van deze relatie. Op deze manier kunnen we op voorhand de verbanden tussen de verschillende TGD's bekijken.

6.1.2 Definitie

Met behulp van de voorafgaande definities kunnen we nu tot de formele definitie van SwA komen. Voor een gegeven set van TGD's Σ zeggen we dat Σ *super weak acyclisch* is als en slechts als de vuur-relatie $\hookrightarrow_\Sigma$ acyclisch is.

SwA garandeerdt de terminatie van de chase voor elke chase sequence en elke instance[31].

Analoog aan zwakke acyclicteit gaat SwA ook op een syntactische manier afdwingen dat er geen creatie van nieuwe variabelen mogelijk is. We zoeken de plaatsen waar de

creatie van nieuwe variabelen mogelijk is en zorgen ervoor dat er geen cycli bestaan tussen deze plaatsen. De creatie van nieuwe variabelen zal dan niet telkens opnieuw gebeuren omdat er geen cycli mogen zijn waarin deze creatie gebeurt.

Belangrijk is dat SwA inderdaad een uitbreiding is van zwakke acycliciteit. Dit is duidelijk zichtbaar in Voorbeeld 6.1.7.

Om de studie van SwA af te sluiten geven we een voorbeeld dat tot stand kwam door een samenwerking tussen Meier en Marnette en dat aantoont dat SwA zwakker is dan zwakke acycliciteit [31, 32]. We werken de details van dit voorbeeld verder uit.

Voorbeeld 6.1.7. *We vertrekken vanuit* $\Sigma = \{\sigma_1, \sigma_2\}$ *met:*

- $\sigma_1 = A(x) \rightarrow \exists y\ B(x,y) \wedge B(y,x) \wedge C(y)$.
- $\sigma_2 = B(x,x) \wedge C(y) \rightarrow A(x) \wedge C(y)$.

Het toepassen van skolemizatie geeft ons dan $P(\Sigma)$ *bestaande uit de volgende twee dependencies:*

- $A(x) \rightarrow B(x, f_y^{\sigma_1}(x)) \wedge B(f_y^{\sigma_1}(x), x) \wedge C(f_y^{\sigma_1}(x))$
- $B(x,x) \wedge C(y) \rightarrow A(x) \wedge C(y)$

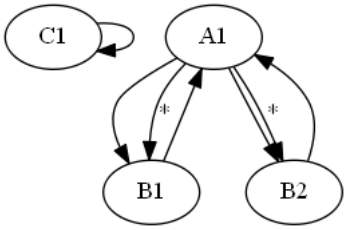
We beginnen nu met het berekenen van de vuur-relatie $\hookrightarrow_\Sigma$*. Om deze relatie te controleren hebben we de* In*,* Out *en* $Move$ *verzamelingen nodig. We berekenen dus eerste deze verzamelingen.*

- $In(\sigma_1, x) = \{(A(x), 1)\}$
- $In(\sigma_2, x) = \{(B(x,x), 1), (B(x,x), 2)\}$
- $In(\sigma_2, y) = \{(C(y), 1)\}$
- $Out(\sigma_1, y) = \{(B(x, f_y^{\sigma_1}(x)), 2), (B(f_y^{\sigma_1}(x), x), 1), (C(f_y^{\sigma_1}(x), 1)\}$
- $Out(\sigma_2, y) = \emptyset$

We zien dat σ_2 *geen existentiële variabelen bevat. Dit wil zeggen dat* σ_2 *nooit het startpunt kan zijn van de vuur-relatie. We weten dus dat* $\sigma_2 \not\hookrightarrow_\Sigma \sigma_1$*. We kijken dus naar het andere geval namelijk* $\sigma_1 \hookrightarrow \sigma_2$*. Hier zien we echter het probleem opduiken dat* σ_1 *en* σ_2 *gebruik maken van verschillende relatie symbolen in body and head. Ze kunnen dus niet overeen komen.*

We kijken dus nog naar $\sigma_1 \hookrightarrow \sigma_1$*. We zien bij de uitwerking dat* $Move(\Sigma, Out(\sigma_1, y)) = Out(\Sigma_1, y)$*. We kunnen immers geen dependency toepassen. We controleren dan of* $In(\sigma_1, x) \triangleleft Move(\Sigma, Out(\sigma_1, y))$*. Dit is echter niet het geval want er is geen subsitutie van* $B(x,x)$ *naar* $B(x, f_y^{\sigma_1}(x)$ *en* $B(f_y^{\sigma_1}(x), x))$*. We zien dus dat de vuur-relatie* $\hookrightarrow_\Sigma = \emptyset$*. En dus is* Σ *super zwak acyclisch.*

We kunnen ook nog controleren dat dit voorbeeld inderdaad niet zwak acyclisch is. Dit doen we door de dependency graph op te stellen voor Σ*. Deze dependency graph* $dep(\Sigma)$ *zou dan een cyclus moeten bevatten die een van de speciale* $\overset{*}{\rightarrow}$ *bevat. We construeren*



Figuur 6.1: *De dependency graph voor Voorbeeld 6.1.7*

dus dep(Σ), deze is te zien in Figuur 6.1. In deze graaf zien we dat er duidelijk cyclussen bestaan die gebruik kunnen maken van een speciale boog. Σ voldoet dus niet aan zwakke acycliciteit maar wel aan SwA. We zien dus dat SwA inderdaad zwakker is dan zwakke acycliciteit.

6.2 C-Stratificatie

C-stratificatie is een verbetering geïntroduceerd door Meier van de reeds eerder bekeken stratificatie (Paragraaf 5.1) van Deutsch [9, 32]. Het achterliggende idee blijft hetzelfde. We willen de verzameling dependencies opdelen op een manier zodat er vanuit de deelverzamelingen een conclusie gemaakt kan worden over de terminatie van de volledige verzameling.

6.2.1 Oblivious Chase

De originele definitie van stratificatie maakte gebruik van een chase step. Voor deze nieuwe vorm van stratificatie gaan we gebruik maken van de *oblivious chase*. Dit is een variant van de chase. Er is slechts één verschil met de originele versie van de chase. Dit verschil is dat we een dependency altijd gaan toepassen als er een homomorfisme is van de body van de dependency naar het tableau T waarop we aan het chasen zijn. We controleren dus niet of er een homomorfisme is van de head van deze dependency naar T of niet. We noemen dit soort chase dan oblivious omdat deze onbewust is van de head van de dependency. De dependency wordt altijd toegepast zodra er een homomorfisme is van de body.

Merk op dat het ook voor de oblivious chase onmogelijk is om de terminatie te bepalen. Het bewijs hiervoor is analoog aan het bewijs dat de terminatie van de gewone chase onbeslisbaar is. Gogacz heeft aangetoond dat er een reductie is van het halting probleem voor eindige automaten naar dit probleem [16].

We maken gebruik van de oblivous chase omdat deze meer gevallen omvat dan de gewone chase. Dit zal dus leiden tot striktere voorwaarden in onze terminatie condities. Stratificatie was immers slechts een terminatie conditie voor $CT_{\forall\exists}$ en we zijn nu op zoek naar terminatie condities voor $CT_{\forall\forall}$. Het is dus logisch dat we hiervoor striktere condities zullen moeten toepassen.

6.2.2 Definitie

Voor twee TGD's of EGD's σ_1 en $\sigma_2 \in \Sigma$ zeggen we dat $\sigma_1 \prec_c \sigma_2$ als en slechts als er een tableau T en tupels $\overline{a}$ en $\overline{b}$ bestaan zodat:

- $T \models \sigma_2(\overline{b})$
- Er is een chase step in de oblivious chase die van T naar T' gaat door de toepassing van σ_1 op $\overline{a}$.
- $T' \not\models \sigma_2(\overline{b})$

We geven een voorbeeld van een dergelijke situatie.

Voorbeeld 6.2.1. *We vertrekken vanuit de volgende gegevens:*

- $\sigma_1 = R(x_1, x_2) \rightarrow \exists y\ S(x_1, y)$
- $\sigma_2 = S(x_1, x_2) \wedge S(x_1, x_3) \rightarrow T(x_1)$
- $T = \{R(a, b), S(a, b)\}$
- $\overline{a} = a, b$
- $\overline{b} = a, b, n_1$

We zien dan dat $T \models \sigma_2$. Dit is triviaal voldaan aangezien T geen tegenspraak bevat voor σ_2. We kunnen dan T' maken door middel van een chase step die σ_1 toepast voor $\overline{a}$. De toepassing van σ_1 leidt tot de toevoeging van $S(a, n_1)$. Dus $T' = \{R(a, b), S(a, b), S(a, n_1)\}$. We zien dan dat $T' \not\models \sigma_2$. We vinden immers wel $S(a, b)$ en $S(a, n_1)$ in T' maar geen $T(a)$ ook al zou deze er volgens σ_2 in moeten zitten.

Voor dit voorbeeld zien we dus dat $\sigma_1 \prec_c \sigma_2$ aangezien aan alle voorwaarden voldaan is.

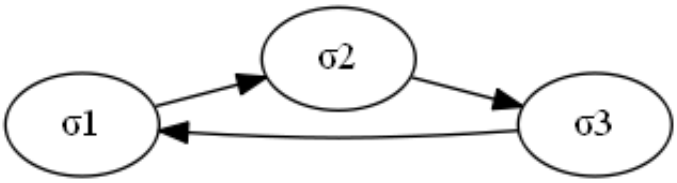
Met behulp van deze definitie kunnen we de *c-chase graph* $G_c(\Sigma)$ voor een verzameling dependencies Σ definiëren. $G_c(\Sigma) = (\Sigma, E)$. De knopen zijn dus de dependencies uit Σ. Verder is er een boog in E tussen twee dependencies $\sigma_1, \sigma_2 \in \Sigma$ als $\sigma_1 \prec_c \sigma_2$. Deze boog gaat van σ_1 naar σ_2.

We zeggen dat Σ *c-stratified* is als en slechts als de dependencies in elke cyclus van $G_c(\Sigma)$ zwak acyclisch zijn.

Voorbeeld 6.2.2. *We kunnen c-stratificatie bekijken voor de set van TGD's $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ met:*

- $\sigma_1 = R(x_1) \rightarrow S(x_1, x_1)$
- $\sigma_2 = S(x_1, x_2) \rightarrow \exists z\ T(x_2, z)$
- $\sigma_3 = T(x_1, x_2) \wedge T(x_2, x_3) \rightarrow R(x_3)$

We kunnen $G_c(\Sigma)$ opstellen. Deze is zichtbaar in Figuur 6.2. We zien dat deze slechts bestaat uit een enkele component die niet zwak acyclisch is. Σ is dus niet aan c-stratified. De terminatie is dus niet zeker. We zien aan σ_2 waarom dit het geval is. We kunnen immers een opbouw van gelabelde nullen krijgen door σ_2 telkens opnieuw toe te passen.



Figuur 6.2: *De c-chase graph voor* Σ *uit Voorbeeld 6.2.2*

Belangrijk is dat c-stratifiatie een uitbreiding is van zwakke acycliciteit. Dit is te zien in Voorbeeld 6.2.3.

Voorbeeld 6.2.3. *We geven een voorbeeld van een verzameling dependencies* Σ *die niet zwak acyclisch is maar wel c-gestratifieerd.* $\Sigma = \{\sigma\}$ *met*

$$E(x_1, x_2) \wedge E(x_2, x_1) \rightarrow \exists y_1, y_2 E(x_1, y_1) \wedge E(y_1, y_2) \wedge E(y_2, x_1)$$

Het is duidelijk in te zien dat Σ *niet zwak acyclisch is. Als we deze dependency bekijken in de context van een graaf dan gaat* σ *afdwingen dat er voor elke cyclus van lente 2 ook een cyclus van lengte 3 is. Als we een tableau* T *hebben zodat* $T \models \sigma$ *dan weten we dat elke cyclus van lengte 2 dus ook een cyclus van lengte 3 heeft. We kunnen echter geen nieuwe cycli van lengte 2 maken door de toepassing van* σ*. Er is dus geen* T' *die het resultaat is van de toepassing van* σ *in de oblivious chase op* T *zodat* $T' \not\models \sigma$*.We zien dus dat* $\sigma \not\prec_c \sigma$*. De c-chase graph van* Σ *bevat dus geen enkele boog. Er zijn dus geen cycli in de c-chase graph die zwak acyclish moeten zijn. Het is dus triviaal dat* Σ *c-gestratifieerd is.*

Verder is het ook belangrijk om te controleren dat verzamelingen van dependencies Σ die c-gestratifieerd zijn inderdaad vervat zitten in $CT_{\forall\forall}$[32]. We schetsen kort de intuïtie achter dit bewijs.
De reden dat dit het geval is, is analoog aan de reden bij stratificatie. We kunnen immers weer terminatie afdwingen door gebruik te maken van de cycli in Σ. We weten immers voor elk van deze cycli dat ze voldoen aan zwakke acycliciteit. We weten dus dat voor deze cycli alle uitvoeringen van de chase tot een terminale chase sequence zullen leiden. We hoeven dan enkel nog rekening te houden met de dependencies die niet voorkomen in een cyclus. Aangezien deze niet voorkomen in een cyclus kunnen ze slechts een eindig aantal keren uitgevoerd worden. De uiteindelijke chase sequence voor Σ zal dus een combinatie zijn van de eindige chase sequence voor de cycli van Σ en de eindige toepassing van de dependencies die niet voorkomen in een cyclus van Σ. Vermits beide onderdelen van deze combinatie eindig zijn zal het resultaat ook eindig zijn.

We merken verder nog op dat testen voor c-stratificatie gedaan kan worden in coNP.

6.3 Safe dependencies

We introduceren de klasse van de *safe* dependencies. Dit is een nog een andere extensie van zwakke acycliciteit die terminatie van de chase garandeert.

Om tot dit concept te komen maken we gebruik van *affected positions*, geintroduceerd door Cali[5]. De definitie is als volgt: Voor een gegeven verzameling TGD's Σ is de

verzameling van affected positions aff(Σ) inductief gedefinieerd als volgt: Neem p een positie in de head van een TGD $\sigma \in \Sigma$.

- Als er een existentieel gekwantificeerde variabele voorkomt op positie p dan $p \in$ aff(Σ).
- Als er eenzelfde universeel gekwantificeerde variabele x voorkomt op positie p en in de body van σ enkel voorkomt op affected positions dan $p \in$ aff(Σ).

Intuïtief zien we dat deze affected positions een overschatting zijn van de mogelijke posities waar nieuw geïntroduceerde gelabelde nullen kunnen voorkomen tijdens de chase.

Gebruikmakend van deze affected positions komen we tot de definitie van een *propagation graph* prop(Σ) voor een verzameling dependencies Σ. prop(Σ) = (aff(Σ), E). De knopen zijn dus de affected positions van Σ. Analoog aan zwakke acycliciteit bestaat E uit twee verschillende soorten bogen. We kunnen deze bogen toevoegen op de volgende manier: voor elke TGD $\forall \overline{x}(\phi(\overline{x}) \rightarrow \exists y\ \psi(\overline{x}, \overline{y}))$ en voor elke x in $\overline{x}$ die voorkomt in ψ en voor elk voorkomen van x op positie p_1 in ϕ:

- Als x enkel voorkomt op affected positions van ϕ dan voor elk voorkomen van x op positie p_2 in ψ voegen we een boog toe van de vorm $p_1 \rightarrow p_2$.
- Als x enkel voorkomt op affected positions van ϕ dan voor elke existentieel gekwantificeerde variabele y and voor elke voorkomen van y op positie p_2 voegen we een boog toe van de vorm $p_1 \stackrel{*}{\rightarrow} p_2$.

Een verzameling dependencies Σ is dan *safe* als en slechts als prop(Σ) geen cycli heeft die een speciale boog bevatten.

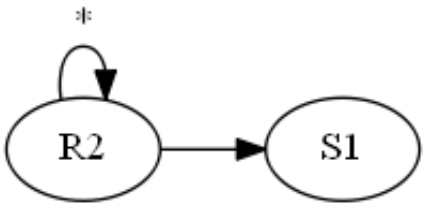
We zien dus dat deze definitie zeer analoog is met die van zwakke acycliciteit met het verschil dat we nu gebruik maken van de propagation graph en niet van de dependency graph.

Voorbeeld 6.3.1. *We bekijken een verzameling TGD's* $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ *met:*

- $\sigma_1 = R(x_1, x_2) \rightarrow \exists y\ R(x2, y)$
- $\sigma_2 = R(x_1, x_2) \rightarrow S(x_2)$
- $\sigma_3 = S(x_1) \wedge R(x_1, x_2) \rightarrow T(x_1)$

We beginnen met het berekenen van aff(Σ). *We weten dat positie* R_2 *affected is want in* σ_1 *komt deze positie voor met een existentiele kwantor. Als* R_2 *affected is dan zien we dat als gevolg heeft dat* S_1 *ook affected is. Dit komt doordat in* σ_2 *de positie* S_1 *het resultaat is van de affected position* R_2. *We zien immers dat* x_2 *enkel voorkomt in affected positions in de body van* σ_2. *Voor* σ_3 *zien we dat* T_1 *niet toegevoegd wordt aan de verzameling affected positions aagezien* x_1 *voorkomt op positie* R_1 *en dit is geen affected position. We zien dus dat:*

$$\mathit{aff}(\Sigma) = \{R_2, S_1\}$$



Figuur 6.3: *De propagation graph prop(Σ) voor Voorbeeld 6.3.1*

Voor deze posities kunnen we nu de propagation graph prop(Σ) opstellen. De knopen van deze graph zijn R_2 en S_1. We zien dat er een speciale boog is van R_2 naar R_2 en een gewone boog van R_2 naar S_1. De resulterende graaf is te zien in Figuur 6.3.

We zien dat prop(Σ) een cyclus bevat die een special boog heef namelijk $R_2 \xrightarrow{*} R_2$. We zien dus dat Σ niet safe *is en de terminatie van de chase niet gegarandeerd is.*

Dit resultaat is logisch aangezien de constante toepassing van TGD σ_1 zou leiden tot een oneindige chase. We kunnen hiermee immers telkens nieuwe variabelen genereren.

Belangrijk aan deze terminatie conditie is dat het een uitbreiding is van zwakke acyclici-teit. We merken hierbij op dat prop(Σ) $\subseteq$ *dep*(Σ). Als Σ zwak acyclisch is dan zal Σ dus ook $safe$ zijn. Omgekeerd is dit echter niet het geval zoals zichtbaar in Voorbeeld 6.3.2. Er bestaan verzamelingen van dependencies die niet zwak acyclisch zijn maar wel safe.

Voorbeeld 6.3.2. *We bekijken $\Sigma = \{\sigma\}$. Hierbij is $\sigma = R(x_1, x_2, x_3) \wedge S(x_2) \rightarrow \exists y R(x_2, y, x_1)$.*

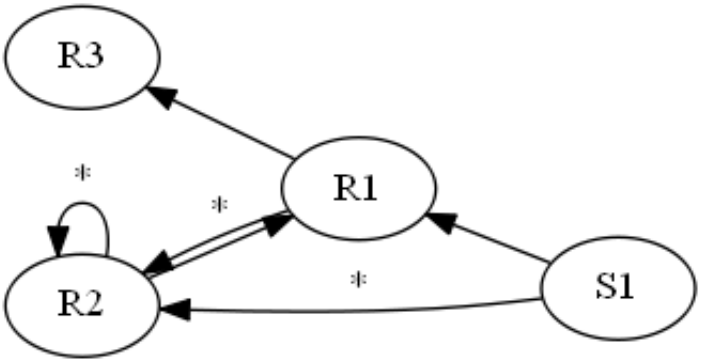
We controleren eerst of Σ zwak acyclisch is. Dit doen we door de dependency graph op te stellen. Deze is zichtbaar in Figuur 6.4. We zien dat deze verschillende cycli met een speciale boog bevat en dat Σ dus niet zwak acyclisch is.

We controleren nu of Σ $safe$ is. We berekenen eerst de affected positions van Σ. We zien dat R.2 een affected position is aangezien hier een existentieel gekwantificeerde variabele voorkomt. Verder zijn er geen affected positions.De proposition graph bevat dus enkel deze positie, zoals zichtbaar is in Figuur 6.5. prop(Σ) bevat dus duidelijk geen cyclus die gebruik maakt van een speciale boog. Σ is dus $safe$.

We zien dus dat Σ niet zwak acyclish is maar wel safe.

Verder is het ook belangrijk dat Σ is $safe \implies \Sigma \in CT_{\forall\forall}$. De reden hiervoor is analoog aan deze voor SwA. We maken gebruik van het feit dat we weten dat alle cycli reeds in $CT_{\forall\forall}$ zitten. We tonen dan aan dat we volgens de structuur van prop(Σ) een chase sequence kunnen opstellen. De chase van alle cycli is eindig aangezien ze in $CT_{\forall\forall}$ zitten. De dependencies die niet in een cyclus zitten kunnen slechts een eindig aantal keer toegepast worden. De chase over de volledige Σ zal dus ook eindig zijn.

We kunnen ons nu ook nog afvragen of er een verband is tussen c-stratificatie en $safe$ dependencies. We vinden echter dat ze niet met elkaar vergelijkbaar zijn. C-stratificatie impliceert dus niet het $safe$ zijn van een verzameling dependencies noch omgekeerd. Om dit aan te tonen geven we een voorbeeld van een verzameling dependencies die $safe$ is maar niet c-gestratifieerd en een verzameling dependencies die niet $safe$ is maar wel c-gestratifieerd.



Figuur 6.4: *De dependency graph dep(Σ) voor Voorbeeld 6.3.2*

Figuur 6.5: *De propagation graph prop(Σ) voor Voorbeeld 6.3.2*

Voorbeeld 6.3.3. *We beschouwen een verzameling dependencies* $\Sigma = \{\sigma_1, \sigma_2\}$ *met*

- $\sigma_1 = R(x_1, x_2, x_3) \wedge S(x_2, x_3) \rightarrow \exists y R(x_2, y, x_1)$
- $\sigma_2 = R(x_1, x_2, x_3) \rightarrow S(x_1, x_3)$

We kunnen voor dit voorbeeld aantonen dat $\sigma_1 \prec_c \sigma_2$ *en* $\sigma_2 \prec_c \sigma_1$*. Verder is* Σ *niet zwak acyclisch. We zien dus dat* Σ *niet c-gestratifieerd is. We zien echter dat* Σ *wel safe is.* $R.2$ *is de enige affected position voor de dependencies in* Σ*. De propagation graph komt dus overeen met Figuur 6.5.* Σ *is dus niet c-gestratifieerd maar wel* $safe$*.*

Voorbeeld 6.3.4. *We beschouwen een verzameling dependencies* $\Sigma = \{\sigma\}$ *met*

$$\sigma = E(x_1, x_2) \wedge E(x_2, x_1) \rightarrow \exists y_1, y_2 E(x_1, y_1) \wedge E(y_1, y_2) \wedge E(y_2, x_1)$$

We zien dan dat Σ *c-gestratifieerd is en niet zwak acyclisch (zie Voorbeeld 6.2.3). Als we testen of de dependency safe is zien we dat de aff(*Σ*)=* $\{E.1, E.2\}$*. We zien dan dat de propagation graph gebruik maak van alle posities dus dep(*Σ*) = prop(*Σ*).* Σ *is niet zwak acyclisch en dus ook niet safe.*

6.4 Safely restricted dependencies

We zullen de methode van c-stratificatie uitbreiden naar het begrip *safe restriction.* Dit begrip bouwt op het verdiepen van de $\prec_c$ relatie.

We benoemen de verzameling van posities uit Σ $pos(\Sigma)$. We kunnen dan het begrip null-pos definieren: Voor een verzameling Σ van dependencies, T een tableau en $N \subseteq V$ is null-pos$(N, T) = \{p \in pos(\Sigma) \mid a \in N, a$ komt voor op positie p in $T\}$.

We zien dus dat null-pos(N, T) de verzameling is van posities uit I die ook voorkomen in Σ en waarop variabelen uit N voorkomen.

We definiëren dan $\prec_P$ als verfijning van $\prec_c$. Zij Σ een verzameling dependencies en $P \subseteq pos(\Sigma)$. Voor alle $\sigma_1, \sigma_2 \in \Sigma$ zeggen we dat $\sigma_1 \prec_P \sigma_2$ als en slechts er tupels $\overline{a}$ en $\overline{b}$ en een tableau T_0 zijn zodat:

- $T_0 \models \sigma_2(\overline{b})$
- Er is een chase step in de oblivious chase die van T_0 naar T_1 gaat door de toepassing van σ_1 op $\overline{a}$.
- $T_1 \not\models \sigma_2(\overline{b})$
- Er is een $n \in \overline{b} \cap \Delta_{null}$ in de head van $\sigma_2(\overline{b})$ zodat null-pos$(\{n\}, T_0) \subseteq P$.

We zien dat de eerste drie voorwaarden overeenkomen met deze van c-stratificatie. Het verschil schuilt dus in de laatste voorwaarde. Deze controleert of er een mogelijk gevaarlijke null voorkomt. Merk op dat de P van $\prec_P$ aangeeft met welke verzameling posities we werken. Als we werken met een verzameling f van posities dan krijgt het symbool de vorm $\prec_f$.

We zien dat $\prec_c$ een speciaal geval is van $\prec_P$. Als we $P = pos(\Sigma)$ nemen dan bevat P alle posities uit Σ. Dit zal er voor zorgen dat de laatste voorwaarde uit de definitie altijd voldaan is. We moeten dan enkel de eerste 3 voorwaarden beschouwen en deze komen exact overeen met de voorwaarden voor $\prec_C$.

Met deze definitie kunnen we ons begrip van *affected positions* uitbreiden naar de posities relatief aan een dependency en een verzameling van posities. Voor een verzameling posities P en een TGD σ is aff-cl(σ, P) de verzameling posities p uit de head van σ zodat:

- Voor elke universeel gekwantificeerde variabele x op positie p: x voorkomt in de body van σ op posities uit P of,
- p een existentieel gekwantificeerde variabele bevat.

We maken gebruik van deze definitie en de verfijning van $\prec_c$ om een restrictie systeem te definiëren. Dit systeem is een generalisatie van de c-chase graph (Paragraaf 6.2.2).

Een *2-restrictie systeem* is een koppel $(G'(\Sigma), f)$ waarbij $G'(\Sigma) = (\Sigma, E)$ een gerichte graaf en $f \subseteq pos(\Sigma)$ zodat:

1. voor alle TGD's σ_1 en voor alle $(\sigma_1, \sigma_2) \in E$: aff-cl$(\sigma_1, f) \cap pos(\Sigma) \subseteq f$
2. voor alle TGD's σ_2 en voor alle $(\sigma_1, \sigma_2) \in E$: aff-cl$(\sigma_2, f) \cap pos(\Sigma) \subseteq f$
3. voor alle $\sigma_1, \sigma_2 \in \Sigma : \sigma_1 \prec_f \sigma_2 \implies (\sigma_1, \sigma_2) \in E$

Intuïtief zien we dat deze voorwaarden afdwingen dat alle beschouwde posities in f zitten. De affected positions moeten immers een deel zijn van f voor beide TGD's. Verder doet de laatste voorwaarde ons terugdenken aan de c-chase graph. Er is immers een boog tussen σ_1 en σ_2 als $\sigma_1 \prec_f \sigma_2$.

We noemen een dergelijk 2-restrictie systeem *minimaal* als het verkregen kan worden vanuit $((\Sigma, \emptyset), \emptyset)$ door toepassing van de voorwaarden uit de eerste twee punten totdat alle voorwaarden gelden. Dit wil zeggen dat bijvoorbeeld voor de eerste voorwaarde de verzameling van posities f enkel uitgebreid wordt met de posities die nodig zijn om aan de voorwaarde te voldoen.

We geven nu twee voorbeelden van 2-restrictie systemen. Ons eerste voorbeeld toont aan dat er altijd een 2-restrictie systeem bestaat.

Voorbeeld 6.4.1. *Neem Σ een willekeurige verzameling dependencies. Dan is $((\Sigma, \Sigma \times \Sigma), f)$ met $f = pos(\Sigma)$ een 2-restrictie systeem voor Σ. We kunnen de voorwaarden afgaan om te controleren dat dit het geval is. De eerste vereiste is dat al onze affected positions aff-cl(σ_1, f) en aff-cl(σ_2, f) vervat moeten zijn in f. Aangezien alle mogelijke posities in f zitten zijn deze er dus ook een deel van. De eerste twee voorwaarden zijn dus voldaan. Dan moeten we nog controleren dat voor elke σ_1 en σ_2 uit Σ geldt dat $\sigma_1 \prec_f \sigma_2 \implies (\sigma_1, \sigma_2) \in E$. We zien echter dat E alle mogelijke bogen bevat. Elke twee knopen zijn dus met elkaar verbonden. Dit wil zeggen dat $(\sigma_1, \sigma_2) \in \Sigma$ zelfs als $\sigma_1 \not\prec_f \sigma_2$. Dit is dus een manier om een 2-restrictie systeem te maken voor een willekeurige verzameling dependencies Σ. Merk op dat dit systeem meestal niet minimaal zal zijn. We voegen immers alle mogelijke bogen toe aan de graaf dus waarschijnlijk zijn er ook bogen voor gevallen waar $\sigma_1 \not\prec_f \sigma_2$. Deze bogen zouden niet toegevoegd worden in het minimale geval.*

Het tweede voorbeeld geeft aan dat 2-restrictie systemen zwakker zijn dan de *safe* of c-stratified verzamelingen [32]. Het 2-restrictie systeem uit dit voorbeeld is ook minimaal.

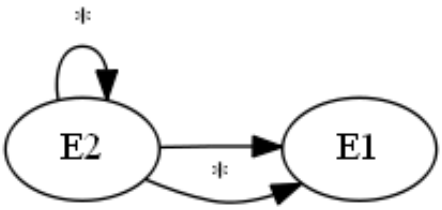
Voorbeeld 6.4.2. *We maken gebruik van een databaseschema zodat $S(x)$ een knoop is in de context van een graaf en $E(x, y)$ een boog. We kiezen dan $\Sigma = \{\sigma_1, \sigma_2\}$ met:*

- $\sigma_1 = S(x) \wedge E(x, y) \rightarrow E(y, x)$
- $\sigma_2 = S(x) \wedge E(x, y) \rightarrow \exists z\ E(y, z) \wedge E(z, x)$

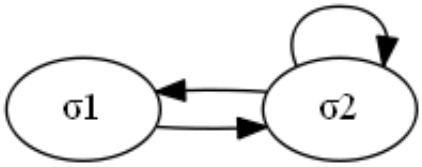
Deze dependencies zorgen ervoor dat alle knopen uit S een cyclus hebben van lengte 1 en van lengte 2. aff$(\Sigma) = \{E_1, E_2\}$. We kunnen dan controleren of dit voorbeeld safe is door te kijken naar prop(Σ). Deze is te zien in Figuur 6.6. We zien dat deze graaf duidelijk cyclussen bevat die gebruik maken van een speciale pijl ($\xrightarrow{x}$) en dus is Σ niet safe.

We kunnen controleren of Σ voldoet aan c-stratificatie. We zien dat $\sigma_1 \prec_c \sigma_2$, $\sigma_2 \prec_c \sigma_1$ en $\sigma_2 \prec_c \sigma_2$. We kunnen dan de c-chase graph $G_c(\Sigma)$ opstellen. Deze graaf is te zien in Figuur 6.7. We zien dat deze slechts uit een sterk verbonden component bestaat die niet zwak acyclisch is. Σ voldoet dus niet aan c-stratificatie.

We gaan nu een minimaal 2-restrictie systeem opstellen voor Σ. We vertrekken dus vanuit $((\Sigma, \emptyset), \emptyset)$. We kunnen dan controleren of $\sigma_2 \prec_\emptyset \sigma_1$. We kiezen een tableau $T_0 = \{S(c), S(d), E(c, d)\}$ en $\overline{a} = c, d$ en $\overline{b} = d, n_1$ met $n_1 \in \Delta_{null}$ en $c, d \in \Delta$. We kunnen dan I_1 berekenen als het resultaat van de toepassing van σ_2 op $\overline{a}$. We zien dan dat $T_1 = T_0 \cup \{E(d, n_1), E(n_1, c)\}$. Voor deze tableaus zien we dan dat $T_0 \models \sigma_1(\overline{b})$ en $T_1 \not\models \sigma_1(\overline{b})$. We moeten dan enkel de laatste conditie nog controleren. We kiezen hiervoor n_1 uit $\overline{b}$. We zien dan dat $null - pos(\{n_1\}, T_0) = \emptyset$ aangezien n_1 niet voorkomt in T_0. Aangezien $P = \emptyset$ zien we dus dat $null - pos(\{n_1\}, T_0) \subseteq P$ en dus is de laatste voorwaarde ook voldaan. We kunnen dus concluderen dat $\sigma_2 \prec_\emptyset \sigma_1$.



Figuur 6.6: *prop*(Σ) *uit Voorbeeld 6.4.2*



Figuur 6.7: $G_c(\Sigma)$ *uit Voorbeeld 6.4.2*

Dit zorgt dus voor een boog van σ_2 *naar* σ_1*. We krijgen dus als minimaal 2-restrictie systeem* $G'(\Sigma) = ((\Sigma), \{(\sigma_2, \sigma_1)\}), \emptyset)$*. Om te voldoen aan punt 1 en 2 uit de definitie van een 2-restrictie systeem moeten de affected positions uit* σ_1 *en* σ_2 *vervat zitten in* f*. We krijgen dus* $f = \{E_1, E_2\}$ *en als systeem* $((\Sigma), \{(\sigma_2, \sigma_1)\}), f)$

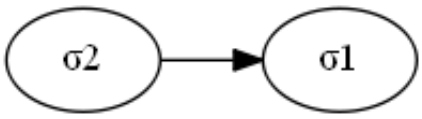
Verder zien we ook dat de positie S geen invloed heeft op de creatie van nieuwe gelabelde nullen. Deze zal dus niet toegevoegd moeten worden aan onze f*. We kunnen ook aantonen dat* $\sigma_1 \not\prec_f \sigma_1$ *en* $\sigma_1 \not\prec_f \sigma_2$ *en* $\sigma_2 \not\prec_f \sigma_2$*. Dit heeft als resultaat dat we niks meer moeten toevoegen aan ons 2-restrictie systeem. Het minimale 2-restrictie systeem voor* Σ *is dus* $((\sigma, \{(\sigma_2, \sigma_1)\}), f)$*.*

We maken nu gebruik van 2-restrictie systemen om tot de definitie van *safely restricted* te komen. Een verzameling dependencies Σ is *safely restricted* als en slechts als er een 2-restrictie systeem $(G'(\Sigma), f)$ is voor Σ zodat elke sterk verbonden component in $G'(\Sigma)$ *safe* is.

Een component van een graaf is een sterk verbonden component als elke knoop uit deze component in verbinding staat met elke andere knoop uit deze component.

Voorbeeld 6.4.3. *We kunnen de graaf opstellen voor ons restrictie systeem uit Voorbeeld 6.4.2. We zien dat deze slecht één boog heeft van* σ_2 *naar* σ_1*. Dit is te zien op Figuur 6.8. Aangezien deze component* safe *is, is* Σ *safely restricted. We zien dus dat Voorbeeld 6.4.2 niet voldoet aan c-startificatie of* safe *is maar wel safely restricted is.*

Belangrijk is dat we zien dat safely restricted inderdaad een uitbreiding is voor zowel safe als c-gestratifieerde verzamelingen van dependencies.



Figuur 6.8: *De graaf voor het 2-restrictie systeem uit Voorbeeld 6.4.2*

Het verband tussen deze terminatie condities is dus als volgt :

- Als Σ *safe* is dan is Σ safely restricted.
- Als Σ c-gestratifieerd is Σ safely restricted.
- Er is een Σ die safely restricted is maar niet *safe* noch c-gestratifieerd.

Verder moeten we ook nog aantonen dat als een verzameling dependencies Σ safely restricted is dat hieruit volgt dat $\sigma \in CT_{\forall\forall}$. We geven een schets van de manier waarop dit bewezen kan worden. Voor het bewijs steunen we op het feit dat er een uniek minimaal 2-restrictie systeem is voor Σ. Verder maken we ook gebruik van de eigenschap dat als alle strongly connected components van het minimale k-restrictie probleem in $CT_{\forall\forall}$ zitten dan $\Sigma \in CT_{\forall\forall}$. Analoog aan vorige bewijzen is de intuïtie weer dat een chase volgens deze componenten slechts eindig kan zijn aangezien alle sterk verbonden componenten reeds deel zijn van $CT_{\forall\forall}$.

6.5 Inductively restricted dependencies

In deze pragraaf veralgemenen we onze *safe* beperking door gebruikt te maken van een 2-restrictie systeem. Met behulp van deze uitgebreide *safety* introduceren we dan de terminatie conditie *inductive restriction*. Het idee achter deze techniek is weeral het opdelen van de verzameling van dependencies in kleinere deelverzamelingen. Hierna maken we dan gebruik van de safety eigenschap om te controleren of de deelverzamelingen veilig zijn.

We beginnen met een motiverend voorbeeld van een verzameling dependencies Σ die niet $safe$ noch safely restricted is.

Voorbeeld 6.5.1. *Dit voorbeeld is een verder uitbreiding van Voorbeeld 6.4.2. We weten dus al dat het voorbeeld niet* $safe$ *en niet c-gestratifieerd is. We voegen een extra dependency toe aan* Σ *zodat* $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ *met :*

- $\sigma_1 = S(x) \wedge E(x, y) \rightarrow E(y, x)$
- $\sigma_2 = S(x) \wedge E(x, y) \rightarrow \exists z\ E(y, z) \wedge E(z, x)$
- $\sigma_3 = \exists x, y\ S(x) \wedge E(x, y)$

We zoeken dan het minimale 2-restrictie systeem voor Σ*. Dit geeft ons* $G'(\Sigma') = (\Sigma, \{(\sigma_1, \sigma_2), (\sigma_2, \sigma_1), (\sigma_3, \sigma_1), (\sigma_3, \sigma_2)\}, f\}$ *met* $f = \{E.1, E.2, S.1\}$*. De reden dat dit systeem zo sterk verschilt van het systeem uit Voorbeeld 6.4.2 is dat de toevoeging van* σ_3 *ervoor zorgt dat* $S.1$ *ook een affected position is. Dit systeem bevat de sterk verbonden component* $\sigma_1, \sigma_2\}$*. We zien nu dat het voorbeeld ook niet safely restricted is aangezien* $\{\sigma_1, \sigma_2\}$ *niet safe is.*

We hebben dus een verzameling dependencies gevonden waarvoor we geen terminatie conditie kunnen vinden. Om dit op te lossen gaan we een nieuwe terminatie conditie introduceren.

Het eerste wat we introduceren om tot deze nieuwe terminatie conditie te komen is een algoritme voor het berekenen van het minimale 2-restrictie systeem voor de sterk verbonden componenten. Dit algoritme berekent dus de deelverzamelingen waarin we Σ gaan verdelen en berekent dan een restrictie systeem hiervoor. Dit $\mathbf{part}(\Sigma)$ algoritme werkt op een verzameling dependencies Σ. Het algoritme zal dan recursief alle deelverzamelingen berekenen.

Algorithm 1 Het $\mathbf{part}(\Sigma)$ algoritme

input: Σ, een verzameling TGD's en EGD's.
begin
bereken de sterk verbonden componenten $C_1, \ldots, C_n$ van
het minimale 2-restrictie systeem van Σ
$D = \emptyset$
if $(n == 1)$ **then**
if $(C_1 \neq \Sigma)$ **then**
return $\mathbf{part}(C_1)$
endif
return $\{\Sigma\}$
endif
for $i = 1$ to n **do**
$D = D \cup \mathbf{part}(C_i)$.
endfor
output D
end

Met behulp van dit algoritme kunnen we onze definitie formuleren. Een verzameling van dependencies Σ is inductively restricted als en slechts elke $\Sigma' \in \mathbf{part}(\Sigma)$ *safe* is.

Voorbeeld 6.5.2. *We kunnen nu de inductively restricted conditie toepassen op Voorbeeld 6.5.1. We doen dit door het part algoritme toe te passen op Σ.*

Dit begint met het berekenen van de sterk verbonden componenten van Σ. We weten dat deze component $\{\sigma_1, \sigma_2\}$ is (Voorbeeld 6.5.1). We zitten dan in het $n == 1$ geval en gaan dus part toepassen op $\{\sigma_1, \sigma_2\}$ en het resultaat hiervan als output geven.

Als we voor component $\{\sigma_1, \sigma_2\}$ het minimale 2-restrictie systeem berekenen dan zien we dat dit geen cyclus bevat(zie Voorbeeld 6.4.2). Dit toont aan dat $\mathbf{part}(\Sigma, 2) = \emptyset$. Er zijn immers geen sterk verbonden componenten meer in het minimale 2-restrictie systeem van $\{\sigma_1, \sigma_2\}$. Dit zorgt dus voor een $n = 0$ situatie zodat ons algoritme gewoon $D = \emptyset$ als output geeft. We kunnen dus concluderen dat Σ inductively restricted is.

Intuïtief is dit een logisch resultaat aangezien de chase voor deze Σ inderdaad zal eindigen. Enkel σ_3 voegt nieuwe knopen toe en deze dependency zal slechts een eindig aantal keer toegepast kunnen worden. σ_1 en σ_2 gaan dan alle 1- en 2-cycli genereren. Dit zijn er echter een eindig aantal als er geen nieuwe knopen meer geïntroduceerd worden.

Ons voorbeeld toont aan dat inductively restricted nog een verdere uitbreiding is van safely restricted.

Als σ inductively restricted is dan $\Sigma \in CT_{\forall\forall}$. Het bewijs hiervoor is analoog aan dit voor safely restricted. We maken gebruik van de gegeven terminatie conditie voor de sterke componenten om tot een eindige chase te komen voor Σ[32].

Hoofdstuk 7

Data-afhankelijke terminatie van de chase

De volgende stap in ons onderzoek van de terminatie van de chase is de studie van $CT_{T,\forall}$ en $CT_{T,\exists}$. Een belangrijk verschil met de vorige hoofdstukken is dat de terminatie in dit geval afhankelijk is van de data die gebruikt wordt.

7.1 Irrelevantie

We weten dat de chase altijd zal eindigen voor een tableau T als de verzameling van de dependencies $\Sigma \in CT_{\forall\forall}$

We kunnen verder bouwen op deze observatie door te berekenen welke dependencies uit Σ niet van toepassing zijn voor de chase over T. Hiervoor introduceren we het begrip *irrelevant*. We zeggen dat een dependency $\sigma \in \Sigma$ irrelevant is voor een tableau T, genoteerd als (T, Σ)-irrelevant, als en slechts als er geen chase sequence is zodat σ kan vuren. Er is dus geen enkele stap in de chase sequence die σ toepast voor een bepaalde tupel $\overline{a}$ uit T.

We willen gebruik maken van deze irrelevante dependencies bij het bepalen van de terminatie van de chase. De vraag is dus of het makkelijk is om de verzameling van alle (T, Σ)-relevante dependencies te berekenen. We zien echter dat het voor een gegeven set van dependencies Σ, een dependency $\sigma \in \Sigma$ en een tableau T onbeslisbaar is of $\sigma(T, \Sigma)$-irrelevant is. Het bewijs van deze stelling steunt op de reductie van een gekend onbeslisbaar probleem naar het irrelevantie probleem. Het gekende onbeslisbare probleem dat we hiervoor gebruiken is het probleem of een Turing machine M een zekere transitie t bereikt (met de lege string als input). We gebruiken hiervoor het verband tussen de uitvoering van een Turing machine en de uitvoering van de chase analoog aan het bewijs uit Paragraaf **??**[32].

We zien dus dat het niet mogelijk is om de minimale verzameling dependencies te bepalen die kunnen vuren tijdens de chase van T. We kunnen echter wel voorwaarden bepalen die de (T, σ)-irrelevantie van een dependency afdwingen. Hiervoor introduceren

we de speciale dependency σ_T.

$$\sigma_T = \exists \overline{x} \bigwedge_{R(\overline{x}') \in T} R(\overline{x}')$$

Hierbij is $\overline{x} = \cup_{R(\overline{x}') \in T} \overline{x}'$. Voor een tableau T is dit een conjunctie van de elementen uit T met een existentiële kwantor voor de variabelen uit T.

Deze σ_T kunnen we gebruiken bij het opstellen van een c-chase graph $G_c(\Sigma \cup \{\sigma_T\}$ We maken dan gebruik van deze c-chase graph om te bepalen welke dependencies bereikbaar zijn vanuit σ_T, de verzameling van deze dependencies noemen we $\mathrm{WCC}_\Sigma(\sigma_T)$. We kunnen dan aantonen dat als $\sigma \notin \mathrm{WCC}_\Sigma(\sigma_I)$ dan is σ (I, Σ)-irrelevant [32].

Intuitief kunnen we inzien dat dit het geval is. Σ_T zal immers de elementen die overeenkomen met de informatie van tableau T introduceren in de c-chase graph. De bogen in deze graaf geven aan welke dependencies mekaar kunnen afvuren. De dependencies die bereikbaar zijn vanuit σ_T zijn dus de dependencies die afgevuurd kunnen worden als resultaat van de toepassing van σ_T. Aangezien Σ_T overeen komt met de data uit tableau T zijn dit dus de dependencies die afgevuurd kunnen worden op de data van T. De dependencies die niet voorkomen in $\mathrm{WCC}_\Sigma(\sigma_T)$ worden dus niet afgevuurd op de data uit T. Ze zijn dus niet relevant.

Voorbeeld 7.1.1. *We bekijken een verzameling dependencies $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ met:*

- $\sigma_1 = E(x_1, x_2, d) \rightarrow S(x_1) \wedge S(x_2)$
- $\sigma_2 = R(x_1, x_2, d) \rightarrow R(x_2, x_1, d)$
- $\sigma_3 = E(x_1, x_2, d) \rightarrow \exists y, d'\ E(x_2, y, d')$

We zien dat σ_3 ervoor zorgt dat $\Sigma \notin CT_{\forall\forall}$ en $\Sigma \notin CT_{\forall\exists}$.

We willen nu echter de terminatie bekijken voor een zeker tableau T met

$$T = \{R(c_1, x_1, d_1), E(x_1, x_2, d_2), E(x_2, x_1, d_2), R(x_1, c_1, d_1)\}$$

We kunnen dan σ_T opstellen voor deze T:

$$\sigma_T = \exists c_1, d_1, d_2, x_1, x_2\ R(c_1, x_1, d_1) \wedge E(x_1, x_2, d_2) \wedge E(x_2, x_1, d_2) \wedge R(x_1, c_1, d_1)$$

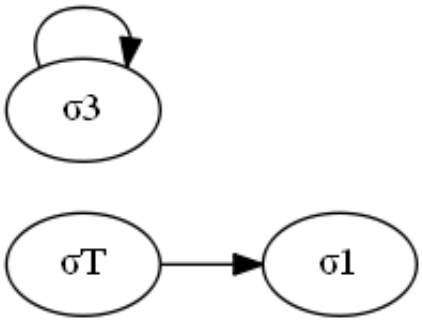
We stellen de c-chase graph $G_c(\Sigma \cup \{\sigma_T\})$. De dependencies die in deze graaf bereikbaar zijn vanuit σ_T zijn de relevante dependencies. $G_c(\Sigma \cup \{\sigma_T\})$ is zichtbaar in Figuur 7.1. We zien dus dat σ_T enkel verbonden is met σ_1 dus:

$$WCC_\Sigma(\sigma_T) = \{\sigma_1\}$$

Het resultaat hiervan is dat zowel σ_2 als σ_3 (T, Σ)-irrelevant zijn.

7.2 $CT_{T,\forall}$

We kunnen dan van deze irrelevantie gebruik maken om de terminatie van de chase te bepalen. We weten immers dat enkel de dependencies uit Σ' afgevuurd zullen worden. Als deze dus deel zijn van $CT_{\forall\forall}$ dan weten we dat de terminatie verzekerd is.



Figuur 7.1: *De c-chase graph voor Voorbeeld 7.1.1*

We hergebruiken dus de terminatiecondities van $CT_{\forall\forall}$ maar passen deze enkel toe op de dependencies die relevant zijn voor het gegeven tableau.

Voorbeeld 7.2.1. *We kunnen deze terminatieconditie toepassen voor de informatie uit Voorbeeld 7.1.1. We moeten dus enkel de terminatie controleren voor* $\Sigma \setminus \{\sigma_2, \sigma_3\} = \{\sigma_1\}$. *Als we enkel* σ_1 *bekijken dan zien we dat* $\{\sigma_1\} \in CT_{\forall\forall}$. *We zien dus dat* $\Sigma \in CT_{T,\forall}$.

7.3 $CT_{T,\exists}$

We kunnen nu deze redenering ook toepassen voor $CT_{T,\exists}$. We komen dus tot eenzelfde resultaat als voor $CT_{T,\forall}$. Als $\text{WCC}_\Sigma(\sigma_T) \in CT_{\forall\exists}$ dan $\Sigma \in CT_{T,\exists}$.

We weten dat als $\sigma \notin \text{WCC}_\Sigma(\sigma_T)$ dan is σ (T, Σ)-irrelevant. We controleren nu echter alleen of de dependencies die wel in $\text{WCC}_\Sigma(\sigma_T)$ zitten vervat zijn in $CT_{\forall\exists}$. Dit zijn dus enkel dependencies die wel relevant zijn. We weten dus voor de dependencies die afgevuurd zullen worden dat ze deel zijn van $CT_{\forall\exists}$ en dus dat er een eindige chase sequence zal zijn als we enkel gebruik maken van deze dependencies. De niet relevante dependencies worden niet gebruikt en hebben dus ook geen invloed op de terminatie van de chase.

7.4 Monitoring

Een andere methode om te testen of de chase stopt is door het algoritme uit te voeren en de resultaten te bekijken. Zo kunnen we de chase stoppen als er tupels gegenereerd worden die tot een oneindige chase kunnen leiden. Hiervoor introduceren we een *monitor graph*. Een monitor graph is een tupel (W, E) waarbij $W \subseteq V \times 2^{pos(\Sigma)}$ en $E \subseteq W \times \Sigma \times 2^{pos(\Sigma)} \times W$.
Een knop is dus een tupel(n,P) met n een waarde uit V en P de posities waarop deze n gegenereerd is. Een edge $(n_1, P_1, \sigma_i, , P, n_2, P_2,)$ is een boog tussen (n_1, P_1) en (n_2, P_2). De boog is gelabeld met de dependency σ_i die verantwoordelijk is voor de creatie van n_2 en met de posities P uit de body van σ_i waarop n_1 voorkwam bij de creatie van n_2.

Voor een chase sequence $S = T_0, \ldots, T_r$ kunnen we de monitor graph inductief definiëren op de volgende manier:

- $G_0 = (\emptyset, \emptyset)$ is de graaf voor het lege chase segment.
- Als $i < r$ en σ_i is een EGD dan is $G_{i+1} = G_i$
- Als $i < r$ en σ_i is een TGD dan wordt G_{i+1} verkregen vanuit G_i op de volgende manier:
  - Als de chase stap van T_i naar T_{i+1} geen nieuwe variabelen introduceert dan is $G_{i+1} = G_i$
  - Anders, als de chase stap van T_i naar T_{i+1} door de toepassing van σ_i op $\overline{a}_i$ wel nieuwe variabelen introduceert. Dan is V_{i+1} de unie van V_i met alle paren (n, P) waarbij n een nieuw geïntroduceerde variabele is en P de verzameling van posities waarop n voorkomt.
    $E_{i+1} = E_i \cup \{(n_1, P_1, n_2, P_2, \sigma_i, P) \mid (n_1, P_1) \in V_i, (n_2, P_2) \in V_{i+1} \setminus V_i$ en P gelijk aan de verzameling posities uit de body van $\sigma_i(\overline{a}_i)$ waar n_1 voorkomt $\}$.

Aan de hand van deze monitor graph kunnen we nu een heuristiek invoeren voor het niet eindigen van de chase. Deze heuristiek is *k-cycliciteit*.
Voor een monitor graph $G = (V, E)$ en een $k \in \mathbb{N}$ noemen we G k-cyclisch als en slechts als er paarsgewijs verschillende bogen $e_1, \ldots, e_k \in E$ bestaan zodat:

- Er is een pad in G dat $e_1, \ldots, e_k$ bevat en e_i komt ergens voor in het pad voordat e_{i+1} voorkomt in dit pad.
- Voor alle $i \in [1, k-1]$ geldt dat $p_{2,3,4,6}(e_i) = p_{2,3,4,6}(e_{i+2})$.

Hierbij betekent de notatie $p_{2,3,4,6}$ dat ze enkel op deze plaatsen moeten overeenkomen.

We noemen een chase sequence dan k-cyclisch als de monitor graph van deze chase sequence k-cyclisch is. We weten dat voor alle $k \in \mathbb{N}$ en voor elke oneindige chase sequence S van de chase van I over Σ dat er een eindige prefix is van S die k-cyclisch is [32]. Dit is het geval aangezien een oneindige chase sequence enkel kan voorkomen als er nieuwe variabelen geïntroduceerd worden. Er moet dus ergens in de chase sequence een herhaling zijn van dependencies die nieuwe variabelen introduceren . Deze herhaling zal ervoor zorgen dat de monitoring graph k-cyclisch zal zijn.

Voorbeeld 7.4.1. *We beschouwen een verzameling dependencies* $\Sigma = \{\sigma_1, \sigma_2\}$ *met:*

- $\sigma_1 = R(x_1, x_2) \rightarrow \exists y\ S(x_2, y)$
- $\sigma_2 = S(x_1, x_2) \rightarrow \exists y\ R(x_2, y)$

Het is duidelijk in te zien dat de chase over Σ *oneindig zal zijn. We tonen nu hoe me monitoring kunnen gebruiken als heuristiek om de uitvoering van de chase af te breken. We beginnen de chase vanuit tableau* $T = \{R(a, b)\}$*. Bij het begin van de chase is de monitor graph* $G_0 = (\emptyset, \emptyset)$*. Om de notatie van het voorbeeld te verkorten noteren we geen* $\{\}$ *als de verzameling posities slechts uit een enkele positie bestaat.*

We beginnen dan met de toepassing van σ_1 *volgens* $h = \{(x_1, a), (x_2, b)\}$*. We krijgen dan* $T_1 = T_0 \cup \{S(b, n_1)\}$*. We gaan de knopen van onze monitor graph dan uitbreiden met* $(n_1, S.2)$*. We krijgen dus* $W_1 = \{(n_1, S.2)\}$*. We krijgen nog geen nieuwe boog aangezien er geen knoop is in* W_0 *dus* $G_1 = W_1, \emptyset)$*.*

Voor de volgende chase step passen we σ_2 toe volgens $h = \{(x_1, b), (x_2, n_1)\}$. We krijgen dan $T_2 = T_1 \cup \{R(n_1, n_2)\}$. We gaan de knopen van onze monitor graph dan uitbreiden en krijgen $W_2 = W_1 \cup \{(n_2, R.2)\}$. We controleren dan of we een boog kunnen vinden tussen een knoop uit W_2 en een knoop uit W_1. We krijgen inderdaad een boog van $(n_1, S.2)$ naar $(n_2, R.2)$ deze boog heeft als dependency σ_2 en als verzameling posities $P = \{S.2\}$. Dus $E_2 = E_1 \cup \{(n_1, S.2, \sigma_2, S.2, n_2, R.2)\}$.

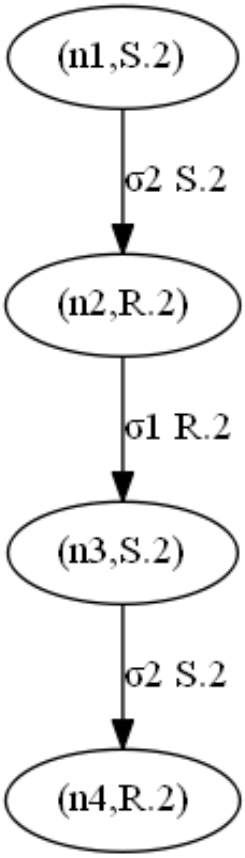
voor de volgende chase step passen we opnieuw σ_1 toe, deze keer volgens $h = \{(x_1, n_1), (x_2, n_2)\}$. We krijgen dan $T_3 = T_2 \cup \{S(n_2, n_3)\}$. We kunnen ook weer onze monitor graph aanpassen aan deze nieuwe chase step. We zien dat we een nieuwe knoop krijgen zodat $W_3 = W_2 \cup \{(n_3, S.2)\}$. We voegen ook de nieuwe knoop toe: $E_3 = E_2 \cup \{(n_2, R.2, \sigma_1, R.2, n_3, S.2)\}$.

We passen nogmaals σ_2 toe, nu volgens $h = \{(x_1, n_2), (x_2, n_3)\}$. Dit geeft ons $T_4 = T_3 \cup \{R(n_3, n_4)\}$. We voegen dan ook een nieuwe knoop toe aan onze monitor graph zodat $W_4 = W_3 \cup \{(n_4, R.2)\}$. We krijgen dan nog een extra boog zodat: $E_4 = E_3 \cup \{(n_3, S.2, \sigma_2, S.2, n_4, R.2)\}$.

De monitoring graph na deze iteraties is zichtbaar in Figuur 7.2. Om het voorbeeld beknopt te houden gaan we op zoek naar 1-cycliciteit.
We vinden dan boog $(n_1, \{S.2\}, \sigma_2, \{S.2\}, n_2, \{R.2\})$ en boog $(n_3, S.2, \sigma_2, S.2, n_4, R.2)$ zodat voldaan is aan de voorwaarden voor 1-cycliciteit. De elementen op posities 2,3,4 en 6 komen immers overeen en er is een pad in de graaf waarin ze beide in voorkomen. Ook zijn ze niet gelijk aangezien de elementen op positie 1 en 5 niet overeenkomen. We stoppen dan met de uitvoering van de chase omdat onze heuristiek aangeeft dat deze niet zal stoppen.

Uiteraard zouden we in de realiteit een grotere k kiezen.

Als we dus tijdens het monitoren van de uitvoering van de chase vinden dat de monitor graph k-cyclisch is dan is dit een goede indicatie dat de chase oneindig zal zijn. Het efficiënter om gebruik te maken van k-cycliciteit dan van eenvoudige heuristiek zoals het aantal stappen in de chase sequences of het aantal toepassingen van een bepaalde dependency. Door rekening te houden met cycli maken we een onderscheid tussen een chase die gewoon veel stappen bevat en een chase waarin de dependencies leiden tot de voortdurende creatie van nieuwe variabelen.



Figuur 7.2: *De monitor graph voor Voorbeeld7.4.1. De toevoeging van de bogen en de knopen in deze graaf verliep van boven naar onder zoals in het voorbeeld.*

Hoofdstuk 8

Toepassingen van de Chase

8.1 Oplossen van het implicatie probleem

De chase werd geïntroduceerd als oplossing voor het implicatie probleem[3]. Dit is dus ook een eerste toepassing van het chase algoritme.

Voor een gegeven verzameling dependencies Σ en een dependency σ die niet in Σ zit, kunnen we bepalen of $\Sigma \models \sigma$. Dit wil zeggen dat σ een logisch gevolg is van de dependencies in Σ. Hiermee kunnen we bijvoorbeeld voor een nieuwe dependency bepalen of een bepaald tableau T zal voldoen aan deze dependency. Stel immers dat ons tableau T al voldoet aan Σ dus $T \models \Sigma$. Als we dan aan kunnen tonen dat $\Sigma \models \sigma$ dan weten we ook dat $T \models \sigma$.

Als we een $chase_\Sigma(T)$ vinden als resultaat van een chase van het tableau T dat overeenkomt met σ over Σ dan kunnen we aan de hand van dit resultaat controleren of $\Sigma \models \sigma$. We doen dit door te controleren of de head van σ vervat zit in $chase_\Sigma(T)$. Dit komt overeen met het controleren of $chase_\Sigma(T) \models \sigma$. Als dit het geval is dan mogen we concluderen dat $\Sigma \models \sigma$ en als dit niet het geval is dan geldt deze implicatie niet.

Formeel kunnen we zeggen dat $\Sigma \models \sigma \iff chase_\Sigma(T_\sigma) \models \sigma$. Waarbij T_σ het tableau is dat overeenkomt met de body van σ.

Een eerste opmerking over het implicatie probleem is dat dit onbeslisbaar is als er gebruik gemaakt wordt van keys en foreign keys[13, 14]. Verder gaan we enkel de chase kunnen gebruiken om dit probleem op te lossen als $chase_\Sigma(T_\sigma)$ effectief bestaat. We zien dus dat we terminatie condities nodig hebben.

Een voorbeeld van het gebruik van de chase om het implicatie probleem op te lossen werd reeds gegeven in Voorbeeld 3.2.1.

In de volgende paragraaf geven we nog een ander voorbeeld van het gebruik van de chase om tot een oplossing voor het implicatieprobleem te komen. Hier wordt er geen expliciet gebruik gemaakt van de chase maar het gaat om een methode die aanleunt bij de chase.

8.2 Implicatieprobleem voor inclusion dependencies

We bewijzen dat voor de klasse van inclusion dependencies de eindige en oneindige implicatie beide beslisbaar zijn en dat ze samenvallen. We merken op dat dit bewijs gebruik maakt van de chase en dus een toepassing van de chase is.

8.2.1 Afleidingsregels

Om dit probleem te bekijken beschouwen we de afleidingsregels voor inclusie dependencies:

I1 $R[X] \subseteq R[X]$ (reflexiviteit)

I2 als $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ en $i_1, \ldots, i_m$ is een permutatie van $1, \ldots, m$ dan $R[A_{i_1}, \ldots, A_{i_m}] \subseteq S[B_{i_1}, \ldots, B_{i_m}]$ (permutatie)

I3 als $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ en $i_1, \ldots, i_k$ is een deelrij van $1, \ldots, m$ dan $R[A_{i_1}, \ldots, A_{i_k}] \subseteq S[B_{i_1}, \ldots, B_{i_k}]$ (projectie)

I4 als $R[X] \subseteq S[Y]$ en $S[Y] \subseteq T[Z]$ dan $R[X] \subseteq T[Z]$ (transitiviteit)

We bewijzen dat deze set van afleidingsregels $\{I1, I2, I3, I4\}$ sound and complete is voor de logische implicatie van ind's. Hierbij staat het begrip *sound* voor het juist zijn van alle logische afleidingen uit de regels en *complete* voor het volledig zijn van de resultaten van de logische afleiding volgens de regels, dit wil zeggen dat alle mogelijke gevolgen gevonden worden. Deze regels zijn duidelijk sound, de vraag is dus of ze complete zijn.

Merk op dat I2 en I3 (permutatie en projectie) samengevoegd zouden kunnen worden tot één regel van de volgende vorm:

als $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ dan $R[A_{i_1}, \ldots, A_{i_k}] \subseteq S[B_{i_1}, \ldots B_{i_k}]$
voor elke sequentie $i_1, \ldots, i_k$ van distincte integers in$\{1, \ldots, m\}$

8.2.2 Compleetheid

We moeten bewijzen dat $\Sigma \models_{fin} \sigma \Rightarrow \Sigma \vdash \sigma$

veronderstel:

- een databaseschema $\mathsf{S} = \{(R_1, ar_1), \ldots, (R_n, ar_n)\}$.
- Σ een verzameling ind's over S
- Een ind $\sigma = R_a[A_1, \ldots, A_m] \subseteq R_b[B_1, \ldots, B_m]$
- zodat $\Sigma \models \sigma$.

We construeren een instance I van S om aan te tonen dat $\Sigma \vdash \sigma$

Neem een s' een tupel over R_a zodat

$$\begin{cases} s'(a_i) = i \text{ voor } i \in [1, m] \\ s'(b) = 0 \text{ in alle andere gevallen} \end{cases}$$

Deze s' zal er dan bijvoorbeeld als volgt uitzien $(0, 0, 1, 2, 0, 3, 5, 0, 4)$. Waarbij de elementen die overeenkomen met de $A_1, \ldots, A_m$ uit σ hun index i als waarde krijgen. De overige elementen krijgen 0 als waarde.

We maken dan een instance $I(R_a) = \{s'\}$ en $I(R_j) = \emptyset$ voor $j \neq a$ We passen nu de volgende bewerking toe op I totdat deze niet meer kan worden toegepast:

(*) als er een regel $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k] \in \Sigma$ bestaat en $t \in I(R_i)$ dan voeg u toe aan $I(R_j)$ waarbij $u(D_t) = t(C_l)$ voor $l \in [1, k]$ en $u(D) = 0$ for $D \notin [D_1, \ldots, D_k]$

Hiermee bouwen we vanuit de starttupel s' in R_a tupels op die voldoen aan de ind's.

Merk op dat deze regel het verband bevat met de chase. We kiezen immers telkens een dependency die we toepassen op de huidige instance net zoals in de chase gebeurt. We kunnen dit bewijs dus zien als een toepassing van de chase.

Het uitvoeren van deze regel zal zeker stoppen aangezien alle tupels geconstrueerd worden vanuit een verzameling met maximum $m + 1$ waarden. We noemen het unieke resultaat van deze bewerking de instance J.

We zien dan dat $J \models \Sigma$ aangezien J opgebouwd is volgens de ind's uit Σ en omdat $\Sigma \models \sigma$ weten we dat $J \models \sigma$.

Om het bewijs te vervolledigen tonen we het volgende aan:

(**) als er voor een bepaald R_j in S, $u \in J(R_j)$, een getal q en attributen $E_1, \ldots, E_q$ van R_j met $u(E_p) > 0$ voor $p \in [1, q]$ dan

$$\Sigma \vdash R_a[A_{u(e_1)}, \ldots A_{u(e_q)}] \subseteq Rj[E_1, \ldots, E_q]$$

We bewijzen dit door middel van inductie op de tupels van J in de volgorde dat deze geconstrueerd zijn.

Basisgeval

basisgeval: $u = s'$ in $J(R_a)$. We zien dat dit bewijsbaar is via I1.

We bekijken dit basisgeval in meer detail: als we s' kiezen als u dan weten we dat voor $E_1, \ldots, E_q]$ geldt dat $u(E_p) > 0$ voor $p \in [1, q]$ en aangezien $u = s'$ weten we dus dat $s'(E_p) > 0$ voor $p \in [1, q]$.

Dankzij de constructie van s' weten we dat $s'(E_p)$ enkel groter dan 0 kan zijn als deze E_p overeenkomt met een A_i uit σ. De andere elementen kregen immers 0 als waarde.

We zien dus dat $E_p = A_i$ voor $p \in [1, q]$ en $i \in [1, m]$. We kunnen dan afleiden dat $s'(E_p) = s((A_i)$ en we weten uit de constructie van s' dat $s'(A_i) = i$ dus $s'(E_p) = i$.

Dit leidt tot het volgende resultaat:

$$E_p = A_i = A_{s'(E_p)}$$

We moeten nu aantonen dat $\Sigma \vdash R_a[A_{u(E_1)}, \ldots, A_{u(E_q)}] \subseteq R_j[E_1, \ldots, E_Q]$

Door het toepassen van $E_p = A_i = A_{s'(E_p)}$ komen we tot

$$\Sigma \vdash R_a[E_1, \ldots, E_q] \subseteq R_j[E_1, \ldots, E_q]$$

We weten ook dat R_j in dit geval gelijk is aan R_a. We weten immers dat $u \in J(R_j)$ en $u = s'$ en uit de constructie van J weten we dat $s' \in J(R_a)$.

Als we dit toepassen komen we tot het volgende resultaat:

$$\Sigma \vdash R_a[E_1, \ldots, E_q] \subseteq R_a[E_1, \ldots, E_q]$$

Het is duidelijk zichtbaar dat dit gaat over reflexiviteit. Dankzij afleidingsregel I1 weten we dat dit inderdaad bewijsbaar is en dus dat regel (**) geldt voor het basisgeval $u = s'$.

Inductiehypothese

Aangezien het basisgeval voldoet gaan we over tot de inductiestap, we veronderstellen dan dat

- I' is de instance verkregen na k keer toepassen van de regel (*) voor een $k \geq 0$.
- De regel (**) geldt voor alle tupels in I'.
- u is toegevoegd aan R_j door de volgende uitvoer van de regel (*), volgens de ind $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k] \in \Sigma$ en tupel $t \in I'(R_i)$.

Neem $\{E_1, \ldots, E_q\}$ een verzameling attributen uit R_j met $u(E_p) > 0$ voor $p \in [1, q]$

We moeten dan aantonen dat de inductiehypotese geldt dus moeten we kunnen aantonen dat regel (**) waar is voor u en $\{E_1, \ldots, E_q\}$. We moeten dus bewijzen dat:

$$\Sigma \vdash R_a[A_{u(E_1)}, \ldots, A_{u(E_q)}] \subseteq R_j[E_1, \ldots, E_q]$$

We weten dat $\{E_1, \ldots, E_q\}$ enkel attributen bevat zodat voor elke $p \in [1, q]$ geldt dat $E_p > 0$. Door de constructie volgens regel (*) weten we dat het niet nul zijn van deze elementen erop wijst dat ze geconstrueerd zijn vanuit een ind, de overige elementen werden immers altijd op nul gezet. Voor u weten we vanuit het gegeven dat u is toegevoegd volgens de ind $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k]$. De mogelijke niet nul attributen van u zijn dus $\{D_1, \ldots, D_k\}$. Vermits onze $\{E_1, \ldots, E_q\}$ vrij gekozen zijn met als enige voorwaarde dat $u(E_p) > 0$ voor $p \in [1q]$ zal er dus gelden dat $\{E_1, \ldots, E_q\} \subseteq \{D_1, \ldots, D_k\}$.

We zien dat uit de constructie van u in (*) volgt dus dat $\{E_1, \ldots, E_q\} \subseteq \{D_1, \ldots, D_k\}$.

Dit zorgt ervoor dat we weten dat er een mapping r mogelijk is zodat $D_{r(p)} = E_p$ voor $p \in [1, q]$. Merk op dat we deze mapping ook kunnen toepassen op attributen uit het linkerlid van de ind $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k]$. We zien dat C_i vervat zit in D_i voor $i \in [1, k]$ (volgende de definitie van ind) en dus ook dat $C_r(p)$ vervat zit in $D_r(p)$ voor $pin[1, q]$. We passen immers dezelfde mapping toe en krijgen dus dezelfde overeenkomende attributen als resultaat.

We weten dat $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k] \in \Sigma$ en dus dat:

$$\Sigma \vdash R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k]$$

We kunnen nu onze mapping r toepassen op de beide leden van de ind. We krijgen dan dat:

$$\Sigma \vdash R_i[C_{r(1)}, \ldots, C_{r(q)}] \subseteq R_j[D_{r(1)}, \ldots, D_{r(q)}]$$

We weten dat deze bewijsbaarheid mogelijk is door gebruik te maken van I2 en I3. Het resultaat van de toepassing van de mapping r is immers niets meer dan een projectie en/of een permutatie van de attributen. Als we nu gebruik maken van $D_{r(p)} = E_p$ voor $p \in [1, q]$ dan zien we dat:

$$\Sigma \vdash R_i[C_{r(1)}, \ldots, C_{r(q)}] \subseteq R_j[E_1, \ldots, E_q]$$

Verder weten we dat de inductiehypothese geldt voor tupels die eerder toegevoegd zijn dan u. We weten dus dat de inductiehypothese geldt voor tupel t. We kiezen dan een verzameling attributen $\{C_{r(1)}, \ldots, C_{r(q)}\}$ met $t(C_{r(p))} > 0$ voor $p \in [1, q]$. We weten deze inderdaad groter zijn dan nul aangezien ze gelijk zijn aan $u(D_{r(p))} = u(E_p)$ waarvan we weten dat ze groter zijn dan nul. We hebben immers $\{E_1, \ldots, E_q\}$ zodanig gekozen dat dit het geval is. We kunnen dus regel(**) toepassen voor $\{C_{r(1)}, \ldots, C_{r(q)}$ en t. We krijgen dan dat:

$$\Sigma \vdash R_a[A_{t(C_{r(1)})}, \ldots, A_{t(C_{r(q)})} \subseteq R_i[C_{r(1)}, \ldots, C_{r(q)}]$$

Merk op dat we t hier kunnen gebruiken als een mapping van de attributen van R_a naar deze van R_j. Dit is mogelijk dankzij de constructie uit (*). De waarde van $t(C_{r(p)})$ is immers een gevolg van het toepassen van regel (*) op startupel s' waarbij $s'(A_i) = i$. Deze waarde is dus nog steeds de index i van het originele attribuut A_i van R_a.

We weten dan dus dat:

- $\Sigma \vdash R_i[C_{r(1)}, \ldots, C_{r(q)}] \subseteq R_j[E_1, \ldots, E_q]$
- $\Sigma \vdash R_a[A_{t(C_{r(1)})}, \ldots, A_{t(C_{r(q)})} \subseteq R_i[C_{r(1)}, \ldots, C_{r(q)}]$

Volgens I4 (transitiviteit) geldt dus dat:

$$\Sigma \vdash R_a[A_{t(c_{r(1)})}, \ldots, A_{t(c_{r(q)})} \subseteq R_j[E_1, \ldots, E_q]$$

merk op dat $\forall p : t(c_{r(p)}) = u(d_{r(p)}) = u(e_p)$. Dit is het geval omdat we weten dat $R_i[C_1, \ldots, C_k] \subseteq R_j[D_1, \ldots, D_k]$. Dit heeft dus als resultaat dat voor de tupel t uit R_i

zijn elementen in de tupel u van R_j zullen zitten op de plaats die bepaald is door de ind.

Merk nu op dat $t(C_{r(p)}) = u(D_{r(p)})$ voor $p \in [1, q]$. Dit is waar omdat de ind $R_i[C_{r(1)}, \ldots, C_{r(q)}] \subseteq R_j[E_1, \ldots, E_q]$ geldt. We zien ook dat $u(D_{r(p)}) = u(E_p)$ aangezien r zodanig opgesteld is dat $D_{r(p)} = E_p$.

We weten dus dat $\forall p \in [1, q] : t(C_{r(p)}) = u(D_{r(p)}) = u(E_p)$

als we deze gelijkheid toepassen krijgen we het volgende resultaat:

$$\Sigma \vdash R_a[A_{u(E_1)}, \ldots, A_{u(E_q)}] \subseteq R_j[E_1, \ldots, E_q]$$

Hiermee hebben we dus bewezen dat de inductie geldt en dus dat regel (**) waar is voor alle elementen van J.

We weten dan dus dat $\Sigma \models \sigma$. Daarmee hebben we dus de completeness van de axiomaset bewezen.

8.2.3 Eindige en oneindige implicatie van inclusie dependencies

We weten nu dus dat de volgende eigenschappen gelden voor een willekeurige verzameling ind's Σ en een willekeurige ind $\sigma \notin \Sigma$:

- soundness: $\Sigma \vdash \sigma \Rightarrow \Sigma \models_{unr} \sigma$
- completeness: $\Sigma \models_{fin} \sigma \Rightarrow \Sigma \vdash \sigma$

Uit de defenitie van oneindige implicatie weten we dat :

$$\Sigma \models_{unr} \sigma \Rightarrow \Sigma \models_{fin} \sigma$$

Via de toepassing van sound en complete krijgen we dus:

$$\Sigma \models_{fin} \sigma \Rightarrow \Sigma \vdash \sigma \Rightarrow \Sigma \models_{unr} \sigma$$

Dit geeft dus als resultaat $\Sigma \models_{unr} \sigma \iff \Sigma \models_{fin} \sigma$

We zien dus dat de eindige en oneindige logische implicatie samenvallen. Het gevolg hiervan is dat het implicatieprobleem beslisbaar is.

We kunnen de afleidingsregels gebruiken om dit te beslissen. We zien immers dat in deze afleidingsregels de attribuutsequenties nooit langer worden. We kunnen dus een exhaustieve toepassing van de afleidingsregels doen om het implicatieprobleem op te lossen.

8.3 Repareren van de schendingen in een tableau

We kunnen de chase gebruiken om ervoor te zorgen dat een tableau T voldoet aan een verzameling dependencies Σ. Het toepassen van de chase op T volgens Σ zal ons $chase_\Sigma(T)$ als resultaat geven. Dit resultaat is ook een tableau.

We weten dat dit resultaat zal voldoen aan alle dependencies uit Σ. Stel immers dat er een dependency $\sigma \in \Sigma$ is zodat $chase_\Sigma(T) \not\models \sigma$. Dan weten we dat het mogelijk is om σ toe te passen op $chase_\Sigma(T)$. Dit geeft echter een tegenstelling aangezien $chase_\Sigma(T)$ het eindresultaat is van de chase. Het zou dus niet meer mogelijk moeten zijn om hierop een dependency uit Σ toe te passen.

Een voorbeeld van de manier waarop de chase een tableau kan repareren is te zien in Voorbeeld 8.3.1.

Voorbeeld 8.3.1. *We geven een voorbeeld van de chase op een tableau T volgens een verzameling dependencies Σ met:*

- $T = \{verbinding(Brussel, Londen), verbinding(Brussel, Parijs), verbinding(Parijs, Brussel)\}$
- $\Sigma = \{\sigma\}$
- $\sigma = verbinding(x, y) \rightarrow verbinding(y, x)$

We kunnen ons voorstellen dat dit gaat over treinverbindingen. Hierbij dwingt de constraint σ uit Σ af dat er een wederzijdse verbinding is zodat de trein terug kan rijden. We kunnen dan de chase gebruiken om de schendingen van Σ te repareren.

We beginnen met het zoeken van een homomorfisme h van de body van σ naar T. We vinden een mogelijk homomorfisme $h = \{(x, Brussel), (y, Parijs)\}$. We testen dan of de head van σ ook via dit homomorfisme gemapped kan worden naar T. We passen dus h toe op $verbinding(y, x)$ en vinden $verbinding(Parijs, Brussel)$. Aangezien de head ook voldoet kunnen we de TGD σ niet toepassen op T

We beschouwen dus een ander homomorfisme $h_2 = \{(x, Brussel), (y, Londen)\}$. Hiervoor zien we dat er geen uitbreiding is van de head van σ naar T via h_2. We kunnen de TGD σ dus toepassen. Dit doen we door het creëren van $T' = T \cup h_2(head(\sigma))$ Dit resulteert dus in het toevoegen van $verbinding(Londen, Brussel)$ aan ons tableau T.

Het resultaat is dan $T' = \{verbinding(Brussel, Londen), verbinding(Brussel, Parijs), verbinding(Parijs, Brussel), verbinding(Londen, Brussel)\}$. Dit tableau T' voldoet aan σ. Er kunnen dus geen verdere toepassingen van σ gedaan worden op het tableau. Dit is dus het resultaat van de chase. We zien dat de chase de schending van σ die ontstaat door het atoom $verbinding(Brussel, Londen)$ gerepareerd heeft door het atoom $verbinding(Londen, Brussel)$ toe te voegen.

8.4 Data Exchange

We bekijken de toepassing van de chase in verband met Data Exchange. De meeste van deze toepassingen zijn gebaseerd op het vinden van een universal solution of model. We beginnen dus met de introductie van dit begrip.

8.4.1 Universal model en solution

We noemen een tableau T' een *solution* voor (T, Σ) met T een tableau en Σ een verzameling TGD's als er voldaan is aan de volgende voorwaarden:

- $T \subseteq T'$
- $T \models \Sigma$

We kunnen dit uitbreiden zodat er ook gebruik gemaakt mag worden van EGD's.

We noemen een tableau U een*model* voor Σ en T als de volgende voorwaarden gelden:

- Er is een homomorfisme h van T naar U
- $U \models \Sigma$

We noemen een model of een solution U *universeel* als er homomorfismen bestaan van dit model of deze solution naar alle andere modellen of solutions voor Σ en T. Voor de rest van deze paragraaf gebruiken we de term model.

De chase is een algoritme voor het berekenen van deze universele modellen voor (T, Σ)[9]. De toepassing van de chase volgens Σ zal er immers voor zorgen dat het resultaat voldoet aan Σ.

Voorbeeld 8.4.1. *We bekijken een tableau* $T = \{R(a, b), R(b, c)\}$. *Verder hebben we een verzameling dependencies* $\Sigma = \{\sigma\}$ *met:*

$$\sigma = R(x, y) \wedge R(y, z) \rightarrow R(x, z)$$

We kunnen dan een tableau U *maken zodat* T' *een model is voor* (T, Σ). *Deze* $U = \{R(a, b), R(b, c), R(a, c)\}$. U *is een model aangezien* $U \models \Sigma$ *en er is een homomorfisme* $h = \{(a, a), (b, b), (c, c)\}$ *van* T *naar* U.

Het tableau $U' = \{R(a, b), R(a, c)\}$ *is geen model voor* (T, Σ). *Hoewel* $U' \models \Sigma$ *is er geen homomorfisme van* T *naar* U'.

Universele modellen worden gebruikt in verschillende gebieden in verband met databases. Bijvoorbeeld bij data exchange[9], data integration[29], conjunctive query containment[26] en query answering over ontologies[6].

8.4.2 In theorie

Data exchange is geïntroduceerd door Fagin in 2005[12]. Het probleem bij data exchange is de uitwisseling van data tussen twee verschillende schema's. We willen de data van het originele schema omzetten naar het doelschema. Verder moet de data ook voldoen aan de dependencies die gelden voor dit doelschema.

We noemen het originele schema de *source* van de data exchange en het doelschema de *target*. Voor de omzetting van data van het ene schema naar het andere maken we gebruik van source-to-target TGD's. Verder moeten we ook de EGD's en TGD's beschouwen die gelden voor het doelschema. Hieraan moet de omgezette data immers voldoen. De omzetting van de data kan gedaan worden via het chase algoritme. We voeren de chase uit op de originele data volgens de dependencies gegeven door de source-to-target TGD's en de dependencies die gelden voor het doelschema. Het resultaat van deze chase geeft ons een universeel model voor de gegeven instance en verzameling

dependencies. Dit resultaat is dan ook meteen een oplossing voor ons data exchange probleem. Merk op dit we hier zeker rekening moeten houden met voorwaarden voor terminatie van de chase. Als de chase niet eindigt gaan we immers geen oplossing vinden.

We kunnen een dergelijke data exchange probleem formeel beschrijven. Veronderstel een source schema S_1 en een target schema S_2 zodat $\mathsf{S}_1 \cap \mathsf{S}_2 = \emptyset$, een verzameling dependencies Σ over $\mathsf{S}_1 \cup \mathsf{S}_2$ en een tableau T over S_1. Het data exchange probleem is dan het vinden van een tableau T_2 over schema S_2 zodat $T \cup T_2 \models \Sigma$. Hierbij is $\Sigma = \Sigma_{st} \cup \Sigma_t$ met Σ_{st} de verzameling source-to-target TGD's en Σ_t de dependencies die gelden voor de target. $chase_\Sigma(T)$ is een oplossing voor dit probleem als dit gedefinieerd is (dus als de chase eindigt)[12].

De source-to-target TGD's zijn van de vorm $\phi(\overline{x}) \rightarrow \exists y\ \phi(\overline{x}, \overline{y})$. Hierbij bevat ϕ enkel relaties uit source S_1 en ϕ uit target S_2. Intuïtief zien we dus dat de body van een dergelijke TGD zal bestaan uit data uit de source die omgezet wordt naar de data uit de target. Deze target data staat dus in de head van de TGD. Bij de uitvoering van de chase volgens deze dependencies zal de source data dus leidden tot de aanmaak van target data.

We geven een voorbeeld van een data exchange probleem.

Voorbeeld 8.4.2. *We beschouwen een source schema S_1 met relatie-symbolen P, Q, R elk met ariteit drie. Het target schema S_2 heeft slechts een enkel relatie-symbool O met ariteit drie. We leggen geen verdere beperkingen op aan de target dus $\Sigma_t = \emptyset$. De omzetting van de data verloopt volgens $\Sigma_{st} = \{\sigma_1, \sigma_2, \sigma_3\}$ met :*

- $\sigma_1 = P(x_1, x_2, x_3) \rightarrow \exists y, z\ O(x_1, y, z)$
- $\sigma_2 = Q(x_1, x_2, x_3) \rightarrow \exists y, z\ O(y, x_2, z)$
- $\sigma_3 = R(x_1, x_2, x_3) \rightarrow \exists y, z\ O(y, z, x_3)$

We gaan de data exchange toepassen op een tableau T met:

$$T = \{P(a_0, b_1, c_1), Q(a_2, b_0, c_2), R(a_3, b_3, c_0)$$

We kunnen dan op zoek gaan voor een model U voor $(T, \Sigma_t \cup \Sigma_{st})$ Een mogelijke oplossing $U = \{O(a_0, y_0, z_0), O(x_1, b_0, z_1), O(x_2, y_2, c_0)\}$. Want $T \cup U \models \Sigma_{st}$ en $U \models \Sigma_t$.

Formeel ziet het gebruik van de chase om tot een universeel model te komen er als volgt uit: We vertrekken vanuit een tableau dat de combinatie is van input tableau T en een leeg tableau U dat zal dienen als resultaat voor de target, dus $T_0 = T \cup U$. Op deze input chasen we dan volgens $\Sigma = \Sigma_t \cup \Sigma_{st}$. De elementen die we toevoegen tijdens het chasen zullen dan telkens in U geplaatst worden. We weten dat er inderdaad geen nieuwe elementen aan T toegevoegd zullen worden. Dit is het geval omdat Σ_t enkel dependencies bevat die een impact hebben op de relaties in de target. Ook Σ_{st} zal geen nieuwe elementen toevoegen aan T. Na de uitvoering van de chase zal $T \cup U$ een universeel model zijn voor (T, Σ). U zal dan de oplossing zijn voor het data exchange probleem.

Voorbeeld 8.4.3. *We maken gebruik van de chase om een universeel model te berekenen voor het data exchange probleem uit Voorbeeld 8.4.2.*

We vertrekken vanuit $T_0 = T \cup U$. Voordat we aan de chase beginnen is tableau $U = \emptyset$. We chasen dan volgens Σ_{st}. We hoeven Σ_t niet te beschouwen aangezien deze leeg is.

We beginnen met het toepassen van σ_1 volgens $h = \{(x_1, a_0), (x_2, b_1), (x_3, c_1)\}$. Dit geeft ons $T_1 = T_0 \cup \{O(a_0, n_1, n_2)\}$. Onze oplossing U zal dus ook uitgebreid worden met $O(a_0, n_1, n_2)$.

Analoog passen we σ_2 en dan σ_3 toe. Dit geeft ons $T_3 = T_1 \cup \{O(n_3, b_0, n_4), O(n_5, n_6, c_0)\}$. We zien dan dat $T_3 \models \Sigma$. De chase is dus ten einde. T_3 is dus een universeel model voor (T, Σ). Een oplossing voor het data exchange probleem is te vinden in $U = \{O(a_0, n_1, n_2), O(n_3, b_0, n_4), O(n_5, n_6, c_0)\}$. We zien immers dat $T \cup U \models \Sigma_{st}$ en $U \models \Sigma_t$.

Een uitbreiding van dit probleem is data exchange in peer-to-peer netwerken. Fuxman heeft in 2006 aangetoond dat we ook in dit gebied gebruik kunnen maken van de chase het data exchange probleem op te lossen[15].

Voor meer details in verband met data exchange en verdere uitbreidingen verwijzen we naar het werk van Deutsch en Onet [9, 37].

8.4.3 Data integration

Data integration is een speciaal geval van data exchange.

Data integratie is het combineren van data van verschillende bronnen zodat een gebruiker een verenigd beeld krijgt van al deze data[29, 41]. Dit probleem is dus analoog aan het data exchange probleem. We moeten immers ook data uit verschillende schema's verenigen. Het verschil is echter dat we bij data exchange effectief de data gaan uitwisselen. De data zal opgeslagen worden in het target schema. Bij data integration hoeft dit niet het geval te zijn. We kunnen gewoon de data integreren om een beeld te krijgen van de volledige data zonder deze resultaten te materialiseren. Het probleem is dus de omzetting van een query over dit globale beeld naar een query over de verschillende bronnen.

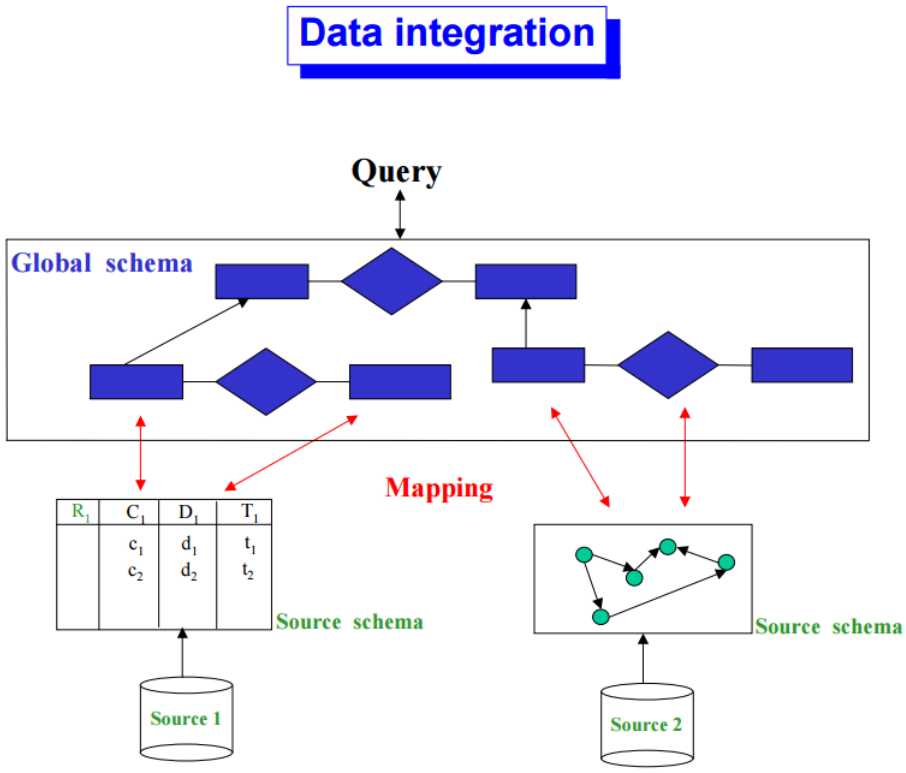
Er zijn verschillende manieren om het probleem van data integration aan te pakken. In de eerste aanpak maken we gebruik van een globaal schema, *global-as-view* dat uitgedrukt is volgens de verschillende source schema's [41]. De tweede aanpak noemt men de *local-as-view* aanpak [23]. Hier maakt men gebruik van een globaal schema dat onafhankelijk van de source schema's uitgedrukt is.

Een algemeen framework voor data integration bevat de volgende elementen[29]:

- S_G: Een globaal schema.
- S_S: Een source schema.
- M: een mapping tussen S_G en S_S zodat we een query q_S uit het source schema kunnen vertalen naar een query q_G uit het globale schema en omgekeerd.

Figuur 8.1 van Lenzerini [1] geeft een grafische voorstelling van een dergelijk probleem.

[1] `http://www.dis.uniroma1.it/~lenzerin/homepagine/talks/TutorialPODS02.pdf`



Figuur 8.1: *Deze slide van Lenzerini toont de verschillende onderdelen van een data integration probleem.*

Analoog aan het gebruik bij data exchange maken we gebruik van de chase bij het opvragen van informatie over deze globale schema's[17, 28]. We zien immers dat we van een databron volgens een bepaald schema overgaan naar een ander schema. Dit is dus volledig analoog aan data exchange. We maken gebruik van de chase om de overgang tussen verschillende schema's te maken. Voorbeeld 8.4.2 geeft dus ook een voorbeeld van data integration. We gaan immers van een bronschema naar een ande (globaal) schema volgens een bepaalde mapping. Het enige verschil is dat het resultaat geen echte tableau hoeft te zijn. Het kan ook gewoon om een tijdelijk antwoord op een query gaan waarbij het resultaat van de data exchange niet opgeslagen wordt

Meer details over het data integration framework zijn te vinden in het werk van Lenzerini [29]. Voor een overzicht van de geschiedenis en de verschillende onderdelen van data integration raden we het werk van Halvey aan[22].

8.4.4 In de praktijk

In deze paragraaf geven we meer gedetailleerde praktijkvoorbeelden van data exchange en integration. Hiervoor bespreken we de CLIO en ORCHESTRA systemen die gebruikt kunnen worden voor zowel data exchange als data integration.

Clio

De eerste toepassing die we introduceren is Clio[2]. Dit is een applicatie om aan data integratie en exchange te doen[11][35][21].

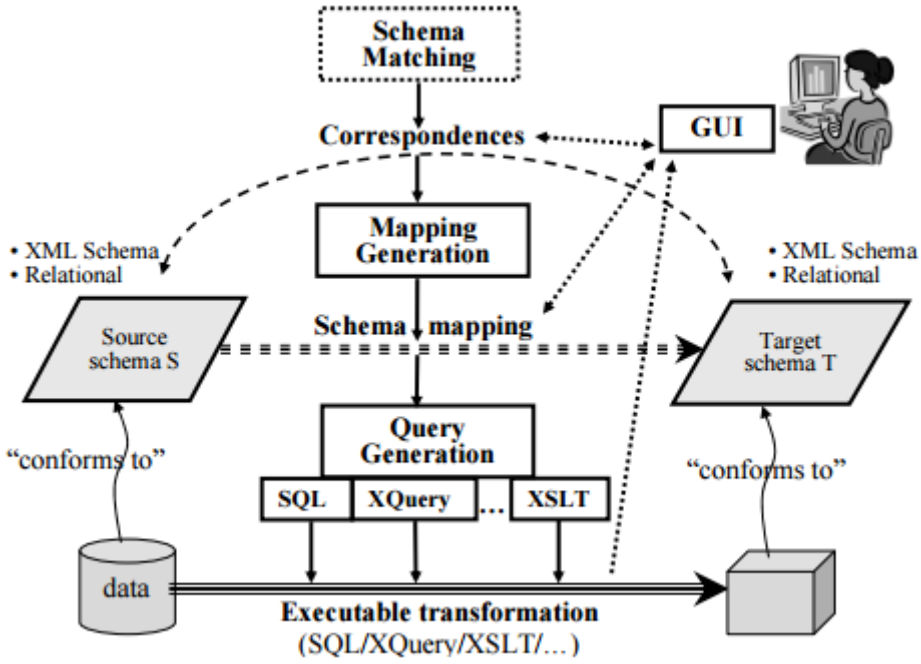
Dit project is ontstaan in 1999 als een samenwerking tussen het IBM Almaden Research Centre en de Univeristy of Toronto. Clio was een pionier in het gebruik van schema mappings, de manier waarop de relatie tussen twee verschillende schema's beschreven kan worden. Vanuit deze mapping tussen de source en de target kan Clio ofwel een view genereren om queries te vertalen en dus aan data integration te doen, ofwel kan Clio code genereren om de data om te zetten van het source schema naar het target schema en zo aan data exchange te doen.

We zullen nu in meer detail naar de werking van dit systeem kijken. Een overzicht van de architectuur van dit systeem kan gezien worden in Figuur 8.2 [21]. We zien dat Clio gebruik kan maken van zowel XML als relationele schema's. De belangrijkste onderdelen zijn duidelijk deze waar de mapping en de query generation gebeurt. Uiteraard is er ook een GUI waar de gebruiker de gegevens kan bekijken en aanpassen en waar de resultaten weergegeven kunnen worden.

Ons interesseert vooral hoe deze toepassing gebruik maakt van de chase. De toepassing van de chase in deze applicatie is te vinden bij de generatie van de mapping. Hierin wordt voor een bepaald tableau een uitbreiding gedaan met de foreign keys van dit tableau. De berekening van alle elementen die bereikbaar zijn via foreign keys vanuit een zeker tableau wordt gedaan via de toepassing van de chase. De chase wordt dus gebruikt om de sluiting of *closure* van deze tableaus te berekenen.

Deze tableaus die de elementen van de verschillende schema's beschrijven gaan dan gebruikt worden om de logische mapping tussen deze schema's te vinden. Hiervoor

[2]urlhttp://dblab.cs.toronto.edu/project/clio/



Figuur 8.2: *De architectuur van het Clio systeem.*

bekijken we alle tableaus uit de source met alle tableaus uit de target en zoeken we de overeenkomsten voor elk koppel. Als er overeenkomsten zijn dan is het koppel een kandidaat voor de logische mapping. De verdere berekening van de mapping en van eventuele queries maakt geen gebruik meer van de chase.

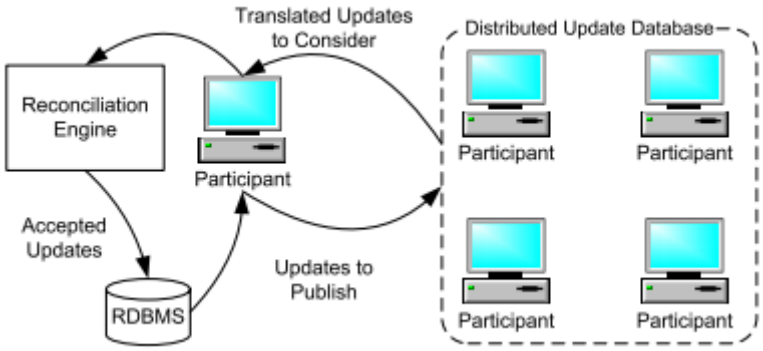
Orchestra

Een tweede applicatie in verband met data exchange en integration is Orchestra[3][25, 24, 20]. Orchestra is een *collaborative data sharing system*(CDSS). Het is een toepassing om data te delen over verschillende data schema's en met verschillende versies van deze data. De uitwisseling van data gebeurt peer-to-peer tussen de verschillende deelnemers aan het system. De architectuur van dit systeem is te zien in Figuur 8.3[20]. In het geval van Orchestra is dit systeem volledig peer-to-peer en wordt er geen gebruikt gemaakt van een centrale server. Hierbij kan elke peer P gebruik maken van een eigen schema en heeft elke peer lokale data. Deze lokale data wordt opgeslagen in een standaard DBMS en Orchestra werkt bovenop deze lokale database.

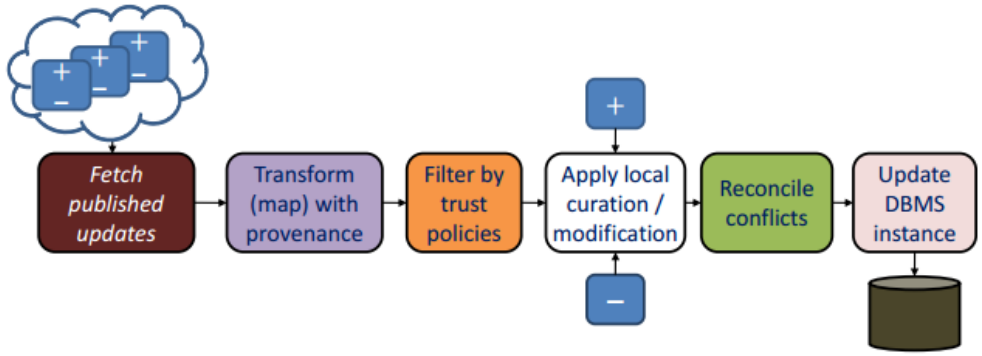
De belangrijkste aspecten van het Orchestra systeem zijn verdeeld in drie modules:

- Publicatie en archivering van update logs.
- Transformeren en filteren van updates.
- Het uitvoeren van een query over peers.

[3] `https://dbappserv.cis.upenn.edu/home/?q=node/8`



Figuur 8.3: *De architectuur van een CDSS*



Figuur 8.4: *De pipeline voor het toepassen van een update.*

De meest complexe van deze modules is de module die instaat voor het transformeren en filteren van updates. Hoe deze updates juist toegevoegd worden is te zien in Figuur 8.4[25]. Dit is ook het onderdeel waar gebruikt gemaakt wordt van de Chase. We zien immers dat het transformeren van data van een andere peer naar het eigen schema een toepassing is van data exchange. Orchestra lost dit op door voor elke peer een universal solution te berekenen. Verder maakt Orchestra gebruik van TGD's om de mapping tussen verschillende schema's uit te drukken.

Orchestra bouwt dus verder op het standaard model voor data exchange zoals in Paragraaf 8.4.2 en maakt dus gebruik van de chase om een universal solution te berekenen. De chase wordt echter niet direct toegepast. In plaats daarvan worden de mappings vertaald naar een programma in een uitgebreide versie van Datalog, met ondersteuning voor Skolem functies. Het resultaat van dit programma is dan een universal solution en dus een oplossing voor het data exchange probleem. Verder vereist de toepassing ook dat de verzameling van mappings zwak acyclisch is. De terminatie van de chase is immers belangrijk, anders is er geen resultaat. Orchestra maakt dus gebruik van Datalog om de chase te implementeren.

8.5 Query optimization

Query optimization is het probleem van het optimaliseren van een gegeven query. Dit probleem wordt aangepakt door voor een gegeven query te kijken of we een kleinere maar equivalente versie van deze query kunnen vinden.

We beginnen met het definiëren van een conjunctieve query q. Dit is een expressie van de vorm:

$$ans(\overline{x}) \leftarrow \phi(\overline{x}, \overline{z})$$

Hierbij zijn $\overline{x}$ en $\overline{z}$ sequenties van variabelen en ϕ is een conjunctie van relatie atomen. Verder moet elke variabele uit $\overline{x}$ die voorkomt in $ans(\overline{x})$ ook voorkomen in ϕ. $ans(\overline{x})$ is een atoom dat het resultaat van de query aangeeft. De verzameling relatie atomen uit ϕ noteren we als $db(q)$ en het atoom $ans(\overline{x})$ noteren we ook als $head(q)$.

We willen dan weten wat het resultaat uit van het uitvoeren van een dergelijke conjuctieve query q op een tableau T. Deze uitvoering noteren we als $q(T)$. Het resultaat van deze uitvoering zijn alle variabelen $\overline{a}$ zodat $T \models \exists \overline{z}\ \phi(\overline{a}, \overline{z})$.

$$q(T) = \{\overline{a} \in V_{\overline{x}} \mid T \models \exists \overline{z}\ \phi(\overline{a}, \overline{z})\}$$

Hierbij staat $V_{\overline{x}}$ voor de verzameling variabelen V beperkt tot de variabelen die voorkomen in $\overline{x}$.

We kunnen het verband tussen twee conjunctieve queries q en q' bekijken voor een zekere verzameling dependencies Σ. We zeggen dat q *contained* is in q' als en slechts als voor alle tableaus$T$ waarvoor geldt dat $T \models \Sigma$ geldt dat $q(T) \subseteq q'(T)$. We noteren dit als $q \sqsubseteq_\Sigma q'$.

Voor een lege verzameling dependencies kunnen we de notatie afkorten: $q \sqsubseteq_\emptyset q'$ kan afgekort worden naar $q \sqsubseteq q'$. Voor dit geval zien we dat $q \sqsubseteq q'$ als en slechts als er een homomorfisme h is van $db(q') \cup head(q')$ naar $db(q) \cup head(q)$. Dit resultaat is logisch aangezien dit gaat over het geval waar $\Sigma = \emptyset$. De tableaus die beschouwt moeten worden zijn dus alle mogelijke tableaus aangezien elke tableau voldoet aan $\emptyset$. Voor elk tableau T moet dus gelden dat $q(T) \subseteq q'(T)$. Als er een homomorfisme is van $db(q') \cup head(q')$ naar $db(q) \cup head(q)$ dan weten we dat de logische elementen van q' vervat zijn in q. q kan echter uitgebreider zijn dan q'. Dit wil zeggen dat het resultaat van het toepassen van q op een tableau T minder uitgebreid zal zijn dan het resultaat van de toepassing van q' op T en dus zal $q(T) \subseteq q'(T)$. Dus $q \sqsubseteq q'$.

Voor een willekeurige verzameling dependencies Σ die niet de lege verzameling is, is het echter moeilijke om te beslissen of $q \sqsubseteq_\Sigma q'$. Hier kunnen we gebruik maken van de chase om te reduceren naar het geval waar $\Sigma = \emptyset$. Voor q en q' twee conjunctieve queries en Σ een verzameling dependencies waarvoor $chase_\Sigma(db(q))$ bestaat, geldt dat $q \sqsubseteq_\Sigma q'$ als en slechts als $chase_\Sigma(head(q) \cup db(q)) \sqsubseteq q'$[26].

Intuïtief komt dit resultaat tot stand doordat de toepassing van de chase ervoor zorgt dat de dependencies uit Σ worden toegepast waardoor we enkel nog rekenen moeten houden met het geval waarin $\Sigma = \emptyset$. Het resultaat van de toepassing van de chase zal immers voldoen aan Σ. We verplaatsen het voldoen aan Σ dus van de $\sqsubseteq_\Sigma$ operatie, naar de chase. Hierdoor moeten we enkel nog een homomorfisme vinden nadat de chase is toegepast om containment te kunnen bepalen.

We vinden dus dat de chase een toepassing is om containment mee te bepalen. Aan de hand van deze containment kunnen we nu de link leggen met de optimalisering van

queries. Voor een conjunctieve query q gaan we immers op zoek gaan naar een kleiner query q' die wel nog gelijk is aan q. Deze gelijkheid testen we door af te dwingen dat $q \sqsubseteq_\Sigma q'$ en $q' \sqsubseteq_\Sigma q$. We gaan dus op zoek naar een *minimale herschrijving* voor q. We introduceren eerst een Σ-equivalente herschrijving. Voor een conjunctieve query q en een verzameling dependencies Σ is dit een herschrijving van q naar een query q' zodat $q \sqsubseteq_\Sigma q'$ en $q' \sqsubseteq_\Sigma q$. Een minimale herschrijving van een query q over Σ is dan een query q' zodat q' een Σ-equivalente herschrijving is van q en q' minimaal is in het aantal relatie-atomen in de body. Een minimale herschrijving is de meeste efficiënte voorstelling van een query aangezien deze het kortste is en dit zal meestal leidden tot de snelste uitvoering.

Voorbeeld 8.5.1. *We bekijken een voorbeeld van containment bij conjunctieve queries. We bekijken twee conjunctieve query q_1, q_2 met:*

- $q1 = ans(x_1, x_2) \leftarrow R(x_1, x_3) \wedge S(x_3, x_4) \wedge R(x_4, x_2)$
- $q2 = ans(x_1, x_2) \leftarrow R(x_1, x_3) \wedge S(x_3, x_3) \wedge R(x_3, x_2)$

Voor deze twee queries kunnen we zien dat $q_2 \sqsubseteq q_1$. We gaan alle resultaten voor q_2 immers ook kunnen vinden via q_1 door het geval $x_4 = x_3$ te beschouwen. Er zijn echter ook gevallen waarin $x_3 \neq x_4$ dus de omgekeerde richting is niet het geval. Dit is dus niet een geval waarin we q_2 zouden kunnen vervangen door q_1.

We bekijken nu een algoritme om de minimale herschrijvingen van een query q te berekenen voor een bepaalde verzameling dependencies Σ. Dit algoritme is het chase en backchase (C & B) algoritme[10, 39]. De chase wordt gebruikt als een onderdeel van dit algoritme. Dit algoritme verloopt in twee fasen. In de eerste fase, de chase fase, gebruiken we Σ om de chase toe te passen op q. Dit geeft ons als resultaat alle mogelijk alternatieve manieren waarop we q kunnen beantwoorden. In de tweede fase, de backchase fase, gaan we alle mogelijk subqueries van dit resultaat bekijken om zo de minimale subqueries te vinden. In deze fase gebruiken we de chase om te testen of de subqueries equivalent zijn met q (volgens $q \sqsubseteq_\Sigma q'$ en $q' \sqsubseteq_\Sigma q$). Op het einde van de backchase fase geeft ons dit alle minimale subqueries van q.

Dit algoritme werkt beter dan gewoon het schrappen van atomen uit q en daarna controleren of het resultaat van deze schrapping equivalent is. De reden dat dit het geval is, is dat het C& B algoritme ook atomen kan introduceren die niet origineel aanwezig waren in q. Het geeft dus alle mogelijke minimale herschrijvingen en niet elke diegene die gebruik maken van de atomen in q.

Een voorbeeld van dit algoritme is te vinden in Paragraaf 8.6.2.

Verder merken we op dat het bij beide fasen van dit algoritme belangrijk is dat de chase eindigt. Het is dus belangrijk om gebruik te maken van een van de terminatie condities.

8.6 Motiverend voorbeeld

We geven een voorbeeld in verband met een klassieke winkelketen. Dit voorbeeld is gebaseerd op een voorbeeld van Deutsch in verband met query optimization [10]. Dit is een interessant voorbeeld omdat het verschillende toepassingen van de chase samenbrengt. We zien immers dat we voor een eenvoudig voorbeeld al snel meerdere mogelijke toepassingen van de chase vinden.

Het voorbeeld bekijkt een winkelketen op verschillende locaties. Het database systeem is gedistribueerd over de verschillende locaties. Hierbij zijn locatie 1 en 3 interne vestigingen. Locatie 2 is een externe database die een catalogus van leveranciers aanbiedt. Elke locatie maakt gebruikt van een aparte database met een apart databaseschema. Deze schema's zien er als volgt uit:

- $\mathsf{S}_1 = \{(WebOrder, 4), (Cust, 2)\}$
- $\mathsf{S}_2 = \{(SuppCatalog, 4)\}$
- $\mathsf{S}_3 = \{(MasterSupp, 4), (Supp2Cust, 3), (MasterCust, 3)\}$.

We zullen nu de elementen van deze relaties in meer detail bekijken zodat we later dependencies kunnen bepalen voor de relaties.

Relatie	Attributen
Weborder	part, supp_id, orderkey, cust_id,qty
Cust	cust_id, cnation
SuppCatalog	supp_id, saddr, snation, directory
MasterSupp	supp_id, saddr, snation,history
Supp2Cust	supp_id, orderkey, cust_id
MasterCust	cust_id, cnation, caddr

We beginnen met het introduceren van dependencies voor onze keys:

- $k_1 = Cust(c, n) \wedge Cust(c, n') \rightarrow n = n'$
- $k_2 = MasterCust(c, n, a) \wedge MasterCust(c, n', a') \rightarrow (n = n') \wedge (a = a')$
- $k_3 = MasterSupp(s, a, n, h) \wedge MasterSupp(s, a', n', h') \rightarrow (a = a') \wedge (n = n') \wedge (h = h')$

Merk op dat we hier gebruik maken van een conjunctie in de head van onze EGD's. Dit is een afkorting voor het gebruik van een aparte EGD voor elk gelijkheidsatoom in deze conjunctie.

Om de foreign keys te beschrijven maken we gebruik van TGD's:

- $f_1 = WebOrder(p, s, o, c, q) \rightarrow \exists n\ Cust(c, n)$
- $f_2 = Supp2Cust(s, o, c) \rightarrow \exists a, n, h\ MasterSupp(s, a, n, h)$
- $f_3 = Supp2Cust(s, o, c) \rightarrow \exists a, n\ MasterCust(c, n, a)$

We kunnen ook gebruik maken van dependencies om algemene condities op te leggen aan onze data. We introduceren bijvoorbeeld een nieuwe EGD e die uitdrukt dat elke customer_id een unieke landcode moet hebben:

$$e = Cust(c, n) \wedge MasterCust(c, n', a) \rightarrow n = n'$$

We hebben dan onze verzameling dependencies $\Sigma = \{e, k_1, k_2, k_3, f_1, f_2, f_3\}$.

Nu we onze verschillende dependencies hebben die moeten gelden zien we meteen een eerste mogelijke toepassing van de chase. We hebben immers gezien dat de chase

gebruikt kan worden om ervoor te zorgen dat een bepaald tableau voldoet aan onze verzameling dependencies Σ. We kunnen dus gebruik maken van de chase om af te dwingen dat onze data voldoet aan de vereisten voor deze data.

We zouden ook gebruik kunnen maken van het feit dat de chase een oplossing biedt voor het implicatie probleem. Stel dat we een nieuwe dependency σ willen toevoegen. We kunnen dan controleren of $\Sigma \models \sigma$ door gebruik te maken van de chase. Als dit het geval is dan weten we dat het toevoegen van σ niet zal leidden tot veranderingen in onze data als deze al voldoet aan $Sigma$.

8.6.1 Data exchange

Uiteraard zal de data tussen deze drie locaties uitgewisseld moeten worden. Ook willen we de mogelijkheid om alle locaties tegelijk te queryen. We kunnen dus zowel data exchange als data integration toepassen. Dit zijn beide problemen waarbij de oplossing gebruikt maakt van de chase. We zien dus al een derde toepassingsmogelijkheid van de chase in verband met ons voorbeeld.

Stel bijvoorbeeld dat we gebruik willen maken van een mappping m_1 van locatie 1 en 2 naar locatie 3 die aangeeft dat de Master data aangevuld wordt met de data van locatie 1 en 2. We kunnen dan een source-to-target dependency definiëren van de volgende vorm:

$$\begin{aligned} m_1 =& WebOrder(p, s, o, c, q) \wedge Cust(c, cn) \wedge SuppCatalog(s, sa, sn, d) \rightarrow \\ & \exists h, ca\ MasterSupp(s, sa, sn, h) \wedge Supp2Cust(s, o, c) \wedge MasterCust(c, cn, ca)) \end{aligned}$$

Een ander voorbeeld is een mogelijke mapping m_2. Deze mapping stelt dat SuppCatalog de autoriteit is op heet vlak van informatie over de leveranciers. Dit wil zeggen dat elke leverancier uit MasterSupp op locatie 3 gevonden zal kunnen worden in de catalogus op locatie 2. De TGD die deze mapping weergeeft heeft de volgende vorm:

$$m_2 = MasterSupp(s, sa, sn, h) \rightarrow \exists d\ SuppCatalog(s, sa, sn, d)$$

We kunnen dan gebruik maken van een data integration en data exchange systeem zoals bijvoorbeeld Orchestra of CLIO om de informatie tussen de verschillende locaties te delen. Het is dus duidelijk dat het delen van informatie tussen de verschillende locaties gebruikt maakt van de chase.

8.6.2 Herformulering van queries

We beschouwen de volgende conjunctieve query:

$$q(p, c, sa, sn) \leftarrow WebOrder(p, s, o, c, q) \wedge SuppCatalog(s, sa, sn, d)$$

Deze query geeft alle onderdelen die besteld zijn in locatie 1 met het adres en het land van de leverancier en het id van de klant. Merk op dat deze query gebruik maakt van data uit zowel locatie 1 als locatie 2.

We geven een voorbeeld van een query q' die equivalent is:

$$q'(p, c, sa, sn) \leftarrow WebOrder(p, s, o, c, q) \wedge MasterSupp(s, sa, sn, h)$$

q' maakt gebruik van locaties 1 en 3 en niet meer van locatie 2. De kans bestaat dus dat deze query sneller uitgevoerd zal worden. We herinneren ons immers dat locatie 2 een externe catalogus van leveranciers is terwijl locatie 1 en 3 gebruik maken van het intern databasesysteem.

Dit is een duidelijk voorbeeld van de herformulering van queries waarbij de herformulering niet voor de hand ligt. We leggen nu aan de hand van dit voorbeeld uit hoe het C & B algoritme gebruikt kan worden om op een systematische manier alle mogelijke herformuleringen op te sommen.

We beschouwen opnieuw query q. We zitten dan in de eerste fase, de chase fase, van het C & B algoritme. In deze face gaan we de chase toepassen op de body van onze query.

We kunnen een chase step doen volgens f_1. Dit geeft ons :

$$\begin{aligned} q_1(p, c, sa, sn) \leftarrow & WebOrder(p, s, o, c, q) \wedge SuppCatallog(s, sa, sn, h) \wedge \\ & Cust(c, cn) \end{aligned}$$

Merk op dat cn hierbij een nieuw geïntroduceerde variabele is.

We kunnen dan een volgende chase step toepassen volgens m_1. Dit geeft ons:

$$\begin{aligned} q_2(p, c, sa, sn) \leftarrow & WebOrder(p, s, o, c, q) \wedge SuppCatallog(s, sa, sn, h) \wedge \\ & Cust(c, cn) \wedge \\ & MasterSupp(s, sa, sn, h) \wedge \\ & Supp2Cust(s, o, c) \wedge MasterCust(c, cn, ca) \end{aligned}$$

Vanaf hier zijn er geen verdere chase steps meer mogelijk. q_2 geeft ons een overzicht van alle relaties die kunnen bijdragen tot deze query.

We gaan nu over tot de backchase fase. In deze fase gaan we de minimale subqueries van q_2 opsommen. We bouwen de mogelijke subqueries door atomen uit de body van q_2 te kiezen zodat de variabelen uit de head nog steeds voorkomen in deze nieuwe body.

Een van deze mogelijke subqueries is q'. Aangezien we weten dat $q_2 \equiv_\Sigma q$ (de chase behoudt de equivalentie) en $q_2 \subseteq q'$ (wegens de constructie) moeten we enkel nog controleren dat $q' \sqsubseteq_\Sigma q$. Dit doen we door q' te chasen volgens Σ.

We kunnen een chase step doen volgens m_2. Dit geeft ons het volgende resultaat:

$$\begin{aligned} q_2'(p, c, sa, sn) \leftarrow & WebOrder(p, s, o, c, q) \wedge MasterSupp(s, sa, sn, h) \wedge \\ & SuppCatalog(s, sa, sn, d) \end{aligned}$$

We zouden nu nog verder kunnen chasen door gebruik te maken van f_1 en daarna van m_1. We kunnen echter na deze stap al zien dat er een mapping is van q naar q_2' volgens de identiteit. Het is dus duidelijk dat $q' \sqsubseteq_\Sigma q$. We weten dus dat $q \equiv_\Sigma q'$. Verder weten

we dat q' minimaal is. De constructie van q' zorgt er immers voor dat deze slechts een minimaal aantal atomen bevat in zijn body. We hebben dus een minimale herschrijving van q gevonden.

We kunnen op eenzelfde manier alle andere minimale herschrijvingen van q vinden in deze backchase fase. Een voorbeeld van een van de andere minimale herschrijvingen die gevonden wordt door het algoritme is q zelf.

Hiermee sluiten we het voorbeeld af. We hebben gezien dat er verschillende toepassingen van de chase gebruikt kunnen worden binnen een simpel databasesysteem. Verder hebben we ook een gedetailleerd voorbeeld gegeven van de uitvoering van het C & B algoritme.

Hoofdstuk 9

Implementatie

In het kader van deze thesis hebben we ook een implementatie van de chase gerealiseerd. Deze implementatie hebben we vanaf nul gemaakt om de verschillende onderdelen van het algoritme in detail te bestuderen. De implementatie is gemaakt in Python[1].

9.1 Architectuur

Onze implementatie bestaat uit een aantal verschillende onderdelen, analoog aan de onderdelen van het chase algoritme. We hebben een klasse voor elke soort dependency, dus voor TGD's en EGD's. Uiteraard hebben we ook een Tableau klasse aangezien we hierop onze chase zullen moeten uitvoeren. Het algoritme zelf word beschreven in de aparte Chase klasse.

De basis voor onze architectuur zit in de TGD en EGD klassen. Deze klassen nemen een string met een TGD of EGD als input en parsen deze naar een formaat waarin de data makkelijk verwerkt kan worden. Belangrijk voor deze TGD's en EGD's is dat de body van deze dependencies omgezet kan worden naar een tableau.

De Tableau klasse slaat een tableau op. Dit bestaat uit een 2D array van variabelen, deze variabelen worden opgeslagen als een string. Elke rij in dit tableau heeft ook een ID. Deze ID geeft aan bij welke relatie ze horen. Stel dat we een tableau $T = \{R(a, b), S(u, v, w)\}$ hebben. In onze tableau klasse zal dit dan opgeslagen worden als:

rowID	row
R	a b
S	u v w

Aan de hand van deze tableaus kunnen we dan het chase algoritme implementeren. We schetsen nu het verloop van onze implementatie van het chase algoritme. We vertrekken vanuit een starttableau T en een verzameling Σ van TGD's en EGD's. We gaan dan de verschillende TGD's en EGD's in deze verzameling af. Voor elk van deze dependencies stellen we alle mogelijke homomorfismen op van de body van de dependency naar het tableau T. Voor al deze homomorfismen testen we dan of we de dependency kunnen

[1] https://www.python.org/

toepassen of niet. Dit herhalen we totdat we een iteratie doen over alle dependencies waarbij er geen verandering meer optreed. De pseudo-code van deze implementatie is te zien in Algoritme 2. We gaan dus elk mogelijk homomorfisme (in de code mapping genoemd) voor een dependency af voordat we proberen de volgende dependency toe te passen.

Bij het toepassen van de dependency op het tableau kan het zijn dat we nieuwe variabelen moeten introduceren. Hiervoor houden we een teller bij. De nieuwe variabele wordt dan weergegeven met de string gegeven door de concatenatie van n en de teller. De nieuw geïntroduceerde variabelen zijn dus $n_1, n_2, \ldots$.

Algorithm 2 Pseudo-code voor het chase algoritme uit onze implementatie.

```
input: Σ, k een verzameling TGD's en EGD's en een tableau T
begin
    while changes do :
        for σ in Σ:
            for mapping in σ.getAllMapping(T):
                if not T ⊨ σ :
                    pas σ toe op T volgens mapping
                else :
                    merk op dat er geen change is voor deze mapping.
    output T
end
```

Er zijn twee complexe onderdelen van deze implementatie. Eén ervan is het vinden van homomorfismen en de andere is het controleren of een tableau voldoet aan een zekere dependency. We zullen nu deze twee onderdelen in meer detail bekijken.

9.2 Controleren of een tableau voldoet aan een dependency

We willen voor een zeker tableau T kunnen controleren of dit tableau voldoet aan een bepaalde dependency σ zodat $T \models \sigma$. Dit is belangrijk omdat we σ enkel gaan toepassen tijdens de chase als T niet voldoet aan deze Σ. Om dit te controleren gaan we alle mogelijk homomorfismen bekijken van σ naar T. Als er geen homomorfismen zijn dan weten we dat $T \models \sigma$ en kunnen we $True$ als resultaat geven.

Als deze er wel zijn dan kunnen we controleren of de body van σ vervat zit in T. Dit doen we door een tableau T_σ te creëren vanuit de body van σ en hierop het homomorfisme toe te passen. Als het resultaat hiervan vervat zit in T dan weten we dat de head van σ ook moet gelden in T. Dit kunnen we controleren door het homomorfisme uit te breiden naar de head en dan te controleren of T deze head bevat in het geval van een TGD, of voldoet aan de gelijkheid in het geval van een EGD. Als dit het geval is dan weten we dat we $True$ kunnen teruggeven als resultaat. Als dit niet het geval is blijven we de andere homomorfismen proberen. Als er geen enkel homomorfisme is dat leidt tot een $True$ resultaat dan geven we $False$ als resultaat.

9.3 Genereren van homomorfismen

Het genereren van homomorfismen is het belangrijkste en meest complexe onderdeel van onze implementatie. Het is zeer belangrijk om voor alle tableaus en dependencies te kunnen testen of deze met elkaar in verband staan of niet. Het is ook het onderdeel van het programma dat de meeste tijd in beslag neemt.

We gaan dus op zoek naar mogelijk homomorfismen van een bron tableau T_S naar een doel tableau T_D. Om de mogelijke homomorfismen te berekenen vertrekken we vanuit twee verzamelingen, een verzameling bron variabelen S en een verzameling doel variabelen D. De bronverzameling bevat de variabelen uit tableau T_S en de doelverzameling bevat de variabelen uit T_D.

Elk homomorfisme gaat dan voor alle bron variabelen een doel variabele zoeken waarop deze afgebeeld wordt. We beginnen met het genereren van elke mogelijke combinaties van S naar D. Dit doen we door het cartesisch product te nemen van de doel variabelen D voor de lengte van de bron variabelen. We berekenen dus $D \times D \times \ldots \times D$ zodat de lengte van het resultaat hiervan overeenkomt met de lengte van S. Als we de elementen van S dan afbeelden op de overeenkomstige posities van dit product dan vinden we alle mogelijke combinaties van variabelen. We hoeven dan enkel nog voor het homomorfisme dat overeenkomt met deze combinatie te testen dat dit ook effectief voorkomt in T_D. Dit doen we door te controleren dat voor elk tupel t uit T_S het resultaat van het toepassen van het homorfisme op t voorkomt in T_D. Als dit het geval is dan slaan we het homomorfisme op. Als dit niet het geval is dan gaan we door naar de volgende combinatie.

We zien dat hier een groot deel van de complexiteit vervat zit. Als we s de kardinaliteit van S en d de kardinaliteit van D nemen dan moeten we $s \times d^s$ combinaties controleren. We zien dus dat het hier gaat over exponentiële complexiteit. Dit is dus het deel van de implementatie dat zorgt voor de grootste vertraging van de uitvoer. We geven een kort voorbeeld van de generatie van deze combinaties.

Voorbeeld 9.3.1. *We hebben een TGD* $\sigma = verbinding(x_1, x_2) \rightarrow verbinding(x_2, x_1)$. *Het tableau dat overeenkomt met de body van* σ *,* $T_\sigma = \{verbinding(x_1, x_2)\}$. *We zoeken nu een homomorfisme van* T_σ *naar* $T = \{verbinding(\text{"}Amsterdam\text{"}, \text{"}Brussel\text{"})\}$. *De verzameling bron variabelen* $S = \{x_1, x_2\}$ *en de verzameling doel variabelen is* $D = \{\text{"}Amsterdam\text{"}, \text{"}Brussel\text{"}\}$. *We kunnen dan de mogelijke combinaties berekenen. We beginnen met het berekenen van* $D \times D$. *Dit geeft ons:*

$$D' = \{(\text{"}Amsterdam\text{"}, \text{"}Amsterdam\text{"}), (\text{"}Amsterdam\text{"}, \text{"}Brussel\text{"}),\\ (\text{"}Brussel\text{"}, \text{"}Amsterdam\text{"}), (\text{"}Brussel\text{"}, \text{"}Brussel\text{"})\}$$

Dit combineren we nu met S *zodat de elementen op overeenkomstige posities gemapt worden. Dit geeft ons de volgende resultaten:*

$(x_1 : \text{"}Amsterdam\text{"}, x_2 : \text{"}Amsterdam\text{"})$
$(x_1 : \text{"}Amsterdam\text{"}, x_2 : \text{"}Brussel\text{"})$
$(x_1 : \text{"}Brussel\text{"}, x_2 : \text{"}Amsterdam\text{"})$
$(x_1 : \text{"}Brussel\text{"}, x_2 : \text{"}Brussel\text{"})$

Uit deze combinaties zullen we het homomorfisme h *kiezen dat overeenkomt met de tweede combinatie,* $h : \{(x_1, \text{"}Amsterdam\text{"}), (x_2, \text{"}Brussel\text{"})\}$. *We zien immers dat* $verbinding(x_1, x_2)$ *voorkomt in* T_σ. *Verder zien we dat het resultaat van het homormorfisme* $h(verbinding(x_1, x_2)) = verbinding(\text{"}Amsterdam\text{"}, \text{"}Brussel\text{"})$ *voorkomt in*

T. Dit is dus een homormofisme dat we kunnen gebruiken bij onze uitvoering van de chase. Als dit resultaat niet voorkwam in T dan was het geen bruikbaar homomorfisme.

9.4 Verbeteringen

Het belangrijkste aspect waar verbeteringen gemaakt kunnen worden is bij de generatie van homomorfismen. Het is duidelijk dat de generatie en controle van de verschillende homomorfismen het meeste complexiteit in beslag neemt. Het zou dus beter zijn om gebruik te maken van reeds bestaande en geoptimaliseerde manieren om dit te doen. Hiervoor kunnen we gebruik maken van reeds bestaande databasetechnieken.

Het oplossen van een conjunctieve query komt immers overeen met het zoeken of er een homomorfisme is. De query zal immers een mapping maken van zijn body naar een instance, net zoals een homomorfisme een maping maakt van het ene tableau naar het andere.

Een andere verbetering is de manier waarop er omgegaan wordt met oneindige chase sequences. Voorlopig controleren we gewoon hoeveel stappen er gedaan worden in de chase. Als het te lang duurt om tot een resultaat te komen gaan we ervan uit dat de chase oneindig is. Het zou echter beter zijn om hiervoor gebruik te maken van een monitor graph. Dit is een betere heuristiek om te voorspellen of de chase zal eindigen of niet.

Hoofdstuk 10

Conclusie

We kijken wat we kunnen concluderen in verband met de terminatie en de toepassingen van de chase en de manier waarop deze met elkaar in verband staan

10.1 Terminatie

We geven een overzicht van de verschillende terminatie condities die we beschouwd hebben.

techniek	klasse	gebaseerd op	compl.
Zwakke acycliciteit	$CT_{\forall\forall}$	dependency graph	P
SwA	$CT_{\forall\forall}$	vuur-relatie $\hookrightarrow_\Sigma$	P
C-stratificatie	$CT_{\forall\forall}$	c-chase graph ($\prec_C$)	coNP
Safe dependencies	$CT_{\forall\forall}$	propagation graph	coNP
Safely restricted	$CT_{\forall\forall}$	2-restrictie systeem ($\prec_P$)	coNP
Inductively restricted	$CT_{\forall\forall}$	$\mathbf{part}(\Sigma, k)$	coNP
Stratificatie	$CT_{\forall\exists}$	chase graph ($\prec$)	coNP
irrelevantie	$CT_{T,\forall}$	$WCC_\Sigma(\sigma_T)$ en $CT_{\forall\forall}$	NP
irrelevantie	$CT_{T,\exists}$	$WCC_\Sigma(\sigma_T)$ en $CT_{\forall\exists}$	NP
monitoring	$CT_{T,\forall}, CT_{T,\exists}$	monitor graph	

We zien dus dat er veel verschillende methoden zijn waarvan de meeste op elkaar gebaseerd zijn. De basis voor deze terminatie is telkens een syntactische methode zoals zwakke acycliciteit.

De meer complexe terminatie condities definiëren dan telkens een manier om de verzameling dependencies op te delen in deelverzamelingen zodat we voor deze deelverzamelingen een terminatie conditie kunnen toepassen en dit een resultaat geeft voor de volledige verzameling. We merken hierbij op dat terminatie condities die zwakker zijn een grotere complexiteit hebben.

10.2 Toepassingen

We hebben gezien dat de chase zeer veel toepassingen heeft in verschillende gebieden. De toepassingen die we bekeken hebben zijn:

- Oplossen van het implicatieprobleem.
- Reperaren van een tableau.
- Data exchange en data integration.
- Query optimization

Al deze gebieden hebben echter iets gemeenschappelijk. Ze hebben allemaal het resultaat van de chase nodig. Dit resultaat bestaat echter enkel als de chase eindigt.

Het voorbeeld uit Paragraaf 8.6 toont duidelijk aan dat al deze toepassingen een plaats hebben binnen een modern databasesysteem.

10.3 Algemeen

We zien dus dat de chase veel toepassingen heeft die gebruik maken van het resultaat van de chase. Voor het praktisch gebruik van de chase is het dus zeer belangrijk om gebruik te maken van terminatie condities. We hebben verschillende terminatie condities gezien. Deze bieden de mogelijkheid om de terminatie van de chase te verzekeren. We kunnen gebruik maken van verschillende condities. Hier kunnen we een afweging maken tussen de complexiteit van de terminatie conditie en de hoeveelheid verzamelingen die voldoen aan de conditie. Meer complexe condities hebben immers meer mogelijke oplossingen maar kosten ook meer tijd om te berekenen.

We kunnen dus concluderen dat de chase zeer belangrijk is en vaak gebruikt wordt en dat de terminatie een belangrijke rol speelt bij het gebruik.

Bibliografie

[1] Serge Abiteboul, Richard Hull en Victor Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

[2] Alfred V Aho, Catriel Beeri en Jeffrey D Ullman. "The theory of joins in relational databases". In: *ACM Transactions on Database Systems (TODS)* 4.3 (1979), p. 297–314.

[3] Catriel Beeri en Moshe Y Vardi. "A proof procedure for data dependencies". In: *Journal of the ACM (JACM)* 31.4 (1984), p. 718–741.

[4] Catriel Beeri en Moshe Y. Vardi. "Formal systems for tuple and equality generating dependencies". In: *SIAM Journal on Computing* 13.1 (1984), p. 76–98.

[5] Andrea Calı, Georg Gottlob en Michael Kifer. "Taming the infinite chase: Query answering under expressive relational constraints". In: *Proc. of KR* (2008), p. 70–80.

[6] Andrea Calì, Georg Gottlob en Thomas Lukasiewicz. "A general datalog-based framework for tractable query answering over ontologies". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 14 (2012), p. 57–83.

[7] Stavros S Cosmadakis, Paris C Kanellakis en Moshe Y Vardi. "Polynomial-time implication problems for unary inclusion dependencies". In: *Journal of the ACM (JACM)* 37.1 (1990), p. 15–46.

[8] Alin Deutsch, Bertram Ludäscher en Alan Nash. "Rewriting queries using views with access patterns under integrity constraints". In: *Theoretical Computer Science* 371.3 (2007), p. 200–226.

[9] Alin Deutsch, Alan Nash en Jeff Remmel. "The Chase Revisited". In: (2008).

[10] Alin Deutsch, Lucian Popa en Val Tannen. "Query reformulation with constraints". In: *ACM SIGMOD Record* 35.1 (2006), p. 65–73.

[11] Ronald Fagin e.a. "Clio: Schema mapping creation and data exchange". In: *Conceptual Modeling: Foundations and Applications*. Springer, 2009, p. 198–236.

[12] Ronald Fagin e.a. "Data exchange: semantics and query answering". In: *Theoretical Computer Science* 336.1 (2005), p. 89–124.

[13] Wenfei Fan en Leonid Libkin. "On XML integrity constraints in the presence of DTDs". In: *Journal of the ACM (JACM)* 49.3 (2002), p. 368–406.

[14] Wenfei Fan en Jérôme Siméon. "Integrity constraints for XML". In: *Journal of Computer and System Sciences* 66.1 (2003), p. 254–291.

[15] Ariel Fuxman e.a. "Peer data exchange". In: *ACM Transactions on Database Systems (TODS)* 31.4 (2006), p. 1454–1498.

[16] Tomasz Gogacz en Jerzy Marcinkowski. “Termination of oblivious chase is undecidable”. In: *CoRR* abs/1401.4840 (2014). URL: http://arxiv.org/abs/1401.4840.

[17] Gösta Grahne en Alberto O Mendelzon. “Tableau Techniques for Querying Information Sources through Global Schemas”. In: *Database Theory–ICDT'99* (), p. 332.

[18] Sergio Greco en Francesca Spezzano. “Chase termination: A constraints rewriting approach”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), p. 93–104.

[19] Sergio Greco, Francesca Spezzano en Irina Trubitsyna. “Stratification criteria and rewriting techniques for checking chase termination”. In: *Proceedings of the VLDB Endowment* 4.11 (2011), p. 1158–1168.

[20] Todd J Green e.a. “ORCHESTRA: Facilitating collaborative data sharing”. In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, p. 1131–1133.

[21] Laura M Haas e.a. “Clio grows up: from research prototype to industrial tool”. In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, p. 805–810.

[22] Alon Halevy, Anand Rajaraman en Joann Ordille. “Data integration: the teenage years”. In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment. 2006, p. 9–16.

[23] Alon Y Halevy. “Answering queries using views: A survey”. In: *The VLDB Journal* 10.4 (2001), p. 270–294.

[24] Zachary G Ives e.a. “ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data.” In: *CIDR*. 2005, p. 107–118.

[25] Zachary G Ives e.a. “The ORCHESTRA collaborative data sharing system”. In: *ACM SIGMOD Record* 37.3 (2008), p. 26–32.

[26] David S Johnson en Anthony Klug. “Testing containment of conjunctive queries under functional and inclusion dependencies”. In: *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*. ACM. 1982, p. 164–169.

[27] Kevin Knight. “Unification: A multidisciplinary survey”. In: *ACM Computing Surveys (CSUR)* 21.1 (1989), p. 93–124.

[28] George Konstantinidis en José Luis Ambite. “Optimizing the Chase: Scalable Data Integration under Constraints”. In: *Proceedings of the VLDB Endowment* 7.14 (2014).

[29] Maurizio Lenzerini. “Data integration: A theoretical perspective”. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2002, p. 233–246.

[30] David Maier, Alberto O Mendelzon en Yehoshua Sagiv. “Testing implications of data dependencies”. In: *ACM Transactions on Database Systems (TODS)* 4.4 (1979), p. 455–469.

[31] Bruno Marnette. “Generalized schema-mappings: from termination to tractability”. In: *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2009, p. 13–22.

[32] Michael Meier. *On the termination of the chase algorithm*. Springer, 2010.

[33] Michael Meier, Michael Schmidt en Georg Lausen. “On chase termination beyond stratification”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), p. 970–981.

[34] Michael Meier, Michael Schmidt en Georg Lausen. “Stop the chase”. In: *arXiv preprint arXiv:0901.3984* (2009).

[35] Renée J Miller e.a. “The Clio project: managing heterogeneity”. In: *SIgMOD Record* 30.1 (2001), p. 78–83.

[36] Dan Olteanu, Jiewen Huang en Christoph Koch. “Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases”. In: *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE. 2009, p. 640–651.

[37] Adrian Onet. “The chase procedure and its applications in data exchange”. In: *Dagstuhl Follow-Ups* 5 (2013).

[38] Adrian Constantin Onet. “The chase procedure and its applications”. Proefschrift. Concordia University, 2012.

[39] Lucian Popa. “Object/relational query optimization with chase and backchase”. In: *IRCS Technical Reports Series* (2001), p. 19.

[40] Riccardo Rosati. “On the finite controllability of conjunctive query answering in databases under open-world assumption”. In: *Journal of Computer and System Sciences* 77.3 (2011), p. 572–594.

[41] Jeffrey D Ullman. “Information integration using logical views”. In: *Database Theory—ICDT'97*. Springer, 1997, p. 19–40.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
De chase

Richting: **master in de informatica-databases**
Jaar: **2015**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Creemers, Mathijs

Datum: **22/06/2015**