Towards a theory for Active XML queries

Niels Van Och

promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Eindverhandeling voorgedragen tot het bekomen van de graad
master in de informatica databases

universiteit hasselt

Abstract

In dit werk bekijken we de theorie achter *actieve databases*, databases waarin elementen kunnen voorkomen die uitvoerbaar zijn.

Eerst stellen we een model op voor deze databases. Hiervoor beginnen we met een eenvoudig basismodel waarop we de concepten van een database en een query grondig onderzoeken.

Vervolgens onderzoeken we een aantal mogelijkheden om dit model uit te breiden om zo tot meer uitdrukkingskracht te komen, en werken een van deze mogelijkheden uit.

Tenslotte zoeken we een querytaal voor dit model die compleet is, die met andere woorden alle mogelijke queries kan uitdrukken.

Voorwoord

Toen ik de opleiding informatica aan de UHasselt (toen nog het Limburgs Universitair Centrum) aanvatte bestond mijn kennis van het vakgebied eigenlijk vooral uit de praktische ervaring die ik opgedaan had in het gebruik van computers.

Met de jaren werd het me echter duidelijk dat ik eerder een theoreticus van aard ben dan een praktijkmens. Met die realisatie lag mijn keuze voor de afstudeerrichting databases snel vast, en ben ik ook aan de ambitieuze taak begonnen een thesis te schrijven die bijzonder theoretisch van aard is.

Ik bedank in dit kader graag de professoren en assistenten waarvan ik gedurende mijn opleiding les gehad heb, en in het bijzonder mijn promotor prof. dr. Jan Van den Bussche. Zijn enthousiasme mag een voorbeeld wezen voor elke academicus!

Verder verdienen ook mijn ouders een vermelding. Zij gaven me de kans om te studeren, waarvoor ik ze eeuwig dankbaar zal zijn. En tenslotte ook een dankwoord voor mijn vrienden. Hun steun en gezelschap hebben de afgelopen jaren veruit de beste van mijn leven gemaakt.

Inhoudsopgave

1 Inleiding **1**
- 1.1 Inleiding . . . 1
- 1.2 Voorafgaand werk . . . 2
  - 1.2.1 Active XML . . . 2
  - 1.2.2 Meta-queries . . . 5

2 Basisconcepten **7**
- 2.1 Inleiding . . . 7
- 2.2 Database . . . 8
- 2.3 Query . . . 9
  - 2.3.1 Typering . . . 10
  - 2.3.2 Genericiteit . . . 10
  - 2.3.3 Berekenbaarheid . . . 12
- 2.4 Querytalen . . . 15
  - 2.4.1 Relationele algebra . . . 15
  - 2.4.2 Algebra + while . . . 16
  - 2.4.3 While$_N$ . . . 17
  - 2.4.4 While$_{new}$ . . . 19
  - 2.4.5 Bewijs van compleetheid . . . 20

3 Actieve databases: eerste model **25**
- 3.1 Inleiding . . . 25
- 3.2 Services . . . 26
- 3.3 Actieve databases . . . 28
- 3.4 Actieve queries . . . 29
  - 3.4.1 Voorbeeld . . . 29
  - 3.4.2 Theorie . . . 30
  - 3.4.3 Conclusie . . . 32

4 Actieve databases: tweede model **33**
4.1 Aanpassingen 33
4.2 Actieve databases 34
4.2.1 Schema's 34
4.2.2 Instanties 34
4.3 Actieve queries 35
4.3.1 Typering 37
4.3.2 Genericiteit 37
4.3.3 Berekenbaarheid 43

5 Actieve databases: querytheorie **46**
5.1 Inleiding 46
5.2 while$_{new}$ + *call* 47
5.3 Compleetheid van while$_{new}$ + *call* 47
5.3.1 Compleetheid van while$_N$ + *call* 48
5.3.2 Compleetheid van while$_{new}$ + *call* 48
5.4 *parallel-call* simuleren 50
5.4.1 Canonieke encoderingen 50
5.4.2 Simulatie van *parallel-call* 52

Hoofdstuk 1

Inleiding

In dit hoofdstuk leggen we het idee en de motivatie achter actieve databases uit.

1.1 Inleiding

In dit werk behandelen we *actieve databases*. Conceptueel zijn dit databases waarin elementen voorkomen die uitgevoerd kunnen worden als een functie. In tegenstelling tot de data-elementen uit klassieke databases spelen deze elementen dus een veel actievere rol, wat meteen de titel verklaart. We zullen deze actieve elementen *services* noemen.

De theorie achter actieve databases is al deels uitgewerkt, maar hetgeen waar we in dit werk naartoe willen is dat nog niet: een querytheorie die ons toelaat om op een theoretisch ondersteunde manier bevragingen te formuleren waarin de services kunnen gebruikt worden, wat dit ook moge betekenen. We zoeken dus achter een querytaal die compleet is voor actieve databases, met andere woorden een taal die alle mogelijke queries kan uitdrukken.

Het vinden van zo'n taal verwezenlijken we in drie stappen: eerst leggen we een reeks basisnotaties en -definities vast voor klassieke databases, dan zoeken we een geschikt model voor actieve databases, en tenslotte proberen we een complete querytaal te formuleren.

1.2 Voorafgaand werk

Eerst en vooral moeten we een duidelijk onderscheid maken tussen het concept van een actieve database uit 1970, en hetgene dat we nu gebruiken. Toen het relationele model nog niet zo lang bestond was er ook al de notie van een database "actief" te maken. Wat men toen echter bedoelde was dat een database management systeem niet enkel meer passief zou wachten op de gebruiker om queries uit te voeren, maar dat er ook regels konden opgesteld worden waarmee het systeem zelf een aantal operaties zou uitvoeren wanneer aan een bepaalde voorwaarde voldaan werd. Dit idee sloeg aan en is tegenwoordig zo alomtegenwoordig dat de term "actief" vergeten is: we kennen deze regels en operaties als triggers. Het huidige concept van actieve databases, waarbij elementen uit de database uitvoerbaar zijn, werd pas in het laatste decennium naar voor geschoven.

Dit werk is echter niet begonnen als een theorie voor actieve databases, maar als een querytheorie voor *Active XML*. Van dit uitgangspunt is het onderzoek dan geëvolueerd naar een algemenere vorm voor databases. De terminologie en ideeën zijn echter nog steeds sterk gerelateerd aan die van Active XML, dus we besteden er hier toch aandacht aan. Tenslotte zeggen we ook nog iets over meta-querying, een concept dat toch ook enige invloed heeft op dit werk.

1.2.1 Active XML

Het concept van Active XML werd pas in 2003 voorgesteld, en laat documenten eigenlijk toe om deel van hun XML-boom impliciet te definiëren door bepaalde webservices toe te voegen. Bij het bekijken van het document worden deze webservices dan aangeroepen, en wordt op basis van hun antwoord de boom opgesteld.

XML

De Extensible Markup Language ligt eigenlijk aan de basis van het moderne Internet, want het is het standaard formaat om data van alle vormen uit te wisselen. We gaan er hier van uit dat de lezer voldoende kennis heeft van het XML-model, en geven enkel een verwijzing naar [1] voor meer informatie.

Webservices

Webservices zijn in de laatste jaren steeds populairder aan het worden, en dankzij een gestandaardiseerd protocol -het Simple Object Access Protocol, kortweg SOAP[1]- al goed geïntergreerd in het huidige World Wide Web.

Een webservice is eigenlijk niets meer dan een programma dat toegankelijk is via een netwerk, typisch het Internet. Ze neemt gegevens in XML-vorm als parameters, en geeft gegevens in XML-vorm terug als resultaat. Een webservice wordt typisch beschreven aan de hand van de Web Services Description Language (WSDL), een taal die zelf op XML gebaseerd is, en in samenwerking met XML schema's de services formeel kan beschrijven.

Active XML

De integratie van XML en webservices ligt eigenlijk voor hand: waarom zouden we ons beperken tot statische XML-documenten aan de ene kant, en dynamische webservices aan de andere? Het zou gemakkelijk zijn als de webservices geïntegreerd konden worden in XML-documenten, zodat de inhoud ervan dynamisch kan veranderen.

Dit is precies het doel dat Active XML bereikt. Een Active XML-document is een standaard XML-document waarin ook de mogelijkheid gegeven wordt om data intensioneel te geven, dankzij het gebruik van aanroepen naar webservices. Dit in tegenstelling tot de klassieke dataknopen, die de data steeds expliciet geven.

Wat precies de constructie en semantiek van Active XML-documenten moet zijn blijft nog enigzins onder discussie. Over het model is grotendeels consensus bereikt, maar een belangrijk punt blijft nog wat er moet gebeuren met het resultaat van een webservice-aanroep. Hier zijn twee mogelijkheden voor:

1. De resultaatboom wordt in plaats van de aanroep naar de webservice geplaatst.
2. De resultaatboom wordt toegevoegd als sibling van de aanroep naar de webservice.

[1]Het acroniem wordt tegenwoordig niet meer gebruikt door het W3C, maar wordt nog vaak gebruikt op het web

Beide mogelijkheden hebben hun voor- en nadelen, en er bestaat nog geen consensus over het probleem.

Een laatste punt waarrond nog discussie bestaat is wanneer de aanroepen naar de webservices gematerialiseerd moeten worden. Ook hier bestaan twee mogelijkheden:

1. Laat de client beslissen wanneer de aanroepen moeten gebeuren. Dit zal typisch zijn wanneer het document opgevraagd wordt door een gebruiker of een ander proces. Deze manier van werken noemt men de pull-modus.

2. Laat de beslissing aan de servers die de webservices hosten. Dit kan bijvoorbeeld periodiek zijn, of wanneer er nieuwe gegevens zijn, of zelfs handmatig door een beheerder gebeuren. Deze methode staat bekend als de push-modus.

Tenslotte geven we nog een aantal verwijzingen. Ten eerste verwijzen we naar [2], de thesis die de directe aanleiding was tot dit werk. Verder geven we verwijzingen naar [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] en [13]. Dit zijn de belangrijkste onderzoeksresultaten rond Active XML op dit moment.

Queriën van Active XML

Rond het queriën van XML-documenten bestaat er al een hoop theorie en praktijk - talen zoals XQuery en XPath worden in veel verschillende gebieden gebruikt. En natuurlijk willen we deze theorie ook kunnen toepassen op Active XML-documenten.

Door de dynamische aard van deze documenten ontstaan hier echter enkele problemen, vooral qua performantie. Denk bijvoorbeeld aan het materialiseren van services: een naïeve manier van queriën zou voor de evaluatie gewoon alle services materialiseren, maar dit is uiteraard niet performant. Een betere methode zou zijn om tijdens het uitvoeren van de query te kijken welke services aangeroepen moeten worden, en dit dan pas te doen. Maar ook dit levert problemen op, want het betekent dat de query processor steeds moet wachten op de webservices om verder te gaan met de uitvoering.

Active XML en dit werk

Zoals we in het begin van dit hoofdstuk al aanhaalden, gaat dit werk niet echt over Active XML, maar over actieve databases. Daarom willen we het verschil van focus tussen de thesis van Tamara Vos en dit werk even naderbij belichten.

In [2] werd er onderzocht wat het uitvoeren van queries op Active XML-documenten precies inhoudt, en hoe dit praktisch kan gebeuren (evenwel zonder een concrete uitwerking te maken van zo'n querysysteem). Het belangrijkste probleem bij het queriën van Active XML-documenten is dat de documenten mogelijk oneindig diepe bomen kunnen voorstellen. Dit wordt in [2] opgelost door te zorgen dat de queries steeds positief en monotoon zijn, wat betekent dat de documenten gewoon gesatureerd kunnen worden (door aanroepen van webservices steeds opnieuw te doen).

In dit werk proberen we de querytheorie te veralgemenen zodat ze toegepast kan worden op databases in het relationele model. We proberen een model te vinden dat dit toelaat, en kijken naar een querytaal die compleet is onder dit model.

De theorie die we in dit werk opstellen is dus niet meer direct toepasbaar op het Active XML-model. De basisprincipes en ideeën blijven echter nog steeds dezelfde, dus dit onderzoek zou ook voor Active XML interessante resultaten kunnen opleveren.

1.2.2 Meta-queries

Wat we in dit werk beschrijven is eigenlijk een vorm van meta-querying, ook al is het uitgangspunt vanwaar we vertrekken behoorlijk anders dan dat van de klassieke theorie hierrond. Daarom besteden we hier weinig aandacht aan het bestaande onderzoek errond, en geven slechts een korte schets en verwijzingen.

Meta-querying is een concept dat in recente jaren steeds meer aandacht heeft gekregen binnen de databasewereld. Het queriën van databases die zelf queries kunnen bevatten lijkt op het eerste zicht weinig nuttig, maar kan veel praktische applicaties hebben. We denken hierbij bijvoorbeeld aan views op een database, die niets meer zijn dan opgeslagen queries. Andere voorbeelden zijn stored procedures of system logs, die zelfs grote hoeveelheden queries

bevatten. Over dit soort data kunnen we weer andere vragen gaan stellen. Het probleem ligt hier natuurlijk in het feit dat deze queries gewoon opgeslagen zijn als lange strings, wat het moeilijk maakt ze te queriën.

Voor meer informatie over meta-queries verwijzen we naar [14], [15] en [16].

Hoofdstuk 2

Basisconcepten

In dit hoofdstuk introduceren we een aantal basisconcepten en hun notatie.

2.1 Inleiding

We beginnen dit werk met de definitie van een aantal basisconcepten uit databasetheorie. Deze concepten worden doorheen dit werk veelvuldig gebruikt. De presentatie en veel van de notatie ervan steunt op *Foundations of Databases* van Abiteboul, Hull en Vianu[17], een van de standaardwerken binnen het onderzoeksgebied. We wijken echter op sommige vlakken af van dit werk, vooral wat notatie betreft.

In het bijzonder behandelen we in dit hoofdstuk de definitie van databases en hun schema's en de definitie van een query. Verder behandelen we een aantal gangbare querytalen, en bewijzen een aantal resultaten rond de compleetheid ervan.

We geven in de volgende sectie ook telkens een incrementeel voorbeeld van een kleine database, zodat de opbouw van de theorie duidelijk is.

2.2 Database

Attributen

We onderstellen een aftelbare verzameling van attributen **att**, die de attributen voorstellen die in de relaties van een database kunnen voorkomen.

Voor elk attribuut $A \in \mathbf{att}$ onderstellen we verder een verzameling $\mathbf{dom}_A$ van abstracte waarden. Intuïtief zijn dit de waarden die dit attribuut kan aannemen. In het vervolg nemen we echter aan dat elk attribuut hetzelfde domein heeft, een aanname die geen belangrijke invloed heeft op de algemeenheid van de beweringen in dit werk. We noteren dit domein met **dom**.

Bijvoorbeeld: $\mathbf{att} = \{A, B, C\}$ met $\mathbf{dom} = \mathbb{N}$.

Schema's

Vervolgens definiëren we schema's: een relatieschema $\mathcal{R}$ is een eindige verzameling van attributen, en een databaseschema $\mathcal{D}$ is een eindige verzameling relatienamen, samen met een functie *sort*. Deze functie beeldt de relatienamen af op relatieschema's. Intuïtief beschrijven deze schema's de vorm waaraan een relatie of een database moet voldoen.

Bijvoorbeeld: als $\mathcal{R} = \{A, B\}$ en $\mathcal{S} = \{C\}$ twee relatieschema's zijn, is $\mathcal{D} = (\{R, S\}, sort)$ met $sort(R) = \mathcal{R}$ en $sort(S) = \mathcal{S}$ een databaseschema.

Tupels

Als we nu een relatieschema $\mathcal{R}$ onderstellen, kunnen we een tupel over $\mathcal{R}$ definiëren: dit is een functie $t : \mathcal{R} \longrightarrow \bigcup_{A \in \mathcal{R}} \mathbf{dom}_A$ waarbij $t(A) \in \mathbf{dom}_A$, en dit voor elke $A \in \mathcal{R}$.

We voeren hier ook een extra notatie voor tupels in: een tupel t over $\{A_1, \ldots, A_n\}$ noteren we als $t = (A_1 : t(A_1), \ldots, A_n : t(A_n))$. We gebruiken deze notatie als alternatief voor de zwaardere verzamelingtheoretische notatie $t = \{(A_1, t(A_1)), \ldots, (A_n, t(A_n))\}$.

Bijvoorbeeld: $t_1 = \{A : 1, B : 2\}$ en $t_2 = \{A : 3, B : 3\}$ zijn tupels over $\mathcal{R}$, terwijl $t_3 = \{C : 2\}$ en $t_4 = \{C : 4\}$ tupels over $\mathcal{S}$ zijn.

Instanties

Nu is een relatie (ook wel een relatie instantie genoemd) over $\mathcal{R}$ een verzameling van tupels over $\mathcal{R}$. We noteren deze verzameling met r. De verzameling van alle relatie instanties over een schema $\mathcal{R}$ noteren we met $inst(\mathcal{R})$.

Tenslotte is een database (of database instantie) over een databaseschema $\mathcal{D}$ nu gedefinieerd als een functie I die elke relatienaam $R \in \mathcal{D}$ afbeeldt op een relatie r over $sort(\mathcal{R})$. De verzameling van alle database instanties over een schema $\mathcal{D}$ noteren we met $inst(\mathcal{D})$.

Bijvoorbeeld: $r_1 = \{t_1, t_2\}$ is een relatie over $\mathcal{R}$, en $r_2 = \{t_3, t_4\}$ is een relatie over $\mathcal{S}$. Een database over $\mathcal{D}$ is I waarbij $I(R) = r_1$ en $I(S) = r_2$.

De database instantie die we nu geconstrueerd hebben ziet er in de tabelvorm als volgt uit:

R

A	B
1	2
3	3

S

C
2
4

2.3 Query

Nu we de definitie van een database kennen, zijn we klaar om queries over deze databases te definiëren.

De definitie voor een query die we hier gebruiken is de volgende: voor een databaseschema $\mathcal{D}$ en een relatieschema $\mathcal{R}$ is een query q van type $\mathcal{D} \longrightarrow \mathcal{R}$ een partiële functie van $inst(\mathcal{D})$ naar $inst(\mathcal{R})$ die *generisch* en *berekenbaar* is. We stellen hier dus eigenlijk drie eisen aan een query, die we stuk voor stuk toelichten.

2.3.1 Typering

Ten eerste eisen we dat een query goed getypeerd is, dus dat de invoer- en uitvoerschema's ervan vastliggen. We laten wel toe dat de functie partieel is, dus dat het resultaat van de query niet op elke instantie gedefinieerd hoeft te zijn. Deze eis ligt voor de hand wanneer we naar querysystemen in de praktijk kijken.

2.3.2 Genericiteit

Dan is er de eis dat een query generisch moet zijn. Intuïtief betekent dit dat de query geen eigenschappen van de elementen in de database mag gebruiken, behalve de onderlinge relaties die worden gespecifieerd door de database zelf. Een voorbeeld van zo'n "verboden" eigenschap is de interne representatie van het element.

Op het eerste zicht lijkt deze eis misschien onnodig, omdat we onszelf een beperking opleggen die qua berekenbaarheid niet nodig is, maar wel een hele groep queries uitsluit. De eis van genericiteit is echter nodig om een van de belangrijkste principes van databases te vatten, namelijk dat van dataonafhankelijkheid: een database moet steeds een abstracte interface van haar data bieden, en de interne voorstelling ervan verbergen. Dit principe bestaat al sinds de ontwikkeling van de eerste databasemodellen, en er wordt niet van afgeweken.

Om de eis van genericiteit uit te drukken moeten we eerst het volgende definiëren: een permutatie ρ van een verzameling S is een bijectie van S naar zichzelf. Als we nu een permutatie ρ van **dom** hebben, definiëren we $\rho(r)$ als de puntsgewijze toepassing van ρ op alle elementen van r. Dit doen we als volgt: de toepassing van ρ op een tupel t over $\mathcal{R}$ is $\rho(t) = \rho \circ t$. De toepassing van ρ op een relatie instantie r is $\rho(r) = \{\rho(t) \mid t \in r\}$. Tenslotte is een permutatie van een database-instantie I gedefinieerd als $\rho \circ I$.

De eis van genericiteit drukken we nu uit met de volgende definitie:

Definitie 1. *Een query q van type $\mathcal{D} \longrightarrow \mathcal{R}$ noemen we* generisch *als voor elke instantie I over $\mathcal{D}$ en voor elke permutatie ρ van* ***dom*** *het volgende geldt:* $\rho(q(I)) = q(\rho(I))$.

We geven een voorbeeld van een query die niet generisch is: kies een totale orde op **dom**, en noteer deze als $<$. Neem nu $\mathcal{D} = \{U\}$ met $sort(U) = \{A\}$, en $\mathcal{R} = \{A\}$, en de query $q(I) := \{min(I(U))\}$, waarbij min het minimum is volgens de orde $<$. Beschouw nu volgende instantie I over $\mathcal{D}$ (waarbij $<$ de orde over de natuurlijke getallen is):

A
1
2
3

en de permutatie ρ die gedefinieerd is als de cyclische permutatie $(1, 2, 3)$, dus de permutatie die 1 afbeeldt op 2, 2 afbeeldt op 3 en 3 afbeeldt op 1.

Als we nu eerst q uitvoeren, en dan ρ (dus $\rho(q(I))$) krijgen we als resultaat

A
2

terwijl het resultaat van eerst ρ uitvoeren, en dan q (dus $q(\rho(I))$) het volgende is:

A
1

Gezien deze twee resultaten verschillen, hebben we een tegenvoorbeeld voor de genericiteit van q. Dit klopt met de intuïtie, want q maakt gebruik van een eigenschap van de elementen die niet gegeven wordt door een relatie in de database, namelijk de onderlinge orde ervan. Dit is iets wat we niet toelaten.

Merk op dat in de praktijk een databasesysteem dit soort queries uiteraard wel toelaat. Vanuit een theoretisch standpunt willen we ze echter steeds uitsluiten.

2.3.3 Berekenbaarheid

De eis dat een query berekenbaar is, ligt meer voor de hand. De uiteindelijke bedoeling van een query is immers steeds ze uit te voeren. Deze eis is echter moeilijker uit te drukken, en vraagt wat voorbereidend werk.

In de klassieke notie van berekenbaarheid werken we steeds met strings als invoer en uitvoer: de Turing Machines die een probleem berekenen werken van $\Sigma^* \longrightarrow \Sigma^*$. De queries waarvan we de berekenbaarheid hier willen vastleggen werken echter op database instanties en relatie instanties. We hebben dus een bepaalde encodering nodig die een TM toelaat om met deze instanties te werken.

Berekenbaarheid ten opzichte van een encodering

We zullen deze encoderingen eerst definiëren. Onderstel hiervoor een enumeratie α van **dom**. Dit is niets meer dan een opsomming van de elementen van **dom** in een bepaalde volgorde. Nu is de encodering van **dom** tegenover α een injectieve functie die het i-de element van **dom** (volgens α) afbeeldt op de binaire representatie van i. We noteren deze encodering met enc_α. Zo'n encodering stelt de elementen uit het domein dus voor als binaire getallen. Deze getallen kunnen we uiteraard gemakkelijk als strings over $\{0, 1\}$ beschouwen.

Vervolgens definiëren we de encodering van tupels en instanties. We spreken af dat een encodering de databases steeds afbeeldt op de verzameling $\Sigma = \{0\,1\,\{\,\}\,,\,(\,)\}$.

- Encodering van tupels: $enc_\alpha((A : x, B : y, \ldots)) = (enc_\alpha(x), enc_\alpha(y), \ldots)$. Hierbij spreken we af dat we steeds de lexicografische volgorde van de attribuutnamen gebruiken.
- Encodering van relaties: $enc_\alpha(r) = \{enc_\alpha(t_1), enc_\alpha(t_2), \ldots\}$ voor alle tupels t_i in r. We beelden de tupels steeds in lexicografische volgorde af.
- Encodering van databases: $enc_\alpha(I) = \{enc_\alpha(r_1), enc_\alpha(r_2), \ldots\}$ voor alle relaties r_i in I. We beelden de relaties steeds af in lexicografische volgorde van de relatienamen.

Deze regels laten ons dus toe om hele database instanties voor te stellen door als strings.

We geven een voorbeeld van een encodering van een database: onderstel een database met een binaire en een unaire relatie, die er als volgt uitzien:

R

A
1
2

S

B	C
2	3
4	5

Een encodering van deze database tegenover de enumeratie $\alpha = 1, 2, 3, 4, 5$ zou dan de volgende string kunnen zijn: $\{\{(1, 10)\}, \{(10, 11), (100, 101)\}\}$.

We kunnen nu de definitie van berekenbaarheid ten opzichte van een encodering geven:

Definitie 2. *Een query q is berekenbaar ten opzichte van een encodering enc_α als er een Turing Machine M bestaat zodat voor elke instantie $I(\mathcal{D})$ geldt dat:*

- *als $q(I(\mathcal{D}))$ niet gedefinieerd is, M niet stopt op invoer $enc_\alpha(I(\mathcal{D}))$, en*
- *als $q(I(\mathcal{D}))$ wel gedefinieerd is, M stopt op invoer $enc_\alpha(I(\mathcal{D}))$ met $enc_\alpha(q(I(\mathcal{D})))$ op zijn (uitvoer)band.*

Deze definitie is echter nog niet voldoende, omdat ze afhangt van de encodering die gebruikt wordt. Echter, doordat we al weten dat de queries steeds generisch moeten zijn, kunnen we aantonen dat de gebruikte encodering onbelangrijk is.

Algemene berekenbaarheid

Eigenschap 1. *Zij $q : inst(\mathcal{D}) \longrightarrow inst(\mathcal{R})$ een generische partiële functie. Dan is q berekenbaar door een TM M tegenover encodering $enc_\alpha \Leftrightarrow q$ is berekenbaar door dezelfde M tegenover alle mogelijke encoderingen.*

Bewijs. We moeten alleen de $\Rightarrow$-implicatie bewijzen, de andere richting is triviaal. We moeten dus bewijzen dat gegeven een encodering enc_α waarvoor q berekenbaar is door M, geldt dat q berekenbaar is door dezelfde M onder elke encodering. Formeel is dit dus: $\forall\beta : enc_\beta(q(I)) = M(enc_\beta(I))$.

Beschouw nu een willekeurige encodering enc_β, en bekijk de samenstelling $\rho = enc_\alpha^{-1} \circ enc_\beta$. Merk op dat we ρ kunnen zien als een permutatie van **dom**.

Nu is $M(enc_\beta(D)) = M(enc_\alpha \circ \rho(D))$. Gezien ρ een permutatie is en q generisch is, is dit gelijk aan $M(\rho(enc_\alpha(D))) = enc_\alpha(q(\rho(D)))$. Dit is nu gelijk aan $enc_\alpha(\rho(q(D))) = enc_\alpha \circ \rho(q(D)) = enc_\beta(q(D))$.

□

Dankzij deze eigenschap kunnen we nu de definitie van berekenbaarheid veralgemenen:

Definitie 3. *Een query q is berekenbaar als er een Turing Machine M bestaat zodat voor elke instantie $I(\mathcal{D})$ en elke encodering enc_α geldt dat:*

- *als $q(I(\mathcal{D}))$ niet gedefinieerd is, M niet stopt op invoer $enc_\alpha(I(\mathcal{D}))$, en*
- *als $q(I(\mathcal{D}))$ wel gedefinieerd is, M stopt op invoer $enc_\alpha(I(\mathcal{D}))$ met $enc_\alpha(q(I(\mathcal{D})))$ op zijn (uitvoer)band.*

In deze definitie is de berekenbaarheid dus onafhankelijk van de gebruikte encodering. We kunnen nu echter nog een sterkere eigenschap aantonen:

Eigenschap 2. *Zij $q : inst(\mathcal{D}) \longrightarrow inst(\mathcal{R})$ een partiële functie berekenbaar door eenzelfde TM tegenover alle mogelijke encoderingen. Dan is q generisch.*

Bewijs. Neem een willekeurige encodering enc_α en een willekeurig isomorfisme ρ. Beschouw nu $enc'_\alpha = enc_\alpha \circ \rho$. Dit is ook een encodering, dus volgens het gegeven weten we dat $enc'_\alpha(q(I)) = M(enc'_\alpha(I)) = M(enc_\alpha(\rho(I))) = enc_\alpha(q(\rho(I)))$.

Nu merken we nog op dat enc_α steeds injectief is, dus we mogen ze schrappen uit de vorige gelijkheid. We kunnen dus stellen dat $\rho(q(I)) = q(\rho(I))$.

□

2.4 Querytalen

Nu we gedefinieerd hebben wat een query precies is, bekijken we ook een aantal manieren om queries uit te drukken, de zogenaamde querytalen. We zijn in het kader van dit werk vooral geïnteresseerd in querytalen die *compleet* zijn, met andere woorden talen die elke mogelijke query (binnen onze definitie) kunnen uitdrukken.

In deze sectie geven we een overzicht van een aantal querytalen, steeds opbouwend in uitdrukkingskracht, tot we een complete taal bekomen. Van deze taal bewijzen we ook de compleetheid.

Een belangrijke opmerking die we hier maken is het feit dat alle talen die we hier geven queries uitdrukken die wel degelijk aan onze definitie van query voldoen, met andere woorden ze drukken generische en berekenbare functies uit. Dit wordt vooral belangrijk in Hoofdstuk 5, waar we onze theorie sterk zullen baseren op deze talen.

2.4.1 Relationele algebra

De bekendste (en meest gebruikte) taal voor queries is ongetwijfeld de relationele algebra, zij het misschien in een andere vorm. Moderne database management systemen (die op het relationele model gebaseerd zijn) nemen bijna allemaal queries in Structured Query Language (SQL) als invoer, vertalen deze naar de relationele algebra, en voeren ze in die vorm uit.

Naast uitdrukkingen in de relationele algebra kunnen we natuurlijk ook programma's erin beschouwen, waar relatievariabelen het resultaat van algebra-uitdrukkingen toegekend krijgen. We gaan er hier van uit dat de lezer de syntax en semantiek van relationele algebra-programma's kent. We gebruiken in

dit werk ook sporadisch uitdrukkingen in de relationele calculus. Ook hier gaan we ervan uit dat de lezen deze begrijpt, en weet dat ze equivalent zijn met relationele algebra-uitdrukkingen.

De uitdrukkingskracht van de relationele algebra is aanzienlijk, maar er zijn toch een heel aantal queries die niet uitgedrukt kunnen worden. Een simpel voorbeeld is de transitieve sluiting: wanneer we een binaire relatie beschouwen als de bogen van een graaf, geeft de query de binaire relatie die alle paren van knopen geeft waartussen een pad bestaat.

2.4.2 Algebra + while

Een eerste toevoeging aan de relationele algebra is de mogelijkheid om lussen toe te voegen. Deze zijn van de vorm:

```
while change do
  begin
    [aantal algebra-statements]
  end
```

De semantiek hiervan is de volgende: zolang een van de statements in de lus een verandering veroorzaakt in een relatie, blijft de lus herhaald worden.[1]

Deze lussen voegen genoeg uitdrukkingskracht toe aan de relationele algebra om de transitieve sluiting-query uit te drukken, en wel op de volgende manier: onderstel dat G de binaire relatie is die de graaf voorstelt, dan slaat volgend programma de transitieve sluiting van G op in T:

$T := G$;
while change do
 begin
 $T := T \cup \pi_{AB}(\delta_{B \to C}(T) \bowtie \delta_{A \to C}(G))$;
 end

Hoe krachtig de relationele algebra + while echter ook is, er zijn nog steeds queries die we niet kunnen uitdrukken in deze taal, bijvoorbeeld de simpele query die vraagt of het aantal elementen in een relatie even is.

[1] Er zijn ook varianten waarbij een expliciete stopconditie wordt gegeven, maar deze zijn equivalent.

2.4.3 While$_N$

Het grootste probleem met de relationele algebra + while is dat ze geen mogelijkheid heeft om berekeningen buiten de database te doen, wat de mogelijke queries drastisch beperkt. De volgende uitbreiding lost dit probleem op: in while$_N$ voegen we functionaliteit toe zodat de taal volledige berekeningskracht heeft buiten de database.

Wat we hiervoor moeten toevoegen is het volgende:

- Integer variabelen
- De constante 0 voor integer variabelen
- De operaties $increment(i)$ en $decrement(i)$, met i een integer variabele
- Een conditionele constructie if $i = 0$ then ... else ...
- Een lus van de vorm while $i > 0$ do ...

De semantiek is hier recht door zee: de gebruikte integer variabelen worden op 0 geïnitialiseerd en de while change-lus neemt geen veranderingen van integer variabelen in rekening.

De toevoeging van deze elementen laat de taal toe om computationeel compleet te zijn op de integers. Immers, een taal waarin programma's integer variabelen kunnen aanmaken, verhogen en verlagen is equivalent met de zogenaamde *counter machines*, waarvan de computationele compleetheid al lang bekend is. Voor meer informatie hierover verwijzen we naar [18].

Compleetheid voor integers is een mooie uitbreiding, maar spijtig genoeg is de taal nog steeds niet compleet voor de queries. De bovenvermelde query die test of het aantal elementen in een relatie even is, kan nog steeds niet worden uitgedrukt.

Ook al is deze taal niet compleet, ze is wel een belangrijke tussenstap in de zoektocht naar een taal die dat wel is. Hiervoor geven we volgende eigenschap:

Eigenschap 3. *While$_N$ is compleet voor databases waarin een relatie bestaat die een totale orde op het effectief domein geeft.*

Als we dus een enumeratie kunnen geven van de elementen van een database instantie, is het mogelijk om elke mogelijke query op deze instantie uit te drukken met een while$_N$-programma. In de rest van dit hoofdstuk gebruiken we de termen "totale orde", "opvolgersrelatie" en "enumeratie" enigzins door elkaar. Deze termen drukken echter allemaal hetzelfde idee aan (een opsomming van de elementen in het effectief domein), maar in een andere encodering. Uit de context blijkt echter steeds wat we bedoelen.

Bewijs. Beschouw een query q van type $\mathcal{D} \longrightarrow \mathcal{R}$. Neem een instantie I van $\mathcal{D}$, en laat α de enumeratie zijn van de elementen uit I, definieerd door de orde die gegeven is.

Dankzij de definitie van een query weten we dat er een Turing Machine M bestaat die de q berekent, met andere woorden op invoer $enc_\alpha(I)$ de uitvoer $enc_\alpha(q(I))$ geeft (wanneer $q(I)$ gedefinieerd is), en dit voor elke encodering.

Gezien while$_N$ complete berekenbaarheid biedt op integers, willen we I nu encoderen als een integer in plaats van een string. Dit kan gemakkelijk, want elk woord over een eindig alfabet met n letters kunnen we zien als een getal in basis n. Laat voor elke instantie J nu $enc^*_\alpha(J)$ de integer-encodering van J zijn die we bekomen door $enc_\alpha(J)$ te zien als een integer.

Nu bestaat er steeds een berekenbare functie f_q over de integers zodat wanneer $q(I)$ gedefinieerd is, $f_q(enc^*_\alpha(I)) = enc^*_\alpha(q(I))$. En gezien while$_N$ elke berekenbare functie over de integers kan uitvoeren, bestaat er een programma w_q dat f_q berekent.

Nu ziet het uiteindelijke while$_N$-programma dat q berekent er als volgt uit:

1. Bereken $enc^*_\alpha(I)$
2. Bereken $f_q(enc^*_\alpha(I)) = enc^*_\alpha(q(I))$
3. Bereken $q(I)$ uit $enc^*_\alpha(q(I))$

In het voorgaande toonden we al aan dat stap 2 mogelijk is. Van de precieze constructie van stap 1 en 3 geven we hier alleen de intuïtie, gezien deze lang, technisch, en grotendeels irrelevant is. Om de encodering van I te simuleren merken we op dat er een maximum aan posities op de tape van de TM die f berekent gebruikt gaan worden. Als we dus een k groot genoeg

kiezen kunnen we met k-tupels elke mogelijke tape die voorkomt simuleren (gezien we met een eindig alfabet werken). Om de uiteindelijke string te decoderen moeten we eigenlijk dit proces gewoon omkeren.

□

2.4.4 While$_{new}$

Om nu ook compleetheid voor niet-geordende databases te bekomen moeten we nog een laatste element toevoegen. We doen dit echter terug bij algebra + while, en laten dus de integer variabelen terug weg.

We voegen het aan relationele algebra + while de mogelijkheid toe om nieuwe waarden te creëren, en wel op de volgende manier:

- Er is een nieuwe instructie $R := new(S)$, waarbij R en S relationele variabelen zijn, en $ariteit(R) = ariteit(S) + 1$
- De lus is van de vorm while R do begin ... end

De semantiek is de volgende: de *new*-operator voegt aan elk tupel in S een waarde (uit het domein) toe die nergens voorkomt in zowel de input als de huidige waarden in het programma. De lus wordt uitgevoerd zolang de variabele R niet leeg is.

De oplettende lezer zal al meteen opmerken dat de *new*-operator niet-deterministisch is, gezien we niet weten welke nieuwe waarden toegevoegd zullen worden. Dit maakt natuurlijk dat queries die we in deze taal stellen mogelijk niet meer voldoen aan onze definitie, gezien we eisen dat het antwoord van een query op een bepaalde invoer uniek is.

Om dit probleem te omzeilen beschouwen we enkel een bepaalde klasse van while$_{new}$-programma's, namelijk degene die *well-behaved* zijn. We noemen zo'n programma well-behaved als de uitvoerrelatie ervan nooit waarden bevat die geïntroduceerd zijn door een *new*-operatie. Dit is een eigenschap waarvoor we syntactische regels kunnen geven, en die dus getest kan worden.

2.4.5 Bewijs van compleetheid

Tenslotte bewijzen we nu dat de taal van well-behaved while$_{new}$-programma's compleet is, dus dat ze elke mogelijke query kan uitdrukken. Hiervoor moeten we eerst formeel bewijzen dat de taal wel degelijk queries kan uitdrukken. Dit wordt geformaliseerd in volgende eigenschap:

Eigenschap 4. *Elk well-behaved while$_{new}$-programma met invoerschema $\mathcal{D}$ en uitvoerschema $\mathcal{R}$ drukt een query uit van het type $\mathcal{D} \longrightarrow \mathcal{R}$.*

Bewijs. We moeten bewijzen dat well-behaved while$_{new}$-programma's functies definiëren van $\mathcal{D}$ naar $\mathcal{R}$, met andere woorden dat hun uitvoer steeds deterministisch bepaald is.

Neem hiervoor een well-behaved while$_{new}$-programma w met invoerschema $\mathcal{D}$ en uitvoerschema $\mathcal{R}$, en laat r en r' twee mogelijke instanties van $\mathcal{R}$ zijn na de uitvoering van w op een instantie I van type $\mathcal{D}$. Nu weten we dat er een isomorfisme ρ bestaat tussen r en r' dat de identiteit is op de waarden in I en constanten die voorkomen in w. En gezien w well-behaved is, zullen er in de uitvoerrelatie alleen waarden voorkomen uit I en w. Dit kan alleen als ρ de identiteit is, en dus als $r = r'$. □

Nu zijn we klaar om aan te tonen dat well-behaved while$_{new}$-programma's alle queries kunnen uitdrukken. Het idee achter dit bewijs is eenvoudig: we herinneren ons dat while$_N$ compleet is op geordende databases, met andere woorden dat er voor elke query q een while$_N$-programma bestaat dat, gegeven een opvolgersrelatie van de waarden in de invoer, q berekent. Als we dus zo'n enumeratie kunnen construeren voor een gegeven invoer (en deze voorstellen in de database) zouden we dit while$_N$-programma kunnen gebruiken om de query te berekenen. Nu is het door de eis van genericiteit duidelijk onmogelijk om zo één enumeratie te construeren, maar wat wel gaat is het construeren van alle mogelijke enumeraties. Als we dan op al de enumeraties het while$_N$-programma uitvoeren, krijgen we door de genericiteit toch telkens hetzelfde resultaat.

Voor we aan het uiteindelijke bewijs komen, moeten we eerst wat voorbereidend werk doen.

Enumeraties

We tonen eerst aan hoe een while$_{new}$-programma alle enumeraties van een instantie kan genereren. Laat in de volgende sectie I een instantie zijn van $\mathcal{D}$.

De relatie $\overline{succ}$ Beschouw de verzameling $Successor$ van alle binaire relaties die een opvolgersrelatie[2] over de waarden die voorkomen in I definiëren. We kunnen alle mogelijke enumeraties in $Successor$ voorstellen met een ternaire relatie door achter alle tupels van een opvolgersrelatie S een nieuw teken δ_S te plaatsen. Formeel wordt dit $\overline{succ} = \bigcup_{S \in Successor} I \times \{\delta_S\}$, waarbij $\{\delta_S \mid S \in Successor\}$.

We geven een voorbeeld:

a	b
a	c
c	a

Voor deze relatie bekomen we de volgende $\overline{succ}$:

a	b	δ_1
b	c	δ_1
a	c	δ_2
c	b	δ_2
b	a	δ_3
a	c	δ_3
b	c	δ_4
c	a	δ_4
c	a	δ_5
a	b	δ_5
c	b	δ_6
b	a	δ_6

[2]Dit is een relatie die een enumeratie van de elementen encodeert op volgende manier:

a	b
b	c
c	d
...	...

$\overline{succ}$ **berekenen** We tonen nu aan dat er een while$_{new}$-programma bestaat dat voor elke I de relatie $\overline{succ}(I)$ berekent.

Het is duidelijk dat er een while$_{new}$-programma bestaat dat een unaire relatie D produceert met daarin alle waarden van I. Het volgende while$_{new}$-programma geeft nu $\overline{succ}$ terug op basis van D:

$\overline{succ} := new(\sigma_{1\neq 2}(D \times D))$;
$\Delta := \{(a, b, \delta) \mid b$ komt niet voor in de opvolgersrelatie die overeenkomt met δ en het laatste element van δ is $a\}$;
while Δ *do*
begin
$S := new(\Delta)$;
$\overline{succ} := \{(x, y, \delta') \mid \exists \delta, x', y'[S(x', y', \delta, \delta') \wedge \overline{succ}(x, y, \delta)] \vee \exists \delta S(x, y, \delta, \delta')\}$;
$\Delta := \{(a, b, \delta) \mid b$ komt niet voor in de opvolgersrelatie die overeenkomt met δ en het laatste element van δ is $a\}$;
end

De intuïtie van dit programma is dat we de enumeraties van deelverzamelingen van grootte 2 construeren, en deze uitbreiden naar grootte 3 etc, tot we aan de enumeraties van D komen.

while$_N$ $\Rightarrow$ **while**$_{new}$ Voor we aan de uiteindelijke constructie kunnen beginnen, hebben we nog een laatste eigenschap nodig:

Eigenschap 5. *Voor elk while$_N$-programma bestaat er een equivalent while$_{new}$-programma.*

Bewijs. Het enige wat we eigenlijk moeten aantonen is dat we integers kunnen voorstellen in while$_{new}$. Hiervoor maken we voor elke integer i die voorkomt in het while$_N$-programma een binaire relatie R_i aan. Als i nu de waarde n heeft, dan bevat R_i een opvolgersrelatie van n tupels.

Het is evident dat er while$_{new}$-programma's bestaan die $increment(i)$ en $decrement(i)$ simuleren. Deze programma's voegen gewoon een element bij in de keten, of halen het laatste element uit de keten weg.

□

We stellen bijvoorbeeld 0 voor door de lege relatie, en 1 door het singleton $\{(\delta_0, \delta_1)\}$.

Bewijs van compleetheid

Nu zijn we klaar om aan te tonen dat well-behaved while$_{new}$-programma's alle queries kunnen uitdrukken.

Eigenschap 6. *De taal van well-behaved while$_{new}$-programma's is compleet.*

Bewijs. Uit sectie 2.4.5 en eigenschap 5 weten we dat we een zeker well-behaved while$_{new}$-programma w_{succ} kunnen construeren dat een query q berekent door gebruik te maken van een opvolgersrelatie die gegeven is in de database.

We construeren nu een programma dat q berekent met behulp van $\overline{succ}$. Intuïtief zal dit programma w_{succ} in parallel simuleren, één keer voor elke mogelijke enumeratie gegeven door $\overline{succ}$.

Om dit te bekomen, breiden we elke relatie in w_{succ} uit met een extra attribuut, en initialiseren we deze relaties met $\bar{R} := R \times \pi_3(\overline{succ})$. Op deze manier duiden we elk tupel dus aan met de enumeratie waarmee het overeen komt.

Nu moeten we de instructies van w_{succ} aanpassen zodat we de verschillende berekeningen in parallel kunnen doen. Dit doen we als volgt:

- Elke toekenningsuitdrukking $R := \{\langle u \rangle \mid \phi(u)\}$ wordt vervangen door $\bar{R} := \{\langle u, \delta \rangle \mid \exists y \exists z\, \overline{succ}(y, z, \delta) \wedge \bar{\phi}(u, \delta)\}$
- De lussen zijn iets moeilijker: het kan immers zijn dat de lus niet voor elke encodering even lang duurt. Op een bepaald moment kunnen sommige parallelle berekeningen dus al klaar zijn, terwijl anderen nog lopen. We moeten dus elke toekenningsuitdrukking in twee stappen doen: eerst voeren we de toekenning uit in een hulpvariabele voor de δ's die nog "actief" zijn (die moeten we in een hulprelatie bijhouden), en daarna doen we de echte toekenning, maar alleen voor de nog lopende berekeningen.
- De new-uitdukkingen moeten niet aangepast worden, deze zal zich in de parallelle berekeningen nog correct gedragen.

Tenslotte voegen we nog een uitdrukking toe achter de laatste uitdrukking: $\pi_{1,\ldots,n}(\bar{S})$, waarbij S de uitvoerrelatie van w_{succ} is, en deze ariteit n heeft.

Nu geldt dat na de uitvoering van dit programma $\bar{S} = \bigcup_{\delta} w(\delta)(I) \times \{\delta\}$, waarbij δ de enumeratiemarkeringen zijn. We weten dat S het resultaat van q bevat, en gezien q generisch is zijn de tupels in $\bar{S}$ per δ gelijk aan dit resultaat.

□

Hoofdstuk 3

Actieve databases: eerste model

In dit hoofdstuk doen we een eerste poging tot het definiëren van een model voor actieve databases.

3.1 Inleiding

Nu kunnen we beginnen met het definiëren van een actieve database. Conceptueel is dit dus een database die elementen kan bevatten die actief zijn, wat betekent dat we ze kunnen uitvoeren als een functie. Deze elementen noemen we *services*.

Er zijn een groot aantal vrijheidsgraden bij het definiëren van onze theorie, en het is helemaal niet duidelijk welke de beste keuzes zijn (als we al van beste kunnen spreken). We zullen dan ook starten vanuit een vrij eenvoudig basismodel. Dit model evalueren we grondig, en we definiëren actieve databases en queries erop. Daarna zullen we het model uitbreiden. In deze paragraaf bekijken we al even een aantal van de richtingen waarin deze uitbreiding zou kunnen gebeuren.

De eerste vraag die we ons kunnen stellen is of het aantal (verschillende) services dat kan voorkomen in een database moet vastliggen of niet. We kunnen vooraf alle services definiëren, of we kunnen een willekeurig aantal services toelaten. Een bijkomende vraag is waar de definitie van een service moet opgeslagen worden. In het geval van een vast aantal lijkt het aangewezen

om deze op te slaan in het databaseschema, maar bij een willekeurig aantal kan dit waarschijnlijk beter op instantieniveau.

Een gerelateerde vraag is of services gecreëerd mogen worden. Zoals er in klassieke databases object-creating queries bestaan die nieuwe elementen kunnen toevoegen aan een database, kunnen we ook toelaten dat services of queries nieuwe services gaan definiëren en toevoegen aan de (actieve) database.

Een laatste vraag waar we even bij stilstaan is wat de definitie van een service precies moet zijn. Nemen we gewoon de definitie van een query over, dus een functie die op invoer van een database een relatie teruggeeft? Of nemen we de analogie van Active XML, en laten we een service een nieuwe database teruggeven?

3.2 Services

In het eerste model kiezen we ervoor om het aantal services en hun semantiek vast te leggen, onafhankelijk van de database instantie. Hiervoor definiëren we een zogenaamd *service repository* **Rep**, ook wel *service schema* genoemd (deze termen zijn in overeenstemming met Active XML en klassieke databasetheorie, respectievelijk). Dit schema bestaat uit drie delen.

Ten eerste bevat het de namen van de services, die we uit **dom** nemen. We veronderstellen dus voor de eenvoud dat de servicenamen al in het domein opgenomen zijn, wat geen verlies van algemeenheid betekent voor de rest van onze theorie.

Ten tweede bevat het de *signature* van elke service. Een signature is een uitdrukking van de vorm databaseschema $\longrightarrow$ relatieschema.[1] Deze signature bepaalt de vorm van de invoer en uitvoer van de service, net zoals we dat bij een klassieke query zagen.

Ten derde bevat het schema de semantiek van een service, genoteerd door $[\![f]\!]$ (waarbij f een servicenaam is). De definitie van de semantiek van een service is voorlopig gewoon een query zoals we in het vorige hoofdstuk definieerden.

[1] Een andere mogelijkheid is om het resultaat van een service een andere database te laten zijn. Deze mogelijkheid beschouwen we hier niet.

Nu kunnen we **Rep** formeel definiëren. Dit doen we als volgt:

Definitie 4. *Een service schema* ***Rep*** *is een 3-tupel* $(\mathcal{S}, sig, [\![.]\!])$*, waarbij:*

- $\mathcal{S}$ *een eindige deelverzameling is van* ***dom****,*
- sig *een functie is over* $\mathcal{S}$ *die aan elke servicenaam* f *een signature* $sig(f)$ *toekent, en*
- $[\![.]\!]$ *een functie is over* $\mathcal{S}$ *die aan elke servicenaam* f *een query* $[\![f]\!]$ *toekent van type* $sig(f)$*.*

Voor het gemak voeren we nog een extra notatie in voor de delen van een serviceschema: voor $\mathbf{Rep} = (\mathcal{S}, sig, [\![.]\!])$ noteren we $\mathcal{S}$ als $\mathcal{S}_{Rep}$, sig als sig_{Rep} en $[\![.]\!]$ als $[\![.]\!]_{Rep}$.

Voorbeeld

Neem het databaseschema $\mathcal{D}$ uit het vorige hoofdstuk. Ter herinnering: dit schema bestond uit een binaire relatie R met attributen A en B, en een unaire relatie S met attribuut C.

Beschouw nu het service repository $\mathbf{Rep} = (\mathcal{S}, sig, [\![.]\!])$ waarbij

- $\mathcal{S} = \{f\}$,
- $sig(f) = \mathcal{D} \longrightarrow \mathcal{R}$, en
- $[\![f]\!] = \pi A(R)$.

Deze repository bevat dus één service die als invoer een databank over $\mathcal{D}$ neemt, en daaruit het attribuut A uit relatie R projecteert en dit als resultaat geeft.

3.3 Actieve databases

Nu zijn we klaar om voor dit model de definities van een database en een query uit de klassieke databasetheorie uit te breiden naar hun actieve tegenhangers.

Een actief databaseschema definiëren we als een koppel $(\mathcal{D}, \mathbf{Rep})$, waarbij $\mathcal{D}$ een klassiek databaseschema is en **Rep** een service schema.

In dit eenvoudige eerste model is een database instantie van een databaseschema $(\mathcal{D}, \mathbf{Rep})$ gewoon een instantie van $\mathcal{D}$, gezien de semantieken van de services op schemaniveau opgeslagen zijn.

Een term die we hier ook al vermelden is het *actief domein*. Dit definiëren we als het domein bestaande uit alle actieve elementen, dus de verzameling van alle services. Een formele definitie is dus:

Definitie 5. *Voor een actieve database instantie I over $(\mathcal{D}, \boldsymbol{Rep})$ is het actief domein $\boldsymbol{Adom}(I) = \{a \mid a \in \boldsymbol{dom}(I)$ en $a \in \mathcal{S}_{Rep}\}$, waarbij $\boldsymbol{dom}(I)$ de verzameling constanten uit **dom** is die voorkomen in I.*

We benadrukken dat de servicenamen uit **Rep** gewoon uit **dom** zijn gekozen, en dus ook gewoon als data-elementen kunnen voorkomen in de actieve database instantie. Waar nodig refereren we aan deze elementen als *actieve elementen*.

Voorbeeld

Neem $\mathcal{D}$ en **Rep** zoals we hierboven definieerden. Een actief databaseschema is nu het koppel $(\mathcal{D}, \mathbf{Rep})$.

Een instantie over dit schema is bijvoorbeeld I waarbij:

- $I(R) = \{\{A : 1, B : 2\}, \{A : 3, B : f\}, \{A : f, B : 4\}, \{A : 4, B : 5\}\}$, en
- $I(S) = \{\{C : 1\}, \{C : 2\}\}$.

3.4 Actieve queries

De precieze definitie van een actieve query ligt niet direct voor de hand, grotendeels omdat we rekening moeten houden met de eisen van genericiteit en berekenbaarheid. Voor we dus de theorie achter actieve queries vastleggen, werken we eerst een voorbeeldquery uit.

3.4.1 Voorbeeld

Beschouw het databaseschema $\mathcal{D}$ zoals hierboven, en de actieve database instantie uit de vorige sectie. Voor de duidelijkheid: de service f projecteert het A-attribuut van R en de instantie van $\mathcal{D}$ ziet er in tabelvorm als volgt uit:

R

A	B
1	2
3	f
f	4
4	5

S

C
1
2

In deze situatie zouden we nu een "actieve" query kunnen beschouwen die gebruikt maakt van f. Deze voorbeeldquery werkt op actieve databases over $(\mathcal{D}, \mathbf{Rep})$, en is als volgt gedefinieerd:

> Op invoer I:
>
> Zij $T := [\![f]\!](I)$;
>
> Vervang elk voorkomen van 'f' in I door T
>
> Geef de resulterende database terug als resultaat

Het resultaat van deze query op de gegeven instantie zou dan het volgende zijn:

R

A	B
1	2
3	1
3	3
3	f
3	4
1	4
3	4
f	4
4	4
4	5

S

C
1
2

3.4.2 Theorie

Het vorige voorbeeld gaf aan wat een actieve query in dit model zou kunnen zijn. We merken op dat de query uit het voorbeeld herschreven kan worden naar een klassieke query uit de relationele algebra. Dit kan als volgt:

$\sigma_{A\neq\text{'f'}\wedge B\neq\text{'f'}}(R)$

$\cup\ \pi_A\sigma_{A\neq\text{'f'}}\sigma_{B=\text{'f'}}(R) \times \rho_{A/B}\pi_A(R)$

$\cup\ \pi_B\sigma_{B=\text{'f'}}\sigma_{A=\text{'f'}}(R) \times \pi_A(R)$

$\cup\ \pi_\emptyset\sigma_{B=\text{'f'}}\sigma_{A=\text{'f'}} \times \pi_A \times \rho_{A/B}\pi_A(R)$

Echter, dit kan alleen omdat de semantiek van het actieve element uitdrukbaar is in de relationele algebra. Dit is op zijn beurt alleen mogelijk omdat we in deze uitdrukking een selectie toelaten op de constante 'f', wat er voor zorgt dat de query niet meer generisch is, en dus niet meer voldoet aan de definitie.

Dit motiveert ons om een nieuwe definitie van genericiteit in te voeren waarbij een generische query een eindige verzameling van constanten kan interpreteren (in dit geval zijn dit de actieve elementen). Dit soort genericiteit bestaat al in de databasetheorie, namelijk de *C-genericiteit*:

Definitie 6. *Een query q van type $\mathcal{D} \longrightarrow \mathcal{R}$ noemen we C-generisch tegenover $C \subseteq \mathbf{dom}$ als voor elke instantie I over $\mathcal{D}$ en voor elke permutatie ρ van $\mathbf{dom}$ die de identiteit is op C, geldt dat $\rho(q(I)) = q(\rho(I))$.*

Doordat de notie van genericiteit nu veranderd is, zullen we ook de eigenschap over encoderingen en berekenbaarheid moeten aanpassen en opnieuw bewijzen.

Berekenbaarheid

We veranderen de eigenschap als volgt:

Eigenschap 7. *Zij $C \subseteq \mathbf{dom}$, en zij $q : inst(\mathcal{D}) \longrightarrow inst(\mathcal{R})$ een partiële functie berekenbaar door eenzelfde TM M onder elke encodering die de identiteit is op C. Dan is q C-generisch.*

Bewijs. Neem een willekeurige encodering enc_α en een willekeurig isomorfisme ρ dat de identiteit is op C. Beschouw nu $enc'_\alpha = enc_\alpha \circ \rho$. Dit is ook een encodering, dus volgens het gegeven weten we dat het volgende geldt: $enc'_\alpha(q(I)) = M(enc'_\alpha(I)) = M(enc_\alpha(\rho(I))) = enc_\alpha(q(\rho(I)))$.

Uit de gelijkheid $enc_\alpha \circ \rho(q(I) = enc_\alpha(q(\rho(I)))$ kunnen we nu enc_α schrappen, want dit is een injectieve functie. We kunnen dus stellen dat de gelijkheid $\rho(q(I)) = q(\rho(I))$ geldt, en dat q dus C-generisch is.

□

De nieuwe genericiteit verandert dus niets wezenlijks aan onze theorie omtrent berekenbaarheid, zolang we er rekening mee houden dat de queries nu C-generisch gaan zijn tegenover de functienamen.

3.4.3 Conclusie

Het mag nu duidelijk zijn dat een actieve query in dit model precies overeenkomt met een klassieke query die C-generisch is tegenover de actieve elementen (dus tegenover $\mathcal{S}_{Rep}$). Dit komt doordat de actieve elementen eigenlijk gewoon gebruikt worden als afkorting voor een klassieke query. En gezien deze queries vastliggen in het service schema, volstaat het om de actieve elementen te kunnen herkennen in een klassieke query.

Deze conclusie geeft een duidelijke motivatie om in het volgende model precies dit te veranderen: we zullen de semantieken van de services niet langer op schemaniveau bijhouden, maar op instantieniveau.

Hoofdstuk 4

Actieve databases: tweede model

In dit hoofdstuk proberen we onze eerste poging te verbeteren om tot een krachtiger model te komen.

4.1 Aanpassingen

Uit het eerste model is gebleken dat we buiten de veralgemening naar C-generische queries geen uitdrukkingskracht kunnen halen uit actieve elementen waarvan de semantiek volledig vastligt in het serviceschema.

In dit hoofdstuk beschrijven we dan ook een tweede model. Hierin proberen we dit probleem te verhelpen door de semantiek van de services een deel te maken van de database, dus door deze op te slaan op instantieniveau in plaats van op schemaniveau.

Verder verandert er niets: er zijn nog steeds een vast aantal services, en de definities ervan zitten opgeslagen in het service schema. We laten ook nog steeds niet toe dat nieuwe services gecreëerd worden, en de semantieken van services zijn nog steeds klassieke queries.

4.2 Actieve databases

4.2.1 Schema's

De definitie van een service schema **Rep** is nu veranderd: het is een tupel bestaande uit servicenamen en hun signatures, maar geen semantieken meer. De definitie ervan wordt dus $\mathbf{Rep} = (\mathcal{S}, sig)$.

Het actieve databaseschema bestaat nog steeds uit een klassiek databaseschema en een service schema, en blijft dus $(\mathcal{D}, \mathbf{Rep})$.

Voorbeeld

Neem het klassieke databaseschema $\mathcal{D}$ uit het vorige hoofdstuk, en beschouw de service repository $\mathbf{Rep} = (\{f\}), sig)$, waarbij $sig(f) = \mathcal{D} \longrightarrow \mathcal{R}$.

Het koppel $(\mathcal{D}, \mathbf{Rep})$ is een service schema.

4.2.2 Instanties

Een actieve database instantie $\mathcal{I}$ heeft nu een extra component gekregen, namelijk de semantieken van de services uit **Rep**. Zo'n instantie bestaat dus nu uit twee delen.

Het eerste deel is de klassieke database, wat een instantie is van $\mathcal{D}$. Dit noemen we vanaf nu het passieve deel, en zullen we noteren als I.

Het tweede deel noemen we het actieve deel. Dit bestaat uit de semantieken van de services. Dit actieve deel is nog steeds een functie $[\![.]\!]$ die servicenamen uit $\mathcal{S}$ afbeeldt op queries (van het type dat door sig bepaald wordt). Vanaf nu noteren we deze functie $[\![.]\!]$ ook als $\mathcal{J}$, en we noemen ze een service instantie (van het schema **Rep**).

Een actieve database is dus nu een koppel geworden, en wordt genoteerd met $\mathcal{I} = (I, \mathcal{J})$.

Voorbeeld

Neem het databaseschema $(\mathcal{D}, \mathbf{Rep})$ zoals hierboven. Beschouw nu de actieve database instantie $\mathcal{I}_1 = (I, \mathcal{J})$ over $(\mathcal{D}, \mathbf{Rep})$ met:

- I een klassieke database instantie over $\mathcal{D}$, waarvoor
  $I(R) = \{\{A:1, B:2\}, \{A:f, B:2\}, \{A:4, B:4\}, \{A:4, B:5\}\}$
  $I(S) = \{\{C:1\}, \{C:2\}\}$, en
- $\mathcal{J}$ een functie $[\![.]\!]_1$ waarvoor $[\![f]\!]_1 = \pi_B \sigma_{A=B}(R)$.

4.3 Actieve queries

Aan de definitie van een actieve query verandert niets: het is nog steeds een functie die een actieve database afbeeldt op een relatie. Het verschil zit natuurlijk in het feit dat de semantiek van de functie kan verschillen voor elke database instantie.

Voorbeeld

Om het verschil met het vorige model nog eens expliciet te maken, beschouwen we hier nog een tweede instantie van het actieve databaseschema $(\mathcal{D}, \mathbf{Rep})$, waarbij het passieve deel I hetzelfde is als bij $\mathcal{I}_1$.

Het actieve deel is hier echter verschillend: $\mathcal{J}$ is een functie $[\![.]\!]_2$ waarvoor $[\![f]\!]_2 = \pi_B(R \bowtie \rho_{C \longrightarrow B}(S))$.

Nu beschouwen we dezelfde query als in het vorige hoofdstuk:

> Op invoer I:
>
> Zij $T := [\![f]\!](I)$;
>
> Vervang elk voorkomen van 'f' in I door T
>
> Geef de resulterende database terug als resultaat

Wanneer we deze query nu uitvoeren op $\mathcal{I}_1$ krijgen we het volgende resultaat:

R

A	B
1	2
4	2
4	4
4	5

S

C
1
2

Het resultaat van de query op $\mathcal{I}_2$ is het volgende:

R

A	B
1	2
2	2
4	4
4	5

S

C
1
2

We zien dus duidelijk het verschil met het vorige model: ook al zijn de klassieke databases waarmee we werken dezelfde, een query kan op verschillende instanties toch een verschillend resultaat geven omdat de semantieken van de services kunnen verschillen per instantie.

Nu is er nog wel een belangrijk punt waar we rekening mee moeten houden: voldoen deze actieve queries nog wel aan de definitie van een query uit sectie 2.3? We herinneren ons dat we hiervoor drie dingen eisten: typering, genericiteit en berekenbaarheid. We bekijken deze eisen weer stuk voor stuk.

4.3.1 Typering

Aan de eis van typering is nog voldaan. Het enige wat hier verandert is dat de invoerschema's van de queries nu actieve databaseschema's geworden zijn. Maar de uitvoerschema's blijven nog steeds klassieke relatieschema's, wat dus geen problemen oplevert.

4.3.2 Genericiteit

In het vorige model kwamen we er achter dat onze queries C-generisch werden tegenover $\mathcal{S}_{Rep}$. We zouden deze genericiteit ook in dit model willen bewaren, met andere woorden willen we voor elke permutatie ρ die de identiteit is op $\mathcal{S}_{Rep}$ dat $q(\rho(\mathcal{I})) = \rho(q(\mathcal{I}))$.

Maar wat is $\rho(\mathcal{I})$? We herinneren ons dat een actieve database instantie $\mathcal{I}$ van de vorm is $\mathcal{I} = (I, \mathcal{J})$. Dus dan is logischerwijze $\rho(\mathcal{I}) = (\rho(I), \rho(\mathcal{J}))$. We zullen dit nu definiëren op de natuurlijke wijze.

Definitie van $\rho(\mathcal{I})$

De definitie van $\rho(I)$ is dezelfde als bij klassieke databases, zoals gegeven in sectie 2.3.2: de puntsgewijze toepassing van ρ op de elementen in I.

$\rho(\mathcal{J})$ is echter lastiger: hiervoor moeten we de puntsgewijze toepassing van ρ op $\mathcal{J}$ definiëren. Om dit te doen definiëren we eerst $\rho(q)$ voor een query q. Merk op dat we zo'n query kunnen zien als een verzameling koppels van de vorm (I, r) met I een database instantie en r een relatie.

We definiëren nu op natuurlijke wijze $\rho(q) = \{(\rho(I), \rho(r) \mid (I, r) \in q\}$. Hierbij hebben we $\rho(r)$ en $\rho(I)$ reeds gedefinieerd als de puntsgewijze toepassingen.

Tenslotte merken we op dat gezien $\mathcal{J}$ een functie is die servicenamen op queries afbeeldt, we $\rho(\mathcal{J})$ kunnen definiëren als volgt: $\rho(\mathcal{J}) = \rho \circ \mathcal{J}$.

Nu is de volgende eigenschap interessant:

Eigenschap 8. *Voor elke service instantie $\mathcal{J}$ en elke permutatie ρ van* ***dom*** *geldt: $\rho(\mathcal{J}) = \mathcal{J}$.*

Voor we dit formeel bewijzen geven we al de intuïtie van het bewijs.

Een service instantie bestaat eigenlijk uit een verzameling queries, die aan servicenamen worden toegekend. Nu is een query een functie die elementen uit **dom** afbeeldt op elementen uit **dom**. We kunnen de query dus zien als een (mogelijk oneindige) verzameling koppels van database instanties en relatie instanties, zodat het tweede element het resultaat is van de uitvoering van de query op het eerste element. Wanneer we nu de puntsgewijze toepassing nemen van een permutatie ρ op het domein, krijgen we precies dezelfde verzameling terug. Formeel:

Bewijs. Aangezien $\rho(\mathcal{J}) = \rho \circ \mathcal{J}$, en aangezien $\mathcal{J}$ een functie is die servicenamen afbeeldt op generische queries, volstaat het aan te tonen dat $\rho(q) = q$ geldt voor elke generische query q.

Aangezien $\rho(q) = \{(\rho(I), \rho(r)) | (I, r) \in q\}$, moeten we twee implicaties aantonen: ten eerste dat $(\rho(I), \rho(r)) \in q$ voor elke $(I, r) \in q$ (dit geeft ons $\rho(q) \subseteq q$), en ten tweede dat $(I, r) \in \rho(q)$ voor elke $(I, r) \in q$ (dit geeft ons $q \subseteq \rho(q)$).

Voor de eerste implicatie: we weten dat $r = q(I)$. Wat is nu $\rho(r)$? Dankzij de genericiteit van q weten we dat $\rho(r) = \rho(q(I)) = q(\rho(I))$. Hierdoor is het duidelijk dat $(\rho(I), \rho(r)) \in q$.

Voor de tweede implicatie: $(I, r) \in \rho(q)$ geldt maar als en slechts als $(I, r) = (\rho(I'), \rho(r'))$ voor een bepaalde $(I', r') \in q$. Neem nu $I' = \rho^{-1}(I)$ en $r' = \rho^{-1}(r)$. Hiervoor is de eerste gelijkheid duidelijk voldaan. Maar geldt $(\rho^{-1}(I), \rho^{-1}(r)) \in q$ wel hiervoor? Het antwoord is ja: we kunnen dezelfde redenering gebruiken als bij de eerste implicatie: gezien q generisch is en $r = q(I)$, weten we dat $\rho^{-1}(r) = \rho^{-1}(q(I)) = q(\rho^{-1}(I))$ en dus $(\rho^{-1}(I), \rho^{-1}(r)) \in q$.

□

Dankzij dit resultaat is dus $\rho(\mathcal{I})$, voor een actieve database instantie $\mathcal{I}$ gelijk aan $(\rho(I), \rho(\mathcal{J})) = (\rho(I), \mathcal{J})$. We kunnen nu de genericiteitseis onder dit model precies uitdrukken.

Definitie van genericiteit

Definitie 7. *Een actieve query q is generisch als en slechts als voor elke permutatie ρ van* ***dom*** *die de identiteit is op $\mathcal{S}_{Rep}$, geldt dat $q(\rho(I), \mathcal{J}) = \rho(q(I, \mathcal{J}))$.*

Er is echter nog een tweede definitie voor genericiteit van actieve queries, die veel intuïtiever zal blijken:

Definitie 8. *Een actieve query q van type $(\mathcal{D}, \boldsymbol{Rep}) \longrightarrow \mathcal{R}$ is generisch als en slechts als voor elke service instantie $\mathcal{J}$ een aparte $\mathcal{S}_{Rep}$-generische query $q_{\mathcal{J}}$ van type $\mathcal{D} \longrightarrow \mathcal{R}$ bestaat, zodat voor elke instantie $(I, \mathcal{J})$ geldt dat $q(I, \mathcal{J}) = q_{\mathcal{J}}(I)$.*

Volgens deze definitie is een generische actieve query dus niets anders dan een familie van $\mathcal{S}_{Rep}$-generische queries, eentje voor elke service instantie. Deze definitie komt goed overeen met de intuïtie dit we van dit model hebben: dat een service zich anders kan gedragen in elke instantie van de actieve database.

Bewijs van de equivalentie van de definities.

Bewijs dat Definitie 7 $\Rightarrow$ Definitie 8

Gegeven is de query q zoals in Definitie 7.

We moeten bewijzen dat voor elke instantie $\mathcal{J}$ van **Rep** een $\mathcal{S}_{Rep}$-generische query $q_{\mathcal{J}}$ bestaat zodat $q = q_{\mathcal{J}}$.

We definiëren deze query als volgt: $q_{\mathcal{J}}(I) := q(I, \mathcal{J})$. Zo is per definitie voldaan aan de gelijkheid $q = q_{\mathcal{J}}$, dus we moeten nog enkel bewijzen dat deze $q_{\mathcal{J}}$'s steeds $\mathcal{S}_{Rep}$-generisch zijn.

Neem hiervoor een $\mathcal{S}_{Rep}$-generische permutatie ρ van **dom**. Nu is volgens de definitie $q_{\mathcal{J}}(\rho(I)) = q(\rho(I), \mathcal{J})$, wat volgens het gegeven gelijk is aan $\rho(q(I, \mathcal{J})) = \rho(q_{\mathcal{J}}(I))$.

Dit betekent dus dat de queries $\mathcal{S}_{Rep}$-generisch zijn, en dat de eerste definitie de tweede impliceert.

Bewijs dat Definitie 8 $\Rightarrow$ Definitie 7

Gegeven is dat $q = (q_{\mathcal{J}})$, voor elke instantie $\mathcal{J}$ van **Rep**, en dat deze $q_{\mathcal{J}}$ steeds $\mathcal{S}_{Rep}$-generisch zijn.

We moeten bewijzen dat voor $\mathcal{S}_{Rep}$-generische permutatie ρ van **dom**, geldt dat $q(\rho(I), \mathcal{J}) = \rho(q(I, \mathcal{J}))$.

We weten uit het gegeven dat $q(\rho(I), \mathcal{J}) = q_{\mathcal{J}}(\rho(I))$. Doordat elke $q_{\mathcal{J}}$ $\mathcal{S}_{Rep}$-generisch is, is dit gelijk aan $\rho(q_{\mathcal{J}}(I))$, wat volgens het gegeven gelijk is aan $\rho(q(I, \mathcal{J}))$.

□

Uitzondering

Er schuilt nog een probleem in onze huidige theorie van queries. In het huidige model is het toegelaten dat een query onder twee verschillende semantieken een totaal verschillend resultaat kan geven. Dit is intuïtief al niet echt wat we verwachten, en het strookt ook niet mooi met definitie 8, want we verwachten toch dat de familie van klassieke queries gelijkaardige berekeningen uitvoeren. We geven nu een geval waar dit niet zo is.

We beschouwen volgend voorbeeld: onderstel een databaseschema $\mathcal{D} = \{R\}$ met $sort(R) = \{A, B\}$, en een service schema **Rep** met daarin een functie f met signature $\mathcal{D} \longrightarrow \{A, B\}$. We definiëren de volgende semantieken voor f:

- $[\![f]\!]_1(r) = r$, en
- $[\![f]\!]_2(r) = r$ indien we r op generische wijze kunnen opbouwen uit een unaire tabel s op één van volgende manieren: $s \times s$, of $\sigma_{1=2}(s \times s)$, of $\sigma_{1\neq 2}(s \times s)$, en de lege verzameling anders.

De eerste semantiek is dus gewoon de identiteit, en de uitvoering ervan verandert nooit iets aan de relatie.

We beschouwen nu de volgende eigenschap, die aangeeft hoe binaire relaties op een generische manier kunnen opgebouwd worden uit een unaire relatie:

Eigenschap 9. *Een binaire relatie r die bekomen is uit een unaire relatie s door uitvoering van een generische query, voldoet steeds aan een van de volgende vormen:*

- $\{\}$
- $s \times s$
- $\sigma_{1=2}(s \times s)$
- $\sigma_{1\neq 2}(s \times s)$

Bewijs. Beschouw de volgende eigenschappen:

- r **bevat een koppel** (x, y) **met** $x \neq y$
  In dit geval bevat r alle koppels (u, v) met $u \neq v$ en $u, v \in s$. Immers, onderstel een permutatie ρ waarvoor $\rho(x) = u$, $\rho(u) = x$, $\rho(y) = v$, $\rho(v) = y$ en $\rho(z) = z$ voor $z \neq x, y, u, v$.
  Nu geldt duidelijk dat $\rho(r) = r \Rightarrow \rho((x, y)) \in r \Rightarrow (u, v) \in r$.
- r **bevat een koppel** (x, x)
  In dit geval bevat r alle koppels (u, u) met $u \in s$. Immers, onderstel een permutatie ρ waarvoor $\rho(x) = u$, $\rho(u) = x$ en $\rho(z) = z$ voor $z \neq x, u$.
  Er geldt duidelijk weer dat $\rho(r) = r \Rightarrow \rho((x, x)) \in r \Rightarrow (u, u) \in r$.

Hiermee komen we nu aan vier mogelijke vormen waaraan r kan voldoen:

- Als r aan noch de eerste, noch de tweede eigenschap voldoet, is r de lege verzameling.
- Als r alleen aan de eerste eigenschap voldoet, moet r van de vorm $\sigma_{1\neq 2}(s \times s)$ zijn.
- Als r alleen aan de tweede eigenschap voldoet, moet r van de vorm $\sigma_{1=2}(s \times s)$ zijn.
- Als r aan beide eigenschappen voldoet, is r van de vorm $s \times s$.

□

Dankzij deze eigenschap zien we dat de semantiek $[\![f]\!]_2(r)$ een binaire relatie die op generische wijze opgebouwd is uit een unaire relatie steeds op zichzelf zal afbeelden.

We merken dus duidelijk op dat er een probleem bestaat: dit geval voldoet niet aan onze algemene definitie van een generische actieve query, want de twee semantieken zullen nooit te onderscheiden zijn. Er zal dus waarschijnlijk iets moeten veranderen zodat dit geval niet toegelaten wordt.

Eerst onderzoeken we de omstandigheden waarin dit geval zich kan voordoen. We merken alvast op dat in ons voorbeeld het schema waarop van de database "zwakker" was dan het uitvoerschema van de service. We vermoeden dat dit aan de basis van het probleem ligt, en formaliseren dit.

Hiervoor gebruiken we de notie van *generic dominance*, gedefinieerd door Hull[19] als volgt: neem twee databaseschema's P en Q, en twee afbeeldingen $\sigma : P \longrightarrow Q$ en $\tau : Q \longrightarrow P$. Dan wordt P gedomineerd door Q via (σ, τ), genoteerd als $P \leq Q$ via (σ, τ), als $\tau \circ \sigma$ de identiteit is op $I(P)$. Intuïtief betekent dit dus dat de informatiecapaciteit van P gesubsumeerd wordt door die van Q, of op zijn minst erdoor evenaart wordt.

We gebruiken hier echter een aangepaste definitie:

Eigenschap 10. *Hull's definitie van dominance is equivalent met te zeggen dat er een $\sigma' : P \longrightarrow Q''$ bestaat die injectief is.*

Bewijs. De $\Rightarrow$-implicatie: Er bestaan dus σ, τ zodat $P \leq Q$ via (σ, τ). We weten dat $\tau \circ \sigma$ de identiteit is. Dan volgt daaruit meteen dat σ injectief moet zijn, want stel dat voor instanties I_1, I_2 geldt dat $\sigma(I_1) = \sigma(I_2)$, dan $\tau(\sigma(I_1)) = \tau(\sigma(I_2))$, en dus is $I_1 = I_2$. We hebben dus een injectie gevonden, zodat $P \leq Q$ via σ'.

De $\Leftarrow$-implicatie: Er bestaat een dus een injectie σ' zodat $P \leq Q$ via σ'. Neem dan σ' als σ, en neem σ^{-1} als τ (dit kan, want σ' is een injectie). Nu geldt dat $P \leq Q$ via (σ, τ).

□

Om het bovenstaande geval te veralgemenen, zoeken we nu een voorbeeld zodat we voor een functie f : service-in-schema $\longrightarrow$ service-out-schema waarbij service-in-schema $\leq_{generic}$ databaseschema, twee verschillende semantieken $[\![f]\!]_1$ en $[\![f]\!]_2$ kunnen vinden zodat $(q_{[\![f]\!]_1}(I)) = (q_{[\![f]\!]_2}(I))$, en dit voor elke generische query q.

Het is gemakkelijk in te zien dat we zo geen voorbeeld kunnen vinden. Inderdaad, gezien de semantieken van f verschillen, moet er minstens één instantie J bestaan zodat $[\![f]\!]_1(J) \neq [\![f]\!]_2(J)$ (dit geldt natuurlijk ook voor alle isomorfismen van J).

Uit het vorige kunnen we nu twee gevallen onderscheiden:

Als het databaseschema minstens één binaire relatie of hoger bevat kunnen we J construeren door een keten op te bouwen. We kunnen nu gewoon telkens J teruggeven.

In het tweede geval bevat het databaseschema dus slechts unaire relaties. Hier geldt dat als het databaseschema minstens evenveel relaties bevat als er elementen zijn in J, deze relaties zelfs unair mogen zijn. We stoppen dan gewoon elk element van J in een aparte relatie, en nemen de unie van de correcte carthesische producten. Op het voorbeeld hierboven toegepast wordt dat dan $(R_a \times R_b \times R_c) \cup (R_b \times R_c \times R_d) \cup (R_d \times R_e \times R_f)$.

We zien dus nu dat ons speciale geval alleen geldt als er alleen unaire relaties in het databaseschema zitten, en er minder zijn dan het aantal elementen in een instantie waarop de semantieken verschillen. Dit is een beperking waar we ons bewust van moeten zijn.

4.3.3 Berekenbaarheid

Ook de eis van berekenbaarheid moeten we opnieuw formuleren, want de notie van berekenbaarheid van actieve queries is nu duidelijk anders geworden dan bij klassieke queries.

We passen hiervoor de theorie rond berekenbaarheid uit Hoofdstuk 2 stap voor stap aan.

Encoderingen

De theorie rond enumeraties en encoderingen blijft wel hetzelfde als in hoofdstuk 2. Het enige wat verandert is dat we nu ook de servicenamen moeten encoderen. Hier spreken we echter af dat deze mee in het alfabet van de encodering zitten. Dit verandert niets aan de theorie.

Oracle Turing Machines

Om de berekenbaarheid van actieve queries te definiëren, hebben we echter een ander soort van Turing Machine nodig: de Oracle Turing Machine (OTM). Intuïtief ziet deze er als volgt uit: een klassieke TM met een "orakel" eraan gekoppeld. Dit orakel kan een bepaald probleem in een enkele stap oplossen, zonder aan te geven hoe dit gebeurt.

Formeel is de OTM als volgt gedefinieerd: een klassieke TM die uitgebreid is met heeft twee extra tapes, een querytape waarnaar alleen geschreven kan worden en een answertape waarvan alleen gelezen kan worden. Verder heeft ze twee extra toestanden, de querytoestand en de answertoestand. Tenslotte is er het orakel, dat een gegeven partiële functie is van $\Sigma^* \longrightarrow \Sigma^*$. De OTM werkt als volgt: wanneer ze in de querytoestand komt leest het orakel de querytape en voert er de functie op uit, waarbij ze het antwoord op de answertape schrijft. Daarbij gaat de TM over naar de answertoestand. Dit gebeurt allemaal in een atomaire stap.

Nu kunnen we deze Oracle Turing Machines gebruiken. Het idee achter de OTM waarvan we het bestaan zullen eisen is het volgende: de TM is het "hoofdalgoritme", en het orakel simuleert een aantal "inplugbare" deelalgoritmen die de TM kan oproepen. Het orakel komt dus overeen met de services van een bepaalde service instantie, terwijl de TM de uitvoering ervan coördineert.

Het orakel dat we nu in deze context zullen construeren werkt als volgt: op invoer s kijkt het orakel eerst of s bestaat uit de naam van een service, gevolgd door de encodering van een database over het schema dat die service als invoer verwacht. Als dit het geval is, simuleert ze de service op de database (volgens de semantiek uit $\mathcal{J}$) en geeft ze het resultaat terug. Zoniet is de functie niet gedefinieerd. Natuurlijk is ook het orakel afhankelijk van de gebruikte encodering, we spreken dus steeds van $oracle_\alpha(\mathcal{J})$, en noteren een OTM M die dit als orakel gebruikt met $M^{oracle_\alpha(\mathcal{J})}$.

We zien zo intuïtief in dat Oracle Turing Machines de notie van berekenbaarheid in dit model mooi omvatten: een hoofdalgoritme (de actieve query) die verschillende deelalgoritmes oproept (de services).

Berekenbaarheid

Een actieve query q is nu berekenbaar ten opzichte van een encodering α als er een Oracle Turing Machine $M^{oracle_\alpha(\mathcal{J})}$ bestaat zodat voor elke actieve instantie $\mathcal{I}$ geldt dat als $q(\mathcal{I})$ niet gedenieerd is, $M^{oracle_\alpha(\mathcal{J})}$ niet stopt op invoer $enc_\alpha(I)$, en als $q(\mathcal{I})$ wel gedenieerd is, $M^{oracle_\alpha(\mathcal{J})}$ stopt op invoer $enc_\alpha(I)$ met $enc_\alpha(q(\mathcal{I}))$ op zijn (uitvoer)band.

Natuurlijk moeten we nu de eigenschap rond encoderingen en genericiteit ook opnieuw doen:

Eigenschap 11. *Zij q een S_{Rep}-generische actieve query. Dan is q berekenbaar door een OTM M tegenover encodering $enc_\alpha \Leftrightarrow q$ is berekenbaar door dezelfde M tegenover alle mogelijke encoderingen.*

Bewijs. We moeten alleen de $\Rightarrow$-implicatie bewijzen, de andere richting is triviaal. We moeten dus bewijzen dat gegeven een encodering enc_α waarvoor q berekenbaar is door M, geldt dat q berekenbaar is door dezelfde M onder elke encodering. Formeel is dit dus: $\forall\beta : enc_\beta(q(\mathcal{I})) = M^{oracle_\alpha(\mathcal{J})}(enc_\beta(I))$.

Beschouw nu een willekeurige encodering enc_β, en bekijk $\rho = enc_\alpha^{-1} \circ \beta_e$.

Nu is $M^{oracle_\beta(\mathcal{J})}(enc_\beta(I)) = M^{oracle_\alpha(\rho(\mathcal{J}))}(enc_\alpha(\rho(I)) = enc_\alpha(q(\rho(I), \rho(\mathcal{J})) = enc_\alpha(\rho(q(I, \mathcal{J}))) = enc_\beta(q(I, \mathcal{J}))$.

□

Hoofdstuk 5

Actieve databases: querytheorie

In dit hoofdstuk proberen we een complete querytaal voor actieve databases te vinden.

5.1 Inleiding

Het tweede model dat we in het vorige hoofdstuk uitwerkten lijkt voldoende robuust om te dienen als model voor actieve databases. We gebruiken dit model dan ook om onze querytheorie op te bouwen.

Nu gaan we op zoek naar een querytaal voor actieve databases die compleet is, met andere woorden die alle mogelijke queries over actieve databases kan uitdrukken.

We hebben in Hoofdstuk 2 al gezien dat voor klassieke databases de taal while$_{new}$ compleet is (wanneer we alleen de well-behaved while$_{new}$-programma's beschouwen), dus we zullen deze als uitgangspunt gebruiken om een complete taal voor actieve databases te vinden.

Eerst geven we een beschrijving van de uitbreiding die while$_{new}$ nodig heeft om compleet te zijn, en vervolgens bewijzen we de compleetheid van de nieuwe taal.

5.2 while$_{new}$ + *call*

Wat moeten we nu toevoegen om ook gebruik te kunnen maken van de uitdrukkingskracht die we gewonnen hebben door services toe te laten in onze databases? Het intuïtieve antwoord ligt eigenlijk voor de hand: we hebben een operator nodig die ons toelaat om de services aan te spreken, en hun resultaat terug te geven. We zullen deze operator *call* noemen.

De querytaal bevat nu dus:

- Alle uitdrukkingen en operators uit while$_{new}$.
- Een *call*-operator van de vorm $call(f, x, y, \ldots)$, waarbij f een servicenaam is en $x, y, ...$ de argumenten van f zijn. Deze operator voert de service f uit en geeft de resulterend relatie terug.

De *call*-operator neemt dus een variabel aantal argumenten mee, waarvan de eerste de naam van een service uit het service schema is, en de andere elementen relaties zijn die samen een database vormen die voldoet aan het invoerschema van die service.

Als f dus een service is die als invoer twee relaties verwacht die voldoen aan schema R en A en B zijn relatievariabelen over dit schema, is het volgende een geldige uitdrukking:

$X := call(f, A, B)$

5.3 Compleetheid van while$_{new}$ + *call*

Om te bewijzen dat deze taal compleet is gebruiken we een analoge redenering aan hetgene waarmee we in Hoofdstuk 2 aantoonden dat while$_{new}$ compleet is voor klassieke databases.

We geven in deze sectie dan ook niet alle stappen in detail, gezien deze in Hoofdstuk 2 al voldoende duidelijk opgenomen staan. We brengen hier alleen de verschillen met het bewijs voor klassieke databases aan.

5.3.1 Compleetheid van while$_N$ + *call*

We hebben eerst volgende eigenschap nodig:

Eigenschap 12. *De taal while$_N$ waar we de call-operator aan toevoegen is compleet voor geordende actieve databases.*

We kunnen eigenlijk het bewijs uit Hoofdstuk 2 grotendeels hergebruiken, op één belangrijk verschil na.

Zolang er geen services aangeroepen worden kan al het rekenwerk gedaan worden door een gewone Turing Machine, die we dus kunnen simuleren in while$_N$ (gezien deze volledige berekenbaarheid biedt op de integers). Wanneer we nu een service aanroepen wordt het iets ingewikkelder, want op dat moment komt het orakel in actie. Hier moeten we dus het getal even decoderen naar relaties, want dit is het enige wat het orakel begrijpt. Het resultaat van de berekeningen van het orakel encoderen we dan weer (volgens dezelfde enumeratie), waarna we verder kunnen met de gewone Turing Machine. Dit encoderen en decoderen kan gedaan worden in while$_{new}$, zoals in Hoofdstuk 2 al werd aangegeven.

5.3.2 Compleetheid van while$_{new}$ + *call*

De constructie die we in Hoofdstuk 2 gebruikten om de compleetheid van while$_{new}$ aan te tonen was de volgende:

Onderstel een programma w_{succ} dat een query q berekent gegeven een totale orde in de database. Voer dan volgende stappen uit:

- Construeer een binaire relatie $\overline{succ}$ die alle enumeraties van elementen uit het effectief domein bevat als ketens.
- Pas w_{succ} aan zodat het de query q berekent voor elke enumeratie uit $\overline{succ}$, in parallel.
- De resultaten hiervan zijn wegens genericiteit allemaal gelijk aan het resultaat van q.

$\overline{succ}$ construeren

De constructie van alle enumeraties kan eigenlijk letterlijk overgenomen uit Hoofdstuk 2. Het enige wat er hier verandert is dat er ook actieve elementen voorkomen in de database, maar gezien we deze in het alfabet van de encodering opnamen verandert dit niets aan de enumeraties.

w_{succ} aanpassen

Hier stuiten we op een probleem: om het programma w_{succ} in parallel uit te voeren voor elke enumeratie, moeten we iets met de *call*-operator doen zodat deze de webservices in parallel kan uitvoeren. Dit is lastig, want gezien het resultaat van de operator slechts een enkele relatie is, wordt het moeilijk om aan te duiden welk resultaat bij welke encodering hoort.

We lossen dit probleem voorlopig op door de *call*-operator tijdelijk te vervangen door een nieuwe operator *parallel-call*. Deze operator werkt precies hetzelfde als de gewone *call*, alleen houdt hij rekening met de relatie $\overline{succ}$. Bij een aanroep van *parallel-call* wordt dus een service uitgevoerd voor elke enumeratie, en wordt er een relatie met markeringen (de δ's uit Hoofdstuk 2) teruggegeven. Verderop bewijzen we dan dat *parallel-call* gesimuleerd kan worden in while$_{new}$ + *call*.

De andere aanpassingen uit Hoofdstuk 2 kunnen we gewoon behouden, hierop heeft het actief zijn van de database geen invloed.

Voorlopig resultaat

Uit het voorgaande kunnen we dus besluiten dat de taal while$_{new}$ + *parallel-call* compleet is. Maar we zijn hier natuurlijk geïnteresseerd in while$_{new}$ + *call*.

Er valt dus nog te bewijzen dat we de *parallel-call*-operator kunnen simuleren met de gewone *call*-operator.

5.4 *parallel-call* simuleren

Het idee achter de simulatie van *parallel-call* is eigenlijk eenvoudig: we voeren gewoon de *call*-operator sequentieel uit voor elke instantie die bij een δ uit $\overline{succ}$ horen door middel van een while-lus, en plakken dan alle resultaten samen.

Deze aanpak kent echter een probleem: er is geen orde op de instanties waarop we de sequentie zouden kunnen baseren. We moeten dus op een of andere manier zelf een orde construeren. Hiervoor hebben we het volgende nodig.

5.4.1 Canonieke encoderingen

We gebruiken hier een bekend concept binnen de wiskunde, namelijk dat van de *canonieke ordeningen*.

We denken even terug aan de encoderingen. Dit zijn dus strings over een eindig alfabet die een database voorstellen door een structuur te geven met daarbinnen de elementen uit de database als binaire getallen voorgesteld. We kunnen nu zonder verlies van algemeenheid stellen dat die binaire getallen gaan van 1 tot en met n, waarbij n het aantal elementen is.

Nu kunnen we alle mogelijke encoderingen nemen, en deze lexicografisch ordenen. Hiermee kom je dus een bepaalde string uit die we de kleinste kunnen noemen. De encodering die hiermee overeenkomt noemen we nu de *canonieke encodering*, en de enumeratie die leidde tot deze encodering noemen we de *canonieke ordening*.

De canonieke encodering van een bepaalde database is duidelijk uniek, maar het kan wel zijn dat een aantal verschillende enumeraties de canonieke encodering genereren. Beschouw bijvoorbeeld een database met precies een unaire relatie, die twee tupels $\{(a), (b)\}$ bevat. Er zijn nu twee verschillende enumeraties en deze leiden allebei tot de encodering $\{\{(1), (10)\}\}$, wat dus ook de canonieke encodering is.

Het simpelste geval waarbij er verschillende enumeraties een verschillende encodering opleveren is een database met precies een binaire relatie die slechts een tupel (a, b) bevat. Er zijn nu twee enumeraties mogelijk, en deze geven

verschillende encoderingen, namelijk $\{\{(1,10)\}\}$ en $\{\{(10,1)\}\}$. In dit geval is er dus slechts één van de enumeraties die de canonieke encodering oplevert.

Een belangrijke eigenschap van canonieke encoderingen die we later zullen nodig hebben is de volgende:

Eigenschap 13. *Twee databases zijn isomorf $\Leftrightarrow$ ze hebben dezelfde canonieke encodering.*

Bewijs.

Bewijs van $\Rightarrow$ Neem twee databases D_1 en D_2 die isomorf zijn. Er bestaat dus een permutatie ρ zodat $D_2 = \rho(D_1)$. Beschouw nu een canonieke ordening π_1 van D_1.

We moeten nu bewijzen dat $\pi_2 = \pi_1 \circ \rho^{-1}$ een canonieke ordening van D_2 is, met andere woorden dat voor een willekeurige ordening $\pi_3 : D_2 \longrightarrow \{1, \ldots, n\}$ geldt dat $enc_{\pi_3}(D_2) \geqslant enc_{\pi_2}(D_2)$.

Uit het gegeven weten we dat voor een willekeurige ordening $\pi_4 : D_1 \longrightarrow \{1, \ldots, n\}$ geldt dat $enc_{\pi_4}(D_1) \geqslant enc_{\pi_1}(D_1)$.

Neem nu $\pi_4 = \pi_3 \circ \rho$. We weten dan dat $enc_{\pi_3 \circ \rho}(D_1) \geqslant enc_{\pi_1}(D_1)$. Dit betekent dus dat $enc_{\pi_3}(\rho(D_1) \geqslant enc_{\pi_1}(\rho^{-1}(\rho(D_1)))$, wat betekent dat $enc_{\pi_3}(D_2) \geqslant enc_{\pi_2}(D_2)$.

Hieruit kunnen we besluiten dat de canonieke encoderingen van D_1 en D_2 gelijk zijn.

Bewijs van $\Leftarrow$ Gegeven is twee databases D_1 en D_2 die dezelfde canonieke encodering hebben. We zoeken dus nu een permutatie ρ zodat $D_2 = \rho(D_1)$, wat betekent dat de databases isomorf zijn.

Gezien de encoderingen de volgorde van de relaties en de attributen lexicografisch ordenen, kunnen de databases alleen maar verschillen op de volgorde van de tupels zelf.

We definiëren ρ nu als volgt: wanneer we door de canonieke encodering van D_1 en D_2 lopen, beeldt ρ het element dat in D_1 met i genummerd is af op het element dat in D_2 met i genummerd is.

Nu is het duidelijk dat $D_2 = \rho(D_1)$, en dat D_1 en D_2 dus isomorf zijn.

□

5.4.2 Simulatie van *parallel-call*

De canonieke ordeningen kunnen ons helpen bij het simuleren van *parallel-call*, maar we blijven met een probleem zitten: verschillende instanties kunnen dezelfde canonieke encodering hebben, namelijk wanneer ze isomorf zijn. Dit betekent dat we nog steeds geen volgorde kunnen kiezen om de sequentiële stappen van de simulatie in te doen, want door de genericiteit kunnen we deze verschillende instanties niet onderscheiden.

De constructie die volgt zal ons echter toelaten om, gebruik makend van de canonieke ordeningen, een *parallel-call* te simuleren met gewone *call*s.

De algemene intuïtie erachter is de volgende: we zullen alle instanties die dezelfde canonieke encodering hebben gaan bundelen in equivalentieklassen (equivalent in de zin dat ze isomorf zijn). Zo groeperen we eigenlijk alle verschillende berekeningen die hetzelfde resultaat zullen opleveren. Binnen deze equivalentieklassen construeren we dan afbeeldingen zodat elke instantie erbinnen uiteindelijk een zelfde effectief domein krijgt. Op dat moment zal dus de uitvoering van de service steeds hetzelfde resultaat opleveren voor elke instantie, en kunnen we ze uitvoeren. Daarna moeten we volgens de inversen van elke afbeelding terugkeren naar de oorspronkelijk elementen van elke instantie.

Ketens opbouwen

De eerste stap die we moeten doen is het opbouwen van een keten voor elke instantie r, en wel eentje van lengte $n = |adom(r)|$, dus met een link voor elk element in het effectief domein van de instantie.

In deze en de volgende secties geven we steeds aan hoe de constructie moet verlopen voor één enkele instantie. In werkelijkheid moeten we deze stappen in parallel uitvoeren voor elke instantie die *parallel-call* zou beschouwen. We hebben echter al gezien in Hoofdstuk 2 hoe we dit kunnen doen, dus voor de eenvoud en duidelijkheid laten we het hier weg.

We beschrijven nu hoe we zo'n keten kunnen opbouwen, gegeven een unaire relatie S waarin alle elementen uit het effectief domein van r zitten (zo'n relatie kunnen we gemakkelijk construeren met een gewone algebra-query die de unie van de projecties van elk attribuut neemt). We geven ook een voorbeeld in het geval dat $S = \{a_1, a_2, a_3\}$.

- We gebruiken drie relatievariabelen: C, H en $chain$.

  1. We initialiseren de variabele C als volgt:
     $C := new(S)$
  2. Vervolgens kennen we aan een hulpvariabele H het volgende toe:
     $H := new(\{(x, y) \mid \exists z \exists u(z, y) \in C \wedge (x, u) \in C \wedge (x, y) \notin C\})$
     In H stoppen we dus alle koppels (a_i, δ_j) die nog niet voorkwamen in C, en plaatsen er een nieuw element bij. Het idee hierachter is dat δ_j in een keten te bereiken is via δ_i, dus we labelen het nieuwe element voor het overzicht met δ_{ij} (dit is echter een willekeurig nieuw element).
  3. Nu beginnen we de relatie die de ketens zal bevatten:
     $chain := \pi_{2,3}(H)$
     Deze relatie bevat dus nu ketens van lengte 1.

  In het voorbeeld zien de relaties er als volgt uit:

C:

a_1	δ_1
a_2	δ_2
a_3	δ_3

H:

a_1	δ_2	δ_{12}
a_1	δ_3	δ_{13}
a_2	δ_1	δ_{21}
a_2	δ_3	δ_{23}
a_3	δ_1	δ_{31}
a_3	δ_2	δ_{32}

chain:

δ_2	δ_{12}
δ_3	δ_{13}
δ_1	δ_{21}
δ_3	δ_{23}
δ_1	δ_{31}
δ_2	δ_{32}

- Nu beginnen we met de keten iteratief op te bouwen.

  1. We passen C aan als volgt:
     $C := \{(x, y) \mid \exists z(x, z, y) \in H\} \cup$
     $\{(x, y) \mid \exists z \exists u(x, z) \in C \wedge (u, z, y) \in H\}$
     Dus we stoppen in C de elementen a_i en a_j wanneer er een δ_{ij} in H zit. Dit zijn dus de elementen die al in de keten via δ_i zitten.
  2. Nu passen we H weer aan zoals in de vorige stap:
     $H := new(\{(x, y) \mid \exists z \exists u(z, y) \in C \wedge (x, u) \in C \wedge (x, y) \notin C\})$
  3. We voegen een link toe aan onze keten:
     $chain := chain \cup \pi_{2,3}(H)$

In ons voorbeeld wordt dit:

C:

a_1	δ_{12}
a_2	δ_{12}
a_1	δ_{13}
a_3	δ_{13}
a_1	δ_{21}
a_2	δ_{21}
a_2	δ_{23}
a_3	δ_{23}
a_1	δ_{31}
a_3	δ_{31}
a_2	δ_{32}
a_3	δ_{32}

H:

a_3	δ_{12}	δ_{123}
a_2	δ_{13}	δ_{132}
a_3	δ_{21}	δ_{213}
a_1	δ_{23}	δ_{231}
a_2	δ_{31}	δ_{312}
a_1	δ_{32}	δ_{321}

chain:

δ_2	δ_{12}
δ_{12}	δ_{123}
δ_3	δ_{13}
δ_{13}	δ_{132}
δ_1	δ_{21}
δ_{21}	δ_{213}
δ_3	δ_{23}
δ_{23}	δ_{231}
δ_1	δ_{31}
δ_{31}	δ_{312}
δ_2	δ_{32}
δ_{32}	δ_{321}

- We blijven deze stappen herhalen tot er geen elementen meer in C zitten. Op dat moment zijn onze ketens ook lang genoeg. In ons voorbeeld is dit in de vorige stap het geval.

We hebben dus nu een relatie *chain* geconstrueerd waarin $n!$ ketens van nieuwe elementen zitten, elk n links lang. We moesten echter maar één keten hebben. We construeren deze als volgt:

$U := new(\{\})$

$U := new(U)$

$X := \{(x) \mid \exists y(x, y) \in chain \wedge \forall z(z, x) \notin chain\}$

while X do

$H = new(\{\})$

$U := U \cup \{(x, y) \mid y \in H \wedge \exists z(z, x) \in U \wedge \forall u(x, u) \notin U\}$

$X := \{(x) \mid \exists y(y, x) \in chain \wedge y \in X\}$

end

Na het uitvoeren van dit programma zit er in U een keten van n (nieuwe) elementen lang, wat ons doel was.

Bijecties maken

In de volgende stap zoeken we alle bijecties die het effectief domein van een instantie afbeelden op de elementen van de keten die we in de vorige stap opbouwden voor die instantie, en daarbij de canonieke encodering van die instantie opleveren. Deze constructie bestaat eigenlijk uit twee delen.

In het eerste deel genereren we alle mogelijke bijecties van de elementen van het effectief domein van r naar de keten die we in de vorige stap verkregen hebben. Gezien zo'n bijectie eigenlijk niets anders is dan een verzameling koppels, kunnen we deze bijecties voorstellen door een ternaire relatie: in de eerste twee kolommen de koppels, en in de derde kolom een element om te markeren bij welke bijectie ze horen. Dit is weer een constructie waarop we eigenlijk alle mogelijkheden genereren, en werkt eigenlijk op dezelfde manier als degene in de vorige stap. Ze is dus ook uit te drukken in while_{new}.

In het tweede deel van deze stap filteren we uit alle bijecties degene die canoniek zijn. Dit is een klassieke berekenbare query, waarvan we uit Hoofdstuk 2 weten dat die in while_{new} kan uitgedrukt worden.

Equivalentieklassen maken

Wat we nu kunnen doen is alle instanties die dezelfde canonieke encodering hebben groeperen in equivalentieklassen. Dit is een logische stap op weg naar het simuleren van *parallel-call*, want alle instanties binnen zo'n klasse zullen toch hetzelfde resultaat geven wanneer de service erop uitgevoerd wordt.

Het groeperen in equivalentieklassen steunt gewoon op lexicografische vergelijkingen, en kan duidelijk gebeuren in relationele algebra + while.

Equivalentieklassen ordenen

Vervolgens bouwen we een nieuwe keten per equivalentieklasse, weer zo lang als het aantal elementen in het effectief domein. De constructie hiervan is precies dezelfde als in eerste stap.

Belangrijk bij deze stap is wel dat we steeds de correspondenties bijhouden tussen de verschillende instanties en hun bijecties, en tussen de bijecties en de keten. Op die manier kunnen we later nog terugkeren naar instantieniveau, door de inversen van deze afbeeldingen te nemen.

Doordat we nu een keten per equivalentieklasse hebben geconstrueerd, kunnen we ook deze gaan ordenen volgens dezelfde redenering als voor instanties.

We merken nu op dat gezien het steeds gaat om ketens met geheel nieuwe elementen, de ketens nu wel uniek zullen zijn per klasse en we ze dus kunnen gebruiken om er de canonieke encodering mee te bouwen.

parallel-call simuleren

Nu zijn we eindelijk klaar om de simulatie van de *parallel-call* uitdrukking uit te voeren. Door middel van de bovenstaande constructie hebben we een uiteindelijk een soort orde gecreëerd op de instanties, zodat we de *parallel-call* kunnen simuleren door een sequentiële rij van *call*s.

We beginnen het while$_{new}$ + *call* programma dat deze berekening doet met een lus die door alle equivalentieklassen loopt. Vervolgens beschrijven we wat er moet gebeuren in zo'n iteratie van deze lus:

- Voer het volgende uit voor elke bijectie in de equivalentieklasse, en daarbinnen voor elke instantie die bij de bijectie hoort:

  1. Pas de bijectie toe op de instantie. Zo krijgen we dus een nieuwe instantie die isomorf is met de oorspronkelijke, maar alleen nieuwe elementen bevat (die opgeslagen zitten in het beeld van de bijectie).
  2. Vervang elk element uit de nieuwe instantie door het overeenkomstige element uit de keten die bij de equivalentieklasse hoort. Dit is opnieuw een instantie die nog steeds isomorf is met de oorspronkelijke instantie, maar nu elementen bevat uit de keten die bij de equivalentieklasse hoort.

  Merk nu op dat alle instanties die we nu bekomen zijn, gelijk zijn. Dit doordat de oorspronkelijke relaties dezelfde canonieke encodering hadden, en we ze uiteindelijk afbeelden op dezelfde elementen. Dit is intuïtief ook gemakkelijk in te zijn: al deze instanties zijn isomorf, en we vervangen hun elementen steeds door nieuwe elementen volgens dit isomorfisme.

- Pas nu de service die aangeroepen werd met *parallel-call* toe op al deze (equivalente) instanties. Wegens de genericiteit van de service is het resultaat is weer telkens gelijk: een relatie over de elementen uit de keten die bij de equivalentieklasse hoort, en die isomorf is met het resultaat van de service op elk van de instanties in de klasse.

- Stap nu terug naar de elementen van de oorspronkelijke instanties door eerst het invers van elke afbeelding op de elementen van de keten die bij de equivalentieklasse hoort toe te passen, en daarna het invers van elke bijectie die hierbij hoort daarop toe te passen.

Nu hebben we voor elke instantie die behoorde tot de equivalentieklasse het resultaat van de service voor deze instantie. Het enige wat nu nog moet gebeuren is al deze resultaten bundelen tot één relatie, opnieuw gemarkeerd met de oorspronkelijke δ's. Dit is het resultaat van de *parallel-call* aanroep.

Hiermee hebben we dus aangetoond dat we *parallel-call* kunnen simuleren door een while$_{new}$ + *call* programma. Dit wat het laatste wat we moesten aantonen om volgende eigenschap te mogen besluiten:

Eigenschap 14. *De taal while$_{new}$ + call is compleet, met andere woorden ze kan alle queries uitdrukken.*

Conclusie

In dit werk hebben we geprobeerd de theorie achter actieve databases te concretiseren, en dan in het bijzonder de querytheorie.

We zijn begonnen vanuit het idee van Active XML, en hebben dit proberen te vereenzelvigen met het concept van meta-querying. Hieruit formuleerden we een eerste model dat conceptueel heel mooi was, maar niet krachtig genoeg bleek om echte uitdrukkingskracht toe te voegen aan klassieke databases.

We formuleerden een tweede model, dat zowel robuust als elegant bleek te zijn, en waarop we een querytheorie konden uitbouwen. We formuleerden een querytaal en bewezen tenslotte haar compleetheid.

Wat is nu het uiteindelijke resultaat van dit werk? Het antwoord hierop is tweeledig: ten eerste weten we meer over het queriën van actieve elementen binnen een database, en hebben we een formele theorie ontwikkeld die eventueel terug vertaald kan worden naar Active XML. En ten tweede hebben we aangetoond dat er niet meer nodig is om een complete querytaal voor actieve databases te construeren dan een operator die de actieve elementen aanroept toe te voegen, iets wat we sowieso moeten doen.

We moeten ons echter bewust zijn van de limitaties van dit werk. Zowel op vlak van modellering als querytheorie is er nog veel werk mogelijk. In het eerste geval kunnen we ons model nog in verschillende richtingen uitbreiden, bijvoorbeeld door de schema's van de services ook per instantie te laten verschillen. In het tweede geval kunnen we rekening gaan houden met de complexiteit van onze queries, en bijvoorbeeld queries binnen PTIME proberen te karakteriseren.

We hopen dan ook dat dit werk als een bouwsteen mag dienen in de fundamenten van de theorie van actieve databases!

Bibliografie

[1] W3C. Extensible markup language, 1996. http://www.w3.org/XML/.

[2] Tamara Vos. Active xml. Master thesis, tUL, 2006.

[3] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Towards a flexible model for data and web services integration. proc. Internat. Workshop on Foundations of Models and Languages for Data and Objects,Italy, 2001.

[4] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging intensional xml data. *ACM Transactions on Database Systems*, 30(1):1–40, March 2005.

[5] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Active xml and active query answers. In Lecture Notes in Computer Science, editor, *International Conference on Flexible Query Answer (FQAS)*, volume 3055, pages 17–27. Springer, 2004.

[6] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive active xml. In *ACM SIGMOD-SIGACT- SIGART Symposium on Principles of Database Systems*, June 2004.

[7] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Lazy query evaluation for active xml. In *ACM SIGMOD Conference on Management of Data*, June 2004.

[8] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active xml: Peer-to-peer data and web services integration. In *Very Large Databases Conference (VLDB), demo*, 2002.

[9] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive Active XML. In *PODS*, pages 35–45, 2004.

[10] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Frederic Dang Ngoc. Exchanging Intensional XML Data. In *SIGMOD Conference*, pages 289–300, 2003.

[11] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Active XML and Active Query Answers. In *FQAS*, pages 17–27, 2004.

[12] Serge Abiteboul, Omar Benjelloun, Bogdan Cautis, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Lazy Query Evaluation for Active XML. In *SIGMOD Conference*, pages 227–238, 2004.

[13] Active XML team. Homepage Active XML, 2003. http://activexml.net/.

[14] Jan Van den Bussche, Dirk Van Gucht, and Gottfried Vossen. Reflective programming in the relational algebra. *Journal of computer and system sciences*, 52(3):537–549, 1996.

[15] Frank Neven, Jan Van den Bussche, Dirk Van Gucht, and Gottfried Vossen. Typed query languages for databases containing queries. *Information systems*, 24(7):569–595, 1999.

[16] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Towards practical meta-querying. *Information systems*, 30(4):317–332, 2005.

[17] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison Wesley, 1994.

[18] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[19] Richard Hull. Relative information capacity of simple relational database schemata. *Siam J. Comput.*, 15, no. 3:856–886, 1986.