Equivalences of Schema Mappings

4th year project report for candidate 1035522

Honour School of Computer Science - Part C

Submitted as part of an MCompSci in Computer Science

Trinity term, 2022

*

*Word count: 9364

Abstract

Schema mappings are high-level specifications that describe how to map information from a source schema to a target schema. Schema mapping equivalence refers to situations where a mapping leads to the same output, given a fixed input. The project will discuss methods to check this equivalence, together with results about decidability on the checking procedure.

The majority of computer science areas nowadays deal extensively with different forms of finite relational structures, usually with some form of constraints attached. In the databases domain, we can view the usual relations (tables) as structures, being constrained by dependencies, which impose relationships on the data that can appear in instances. In graph theory, we can view graphs as structures, and the constraints can generate the problems that concern us, for example k-coloring. In this project, we will try to formalize precisely the relational structures and constraints. We will cover a very researched class of constraints, namely tuple-generating dependencies, examining their behavior with respect to database instances. We will also examine how we can decide equivalence of schemas, by imposing conditions on the form of the dependencies.

Acknowledgements

Many thanks to my supervisor Michael Benedikt for coming up with the idea of the project, for all the help and advice provided, and for constantly bringing up new ideas and methods to solve the problems at hand. His papers and lecture notes constituted the basis for the work of this project, and his weekly meeting notes proved to be essential in progressing with the contributions.

Contents

Abstract **3**

Acknowledgements **3**

1. Introduction **6**

1.1. Motivation 6

1.2. Contributions 6

1.3. Challenges 7

1.4. Structure of the Report 7

2. Preliminaries **8**

2.1. Relational Database Schema 8

2.2. First-order Logic 8

2.3. Queries 9

2.4. Dependencies 10

2.5. Open World Query Answering 12

2.6. Forward Chaining: The Chase 12

2.7. Tree-like Property of the Chase 15

3. Equivalences for the Chase and CQs **18**

3.1. Definitions 18

3.2. Implications 18

3.3. Decidability for Equivalences 23

4. Chase Entailment for LTGDs **25**

4.1. Automata over Infinite Trees: 1-way deterministic Buchi 25

4.2. Automata for the Chase 26

4.3. Automata over Infinite Trees: 2-way alternating Buchi 27

4.4. Automata for single-headed LTGDs 29

4.5. Time Complexity 30

4.6. Relation to Chase Compatibility 30

5. Weaker notions of CQ-equivalence **31**

5.1. Query entailment for the chase 31

5.2. Weaker notions of CQ-equivalence 33

6. Conclusions **35**

6.1. Summary 35

6.2. Related work 35

6.3. Future directions 35

1. Introduction

Logical constraints can be used to represent these relationships, telling how the target must relate to the source. Schema mappings are a crucial building block in several areas of database research, including data exchange[4], where we start with data structured under a source schema and we aim to create a target schema that best reflects the input data, and data integration[5], where we provide a unified view of all the data from different sources.

1.1. Motivation

In order to describe the relationship between source and target schemas, we use sets of constraints. Naturally, there has been a lot of research on manipulating schema mappings (with operations such as inverse and composition), but this project is focused on the optimization aspect. This consists of the removal of redundant constraints, an operation which can speed up drastically the computation of our target instances.

Example 1: Starting with source database $D_0 = \{R(1,2)\}$ and constraints $\Sigma = \{R(x,y) \rightarrow S(y,x),\ S(x,y) \rightarrow R(y,x)\}$, we would obtain the target instance $D_0 \cup \{S(2,1)\}$, but we can see that the second constraint in Σ does not bring any contribution to this, so we could get rid of it.

Using this optimization procedure, we ultimately want to tell if schemas are equivalent (different notions of equivalence will be discussed later). If we possess a repository of schemas which already have some established equivalences between them, this makes querying them more efficient. Consequently, it is important to test any new equivalences that would appear from updating this repository.

1.2. Contributions

The main goal of this project is to study the relationship between types of equivalences of schema mappings. We will also prove decidability of equivalence testing for particular (and very used in the literature) classes of constraints. This will be done using automata over trees that encode the source instances. Where this decidability has already been proven, our aim is to obtain a better time/memory complexity. In the cases where undecidability has been established, we wish to restrict the class of constraints until the problem becomes decidable.

1.3. Challenges

The first parts of this project involved getting familiar with the already existing related work, most of it treating similar problems with very different approaches. A crucial concept throughout the project will be the chase procedure, which has been defined in many places and in various ways, therefore trying to encode its structure using graph theory and automata was a big stepping stone. Another challenge arose from the introduction of two new types of equivalences between schemas and our attempt to fit them in the already existing picture of schema equivalences.

1.4. Structure of the Report

The rest of the report is structured as follows: an introduction of the notations and concepts that will be constantly used throughout the project is presented in Section 2. In Section 3 we introduce the types of equivalences of schema mappings that will be studied, together with how they are related and with decidability results. Section 4 will introduce an approach based on automata which will prove an important result from Section 3. Section 5 will relax one notion of equivalence and will contain decidability results as well. Finally, in Section 6, we present the conclusions, related work and future directions for this project.

2. Preliminaries

2.1. Relational Database Schema

A *relational schema* S consists of a set of distinct relation definitions (R_1, R_2, ..., R_n), each with an associated "arity". A *relation definition* (also called *table*) consists of its relation name R_i and a collection of attribute names paired with their types (A_1 : $type_1$, ..., A_n : $type_n$). Since the type of the attributes is not impactful for this project, and for simplicity of definitions, we will mostly use integer types.

A *tuple* (also called *fact*) for a given relation definition R_i is a function that assigns to each attribute A_i one value from the domain given by its type.

A *primary key* for a table R_i is a set of attributes that uniquely determines a tuple in any relation instance (each attribute in the primary key will be underlined).

A *database instance* D_0 over a schema S consists of one relation instance $D_0(R_i)$ for each relation definition R_i. A *relation instance* for R_i is a set of tuples for R_i such that no two tuples agree on all the attributes from the primary key of the table.

Example 2: Schema S contains information about a work environment, with one table *EmployeeInfo(<u>EmployeeID</u>, Name, Job, Salary)* and one table for structural organization *WorksFor(<u>EmployeeID</u>, <u>BossID</u>)*. The presented database instance only consists of a relation instance for *EmployeeInfo* and has three tuples, with primary key {EmployeeID}.

EmployeeID	Name	Job	Salary
81	Gilbert Marshall	Reporter	40000
209	Jane Hunt	Lawyer	45000
333	Neville Barton	Lawyer	40000

2.2. First-order Logic

Throughout the project we will mostly use *function-free first-order logic* for describing our relation instances, constraints and queries. A *signature* (also called *vocabulary*) for this logic consists of a finite collection of constant symbols and a finite collection of *relations* (also called *predicates*), each having a given *arity*. We can therefore view a relational schema as a signature, and we will use both terms interchangeably.

Syntax : A first-order formula is built from *atomic formulas*, which can either be *relational atoms* R($\vec{t}$), where R is a relation and $\vec{t}$ consists of *terms* (constants or variables), or an equality between two terms. From this, we build formulas inductively using boolean operators ($\wedge, \vee, \neg$) and quantifiers ($\forall, \exists$).

$$\varphi \ ::= \ R(\vec{t}) \mid t_i = t_j \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \forall x\, \varphi \mid \exists x\, \varphi$$

A variable x occurring in a formula φ is *bound* if it is in the scope of a quantifier. Otherwise, x is *free*.

Given a signature σ, we define a *structure* over σ to consist of a *domain*, an interpretation for each relation in σ as sets of tuples with values from the domain (must preserve the arities), and also an interpretation for each constant symbol in the domain. Thus, we can view an instance I over schema Sch as a structure (Adom(I), R_1, ..., R_n), where Adom(I) is the set of values that appear in I and is called the *active domain* of I.

Semantics : For a first-order logic formula $\varphi(\vec{x})$, structure M and function σ assigning each variable from $\vec{x}$ to a value in domain(M), we can define inductively on the structure of φ the notion of *satisfiability* M, $\sigma \models \varphi(\vec{x})$:

- M, $\sigma \models$ R(x_1, ..., x_n) iff $\langle\sigma(x_1), ..., \sigma(x_n)\rangle \in$ M(R);
- M, $\sigma \models x_1 = x_2$ iff $\sigma(x_1) = \sigma(x_2)$;
- M, $\sigma \models \varphi_1 \wedge \varphi_2$ iff M, $\sigma \models \varphi_1$ and M, $\sigma \models \varphi_2$ (analog for $\vee$ and $\neg$);
- M, $\sigma \models \exists$ x φ iff there is c of type(x) such that M, $\sigma \cup \{x \mapsto c\} \models \varphi$ (analog for $\forall$).

Example 3: Considering the instance from Example 2, the formula
φ(x, y) = EmployeeInfo(81, x, y, 45000) is not satisfiable by any variable binding, but the formula φ'(x, y) = EmployeeInfo(81, x, "Reporter", y) is satisfied by the mapping
σ = {x $\mapsto$ "Gilbert Marshall", y $\mapsto$ 40000}.

2.3. Queries

A *query* is a function mapping instances D_0 over relational schemas S to relational instances of a fixed relation. If the query has n free variables, then the resulting instance will be n-ary. A *Boolean query* has no free variables and maps instances to either True or False. We will define queries using the function-free first-order logic from above.

To define the particular class of queries that interests us in the project, we recall the fragment of first-order logic called *positive existential* ($\exists^+$) *first-order logic*, which does not allow the $\neg$ and $\forall$ operators.

Conjunctive Queries (CQs) are a fragment of $\exists^+$ first-order logic of the form:

$$Q(\vec{x}) = \exists \vec{y}\, \gamma(\vec{x},\, \vec{y})$$

where $\gamma(\vec{x},\, \vec{y})$ represents a **conjunction** of relational atoms that use free variables from $\vec{x}$ and bound variables from $\vec{y}$.

Given instance D_0 and query $Q(\vec{x})$, a *homomorphism of Q in D_0* (also called *satisfying assignment* or *match*) is a function mapping variables from $\vec{x}$ to values in the active domain of D_0 that makes Q hold in D_0.

Example 4: The query Q(x, y) = $\exists z, t_1, t_2$ EmployeeInfo(x, y, z, t_1) $\wedge$ EmployeeInfo(x, y, z, t_2) will return the following table:

209	Jane Hunt
333	Neville Barton

Each tuple in the table is a homomorphism of Q in D_0. We omit the names of the attributes in this project, as we will only care about their position relative to a particular relation name.

A *homomorphism* h *between two structures* D_0 and $D_0^{'}$ (over the same signature) is a function from the active domain of D_0 to the active domain of $D_0^{'}$ such that for every relation R and tuple $\langle t_1, ..., t_n \rangle$, we have:

$$D_0 \models R(t_1, ..., t_n) \Rightarrow D_0^{'} \models R(h(t_1), ..., h(t_n))$$

2.4. Dependencies

Dependencies are integrity constraints over relational database schemas. Intuitively, they are rules to be fulfilled by instances over schemas. Given instance D_0 and set of dependencies Σ, we say that D_0 is *consistent* with Σ if it satisfies all constraints in Σ. All dependencies that we will define will be from fragments of the first-order logic from above.

A *tuple-generating dependency* (TGD) φ has the form:

$$\forall \vec{x}\, \vec{y}\, \rho(\vec{x},\ \vec{y}) \rightarrow \exists \vec{z}\, \gamma(\vec{y},\ \vec{z}) \tag{1}$$

where ρ and γ are conjunctions of atoms with arguments from $\vec{x}$ and $\vec{y}$, respectively $\vec{y}$ and $\vec{z}$. Since $\vec{y}$ appears in both sides, these variables will be called *exported variables*. We will refer to $\rho(\vec{x},\ \vec{y})$ as the *body* of the TGD and to $\gamma(\vec{y},\ \vec{z})$ as the *head*.

The intuition behind the "tuple-generating" aspect is that when we start with a database instance D_0 that does not initially satisfy a TGD φ (say we have $\rho(\vec{x},\ \vec{y})$ satisfied, but no $\vec{z}$ such that $\gamma(\vec{y},\ \vec{z})$), we can extend D_0 with new facts (mentioning new symbols), until we satisfy φ[1].

Example 5: Consider a database instance D_0 = {R(0,1), R(1,2), S(0,3)}. The TGD $\varphi_1 = \forall$x, y S(x,y) $\rightarrow \exists$ z R(x,z) holds, but the TGD $\varphi_2 = \forall$x, y R(x,y) $\rightarrow \exists$ z S(x,z) does not, because there is no z for x = 1, y = 2 such that S(1,z).

A *guarded tuple-generating dependency* (GTGD) has the form:

$$\forall \vec{x}\, \vec{y}\, R(\vec{x},\ \vec{y}) \wedge \rho(\vec{x},\ \vec{y}) \rightarrow \exists \vec{z}\, \gamma(\vec{y},\ \vec{z})$$

where R is a relation symbol that mentions all variables from $\vec{x}$ and $\vec{y}$ and is called the *guard* of the constraint. In Example 5, both TGDs are guarded, but for instance the TGD $\varphi = \forall$x, y, z R(x,y) $\wedge$ S(x,z) $\rightarrow \exists$t R(t,x) is not.

A *linear tuple-generating dependency* (LTGD) is a GTGD with only one relation symbol in the body[2]:

$$\forall \vec{x}\, \vec{y}\, R(\vec{x},\ \vec{y}) \rightarrow \exists \vec{z}\, \gamma(\vec{y},\ \vec{z})$$

A *schema mapping* M is defined by a triple $\langle$S, T, $\Sigma\rangle$ where S is the source schema, T is the target schema, and Σ is a set of dependencies. Intuitively, Σ maps instances of S into instances of T.

[1] the semantics for TGDs are inherited from first-order logic.

[2] some books call it linear guarded TGD, but we can see that all LTGDs are trivially guarded.

2.5. Open World Query Answering

As we have seen, a database instance D_0 might not be consistent with respect to a set of dependencies Σ. A natural goal here is to see how to "extend" D_0 with facts in order for the newly formed instance to satisfy Σ.

$D_0^{'}$ is a *super-instance* of D_0 if it contains all the facts from D_0. Given instance D_0 and set of constraints Σ, a *world* (also called *model*) of D_0 and Σ is a super-instance of D_0 that satisfies Σ. Given a query $Q(\vec{x})$, a *certain answer* to Q on D_0, Σ is a homomorphism h such that the tuple $h(\vec{x})$ is in every world of D_0 and Σ.

The *Open World Query Answering* problem is to decide, given tuple $\vec{t}$, query Q, instance D_0 and set of constraints Σ, whether $\vec{t}$ is a certain answer to Q on D_0, Σ. There is a distinction between the constrained and unconstrained versions of this problem, depending if we only take into consideration the finite super-instances or not, respectively. A similar problem is to determine all certain answers of Q, given D_0 and Σ.

We will use the notation $D_0 \wedge \Sigma \models Q$ to indicate that Q holds in all possible worlds of D_0 and Σ, where Q is a Boolean CQ.

A useful tool in this project is the correspondence between Boolean CQs and databases:

- **Canonical database**: given CQ Q, the canonical database CanonDB(Q) replaces each existential variable in Q with a constant and has one fact for each fact in the body of Q;

- **Canonical query**: given database D_0, the canonical query CanonQuery(D_0) replaces each constant from D_0 with an existential variable and has one atom in the body of the query for each fact in D_0.

2.6. Forward Chaining: The Chase

One common technique for dealing with the open world query answering problem is to derive, given an initial instance D_0 and set of constraints Σ, the facts that can be implied in all possible worlds.

Given the TGD φ we defined at (1), a *trigger* for φ is a homomorphism h of $\rho(\vec{x}, \vec{y})$ into D_0. Intuitively, a trigger is simply an assignment of $\vec{x}$ and $\vec{y}$ to values in Adom(D_0) such that $\rho(\vec{x}, \vec{y})$ holds in D_0. We say that the trigger is *active* if there is no witness $\vec{z}$ such that $\gamma(\vec{y}, \vec{z})$ holds in D_0, which means that the TGD constraint is not yet satisfied by D_0. In order to derive a world where φ is satisfied, we create **new** constants (known as *labelled nulls*) for $\vec{z}$ and we append the necessary facts to D_0, in order to have a witness for $\gamma(\vec{y}, \vec{z})$. We call this process a *chase step*, and the resulting super-instance will no longer have h as an active constraint. In order to obtain a super-instance which satisfies all constraints from Σ, we start from D_0 and make a chase step until there are no active triggers. This procedure will lead to a sequence of instances D_0, D_1, ..., D_n, ..., where each D_{i+1} is obtained from D_i by performing a chase step. This is called a *chase sequence* and it can be *terminating* if it is finite and the final instance has no active triggers, otherwise it is *non-terminating.*

For efficiency, we can also act in stages, solving multiple active triggers at once:

- Starting from D_0, apply (in parallel) a chase step for every active trigger and form a super-instance D_1;
- Apply (in parallel) a chase step for each active trigger in D_k and form a super-instance D_{k+1};
- Continue until there are no active triggers in (say) D_n

This is called a *maximal parallel chase sequence* and we call D_n the *parallel chase model.* In case that our process is infinite, we consider as our chase model the direct limit of the structures in it, in our case D_0, D_1, ..., D_n, ..., and we will denote this model as $\text{chase}_\Sigma(D_0)$.

Types of chase: In the literature, there are multiple ways to define chase sequences, depending on which active triggers are picked. We will mention a few:

- **oblivious chase**: for every pair $(\vec{a}, \vec{b})$ and TGD φ of the form from (1) with $\rho(\vec{a}, \vec{b}) \subseteq D_0$, apply a chase step to φ only if the same pair has not contributed to any previous chase step; therefore, add to D_0 a new set of atoms $\gamma(\vec{b}, \vec{c})$, where $\vec{c}$ is a fresh set of labelled nulls;
- **restricted chase**: (also called **standard**) this is a refinement of the previous chase, where we only apply a chase step if the TGD φ is not already satisfied by D_0 i.e.

there is no $\vec{a}'$ such that $\gamma(\vec{b}, \vec{a}') \subseteq D_0$;

- **super-oblivious chase**: in this variant, we apply chase steps to all matching bodies, no matter if the pair $(\vec{a}, \vec{b})$ has been previously used or not.

All chase methods can create infinite instances, but it is clear that the restricted chase creates the smallest instances out of the three, and the super-oblivious one the largest.

Example 6: Given the database D_0 from Example 4 and TGD φ_2, which is not initially satisfied, we can apply one chase step and add a fresh constant $null_1$ and the fact S(1, $null_1$) to D_0. Now, the new instance satisfies φ_2 and if we use the restricted or oblivious chase, we would stop here. For the super-oblivious chase, we would add facts S(1, $null_i$), for $i \geq 1$.

Properties: First of all, it is important to notice that the chase procedure will always output a possible world of D_0 and Σ, no matter if the procedure is terminating or not. This can easily be deduced from the fact that there will always be a chase step for each active constraint that occurs, which will automatically solve it.

Theorem 2.6.1 (Universality Theorem). Given database instance D_0 and set of constraints Σ, $chase_\Sigma(D_0)$ is a *universal model* for D_0 and Σ, meaning that for any world $D_0^{'}$ of D_0 and Σ, there exists a homomorphism from $chase_\Sigma(D_0)$ to $D_0^{'}$.

One issue with the chase procedure is that for general TGDs, the termination of the algorithm is undecidable. However, even though for guarded TGDs the chase might be non-terminating, we can decide when to "cut" the procedure, because after some point we no longer obtain new facts (only renamings of the variables).

This crucial theorem will justify the use of the phrase "the chase model", since it implies that for any two chase models, no matter the chase procedure used, there will be a homomorphism between them. Therefore, the chase model is *unique up to homomorphism.*

The main link to the query answering problem is the following proven result:

Corollary 2.6.2. Consider a set of TGDs Σ, any database instance D_0, the chase model $chase_\Sigma(D_0)$ and a set of values Base from D_0 and Σ. Then, for any CQ Q using only constants from Base, and any tuple $\vec{t}$ with values in Base:

$$\vec{t} \text{ is a certain answer for Q, } D_0, \Sigma \text{ iff } \vec{t} \text{ is in Q}(chase_\Sigma(D_0)).$$

It has been proven that the problem of finding certain answers is undecidable for the general class of TGDs, but for restricted versions (such as guarded TGDs) this problem is decidable even in the non-terminating case.

2.7. Tree-like Property of the Chase

An important result studied in [1] is that the chase procedure for GTGDs is "tree-like". We will present how this result works for LTGDs: intuitively, we show that the model can be "coded" as a tree.

Let us fix source instance D_0 and set of LTGD constraints Σ. Let the set of constants from D_0 be $\mathrm{Adom}(D_0) = \{c_1, ..., c_n\}$ and the relation names $(R_1, ..., R_m)$. Let $N = \{n_1, ..., n_i, ... \}$ be the set of labelled nulls that will be used in the chase procedure. We represent the chase model of D_0 and Σ as a labelled ω-tree over a set of predicates which will contain all possible facts, combining relation names R_i and arguments from $\mathrm{Adom}(D_0) \cup N$.

Example 7: Let $D_0 = \{R(a,b,c)\}$ and $\Sigma = \{\forall x,y,z\ R(x,y,z) \rightarrow \exists t\ R(y,t,x); \forall x,y,z\ R(x,y,z) \rightarrow \exists t\ S(z,t)\}$. It is clear that the chase procedure here will be non-terminating, since we will always have a new active trigger for each R-fact that is added.

Formally, we can represent the structure as an infinite labelled tree using the following procedure:

1. create a root node and label it with all the facts from D_0 and mark it as "unexplored";

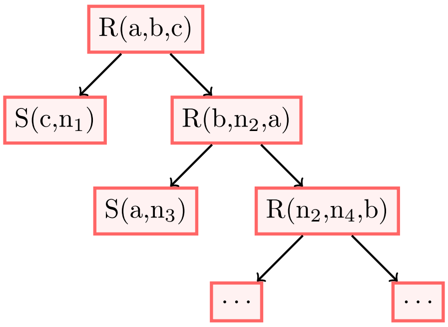
2. for each "unexplored" node v, let F_v be its label (i.e. its set of facts), and let Σ_v be all the active constraints φ from Σ such that there exists a fact in F_v that is a homomorphism for the body of φ;

3. for each such $\varphi \in \Sigma_v$, using a chase step we create new facts for each atom in the head of φ; for each satisfied head we create a new node (which will be a child of v) that is labelled with all the new facts (using fresh nulls from N);

4. mark node v as "explored" (the set of "unexplored" nodes acts as the frontier for a BFS-like algorithm);

5. repeat this process as long as there are "unexplored" nodes; this process might be non-terminating.

Example 8: The labelled tree corresponding to the chase from Example 7 will be:



We can see that N will be infinite, therefore in order to create a labelled ω-tree with finite set of predicates, we need to use a different approach, inspired from [1]:

Observation: For every labelled null[1], the subset of vertices that contain it is connected.

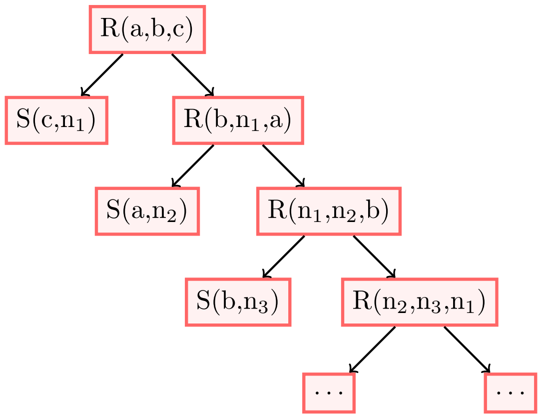
Proof: Suppose there exists a labelled null n such that the set of vertices that contain it is not connected. Therefore, there must be two nodes v_1 and v_2 that are labelled with facts that mention n for which no path that connects them exists. For each of the two nodes, we go up the tree as long as the parent of the current node mentions n. We end up with nodes w_1 and w_2 which are the ancestors of v_1 and v_2, respectively, that are the closest to the root and mention n. If $w_1 = w_2$, then there is a path between v_1 and v_2, which leads to a contradiction. If they are distinct nodes, then in both their cases n was introduced by an existential quantifier from an LTGD applied at their parent (w_1 and w_2 cannot be root since n must be a labelled null). This implies that the n introduced at w_1 is distinct from the one introduced at w_2, as we only introduce fresh nulls each time, which leads to a contradiction. $\square$

Following this observation, we could re-use labelled nulls in our representation, at the cost of changing their semantics:

- Let k be a natural number and $L_k = \{n_1, ..., n_{2k}\}$ be a set of labelled nulls (we will call them *local names* to make a distinction from the previous labelled nulls);
- When we need to create a new node labelled with $S(\vec{x}, \vec{y})$, where $\vec{x}$ are bounded and $\vec{y}$ are fresh (with the size of $\vec{y}$ being q), we use the first "available"[2] q local names.

[1] this also holds for constants from D_0, but not for constants introduced in Σ.
[2] an "unavailable" local name is one used in the parent node.

Example 9: We transform the structure from Example 8 using $k = 3$ and local names in $L = \{n_1, ..., n_6\}$:



The major difference is that if a local name appears in two different nodes that are not connected in the tree, then the two appearances do not represent the same value.

An important question is what value we should choose for k. Our goal is to minimize its value and consequently the number of possible predicates. As discussed in [1], the optimal value for k is the maximal arity (call it w) of any relation in D_0 or in Σ.

We need to have at least $k \geq w$, otherwise, if we encounter two facts $R(x_1, ..., x_w)$ and $R(y_1, ..., y_w)$, where all values mentioned are distinct, we cannot code the values in a consistent way with only 2k local names. In [1] we can see that having $k = w$ is actually sufficient for every instance that we will encounter.

A crucial point is that every chase structure for source instance D_0 and set of LTGDs can be represented using this new signature[1].

[1] in [1], we can find the definition of *k tree-code signature*, a generalization of what we presented here, which is required for a larger class of FO-formulas, called the *Guarded Fragment*.

3. Equivalences for the Chase and CQs

3.1. Definitions

Let us fix a source schema S, a target schema T, and let Σ_1 and Σ_2 be sets of TGD constraints from S to T. We recall three well-studied [6] equivalence types that will be important for the project, together with two more which are newly defined.

1. **Logical equivalence**: $\forall$ instances D_0, $D_0 \vDash \Sigma_1$ iff $D_0 \vDash \Sigma_2$;

2. **Chase compatibility**: $\forall$ source instances D_0, $\text{chase}_{\Sigma_1}(D_0) \vDash \Sigma_2$ and $\text{chase}_{\Sigma_2}(D_0) \vDash \Sigma_1$;

3. **Chase equivalence**: $\forall$ source instances D_0, $\text{chase}_{\Sigma_1}(D_0)$ and $\text{chase}_{\Sigma_2}(D_0)$ are homomorphically equivalent in both directions (via homomorphisms that preserve the constants in D_0);

4. **CQ-equivalence**: $\forall$ instances D_0, CQ $Q(\vec{x})$, tuple $\vec{t}$ from D_0, $D_0 \wedge \Sigma_1 \vDash Q(\vec{t})$ iff $D_0 \wedge \Sigma_2 \vDash Q(\vec{t})$;

5. **DE-equivalence**: $\forall$ source instances D_0, the models of D_0, Σ_1 and the models of D_0, Σ_2 coincide.

The goal of this section is to study the relationship between these types of equivalences and to look at decidability results.

3.2. Implications

For the already known equivalences, the following results hold:

Logical equivalence $\Rightarrow$ **DE-equivalence** $\Rightarrow$ **CQ-equivalence**

We claim the following implications using the new notions:

(A) Logical equivalence $\Rightarrow$ **Chase compatibility**:

Let D_0 be a source instance. By the Universality theorem for the chase, we know that $\text{chase}_{\Sigma_1}(D_0)$ is a universal model for D_0 and Σ_1, therefore $\text{chase}_{\Sigma_1}(D_0) \vDash \Sigma_1$. Using the assumption, we get that $\text{chase}_{\Sigma_1}(D_0) \vDash \Sigma_2$, too. The other branch is symmetrical.

(B) Chase compatibility $\Rightarrow$ Chase equivalence:

Since $\text{chase}_{\Sigma_1}(D_0) \models \Sigma_2$, we have that $\text{chase}_{\Sigma_1}(D_0)$ is a model (world) for Σ_2. Using the Universality of $\text{chase}_{\Sigma_2}(D_0)$ for D_0 and Σ_2, we get that there is a homomorphism from $\text{chase}_{\Sigma_2}(D_0)$ to $\text{chase}_{\Sigma_1}(D_0)$. The other direction is symmetrical.

(C) Chase equivalence $\Rightarrow$ CQ-equivalence:

We will do one direction of the iff statement: suppose Σ_1 and Σ_2 are chase equivalent and let instance D_0, CQ $Q(\vec{x})$ and tuple $\vec{t}$. We start with the assumption that $D_0 \wedge \Sigma_1 \models Q(\vec{t})$. From this, we know that $Q(\vec{t})$ is a certain answer to D_0 and Σ_1, therefore $Q(\vec{t})$ holds in all possible worlds for D_0 and Σ_1, one of which is $\text{chase}_{\Sigma_1}(D_0)$. By chase equivalence, we have a homomorphism from $\text{chase}_{\Sigma_1}(D_0)$ to $\text{chase}_{\Sigma_2}(D_0)$, which preserves every relation and the constants from D_0. Therefore, it preserves all the CQs, which means that $\text{chase}_{\Sigma_2}(D_0)$ satisfies $Q(\vec{t})$. Using Corollary 2.6.2, we get that $D_0 \wedge \Sigma_2 \models Q(\vec{t})$.

(D) CQ-equivalence $\Rightarrow$ Chase equivalence (in case of terminating chase):

Suppose Σ_1 and Σ_2 are CQ-equivalent and let source instance D_0. We will prove that if the chase procedure for the two sets of constraints terminates, there is a homomorphism from $\text{chase}_{\Sigma_1}(D_0)$ to $\text{chase}_{\Sigma_2}(D_0)$ (the other direction is symmetrical). Naturally, $\text{chase}_{\Sigma_1}(D_0) \models \text{CanonQuery}(\text{chase}_{\Sigma_1}(D_0))$. By CQ-equivalence, $\text{chase}_{\Sigma_2}(D_0) \models \text{CanonQuery}(\text{chase}_{\Sigma_1}(D_0))$. This means that there is a valuation h_{12} from all the existentially quantified variables in the query to values in the domain of $\text{chase}_{\Sigma_2}(D_0)$, such that all the facts from the body of the query are satisfied by $\text{chase}_{\Sigma_2}(D_0)$. Since $\text{CanonQuery}(\text{chase}_{\Sigma_1}(D_0))$ creates an existential variable for each value in $\text{chase}_{\Sigma_1}(D_0)$, we can create a homomorphism based on this correspondence and h_{12}. This homomorphism will preserve the constants in D_0.

The new results are:

Logical equiv. $\Rightarrow$ Chase compatibility $\Rightarrow$ Chase equiv. $\Rightarrow$ CQ-equiv.

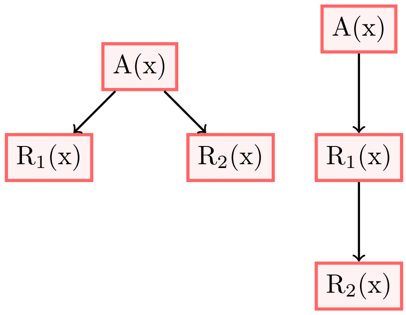
We will further investigate the behavior of the equivalences for the class of LTGDs, by giving counter-examples to the reverse direction of some implications:

(A') Chase compatibility $\not\Rightarrow$ Logical equivalence:

Example 10: Let S consist of unary predicate A(.), T consist of A(.) and unary predicates $R_1(.)$, $R_2(.)$. Let:

- $\Sigma_1 = \{A(x) \rightarrow R_1(x);\ A(x) \rightarrow R_2(x)\}$;

- $\Sigma_2 = \{A(x) \to R_1(x);\ R_1(x) \to R_2(x)\}$.



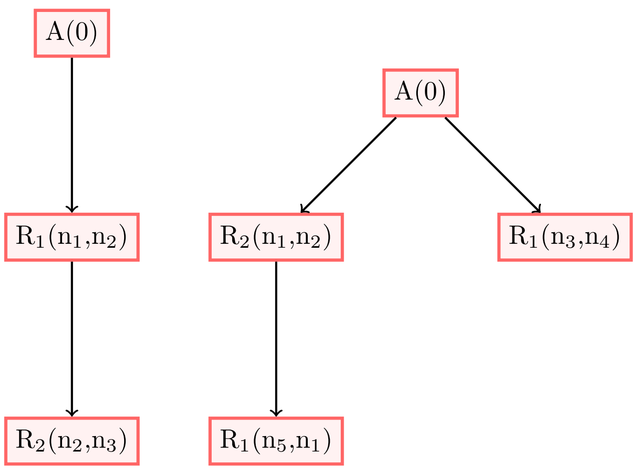
They are chase compatible, as for every source instance D_0 containing facts of type A(x), we will derive the facts $R_1(x)$ and $R_2(x)$. However, they are not logically equivalent, as we have the counter-example $D_0 = \{R_1(0)\}$, which has $D_0 \models \Sigma_1$ and $D_0 \not\models \Sigma_2$.

(B') Chase equivalence $\not\Rightarrow$ Chase compatibility:

Example 11: Let S consist of unary relation A(.) and T consist of A(.) plus two binary relations $R_1(.\ ,\ .)$ and $R_2(.\ ,\ .)$. Let:

- $\Sigma_1 = \{\forall\ w\ A(w) \to \exists\ x, y\ R_1(x,y);\ \forall\ x, y\ R_1(x,y) \to \exists\ z\ R_2(y,z)\}$;
- $\Sigma_2 = \{\forall\ w\ A(w) \to \exists\ x, y\ R_1(x,y);\ \forall\ w\ A(w) \to \exists\ y, z\ R_2(y,z);\ \forall\ y, z\ R_2(y,z) \to \exists\ x\ R_1(x,y)\}$.

They are chase equivalent and it is easy to create homomorphisms between the labelled nulls of the two structures. For chase compatibility, let $D_0 = \{A(0)\}$. The chases are:



where $\text{chase}_{\Sigma_2}(D_0)$ does not satisfy the second constraint from Σ_1, because of $R_1(n_3,n_4)$.

Q: Under what conditions does CQ-equivalence $\Rightarrow$ Chase compatibility hold?

Theorem 3.2.1: Let Σ_1 and Σ_2 be two sets of single-headed[1] LTGDs with the following properties:

1. every constraint mentions only binary predicates;
2. the body of each constraint has two distinct universally quantified variables;
3. the head of each constraint contains exactly one exported variable and one existential variable;
4. the set of constraints does not lead to a chase where there exist a value (constant or labelled null) that appears both in a node and in its grandchild.

Then, if Σ_1 and Σ_2 are CQ-equivalent, then they are also chase compatible.

Proof:

Claim 1: In any chase structure based on this class of LTGDs, if the facts from two distinct nodes share a value, then the relationship between them is either parent-child, or they are children of the same node.

Proof 1: Let n_1 and n_2 be such two nodes, sharing value v. Using the connected subset property of the chase, all nodes on the path between n_1 and n_2 share v. If the length of the path between them is greater than 2, then there must exist a violation of rule 4. If the distance is 2, then either one node is the grandchild of the other, which is not allowed, or they are the children of the same node. If the distance is 1, we get the parent-child relationship.

From the requirements imposed on the constraints, we can derive their form:

$$\varphi = \forall x\ y\ R(x,\ y) \rightarrow \exists z\ S(y,\ z)$$

$$\varphi = \forall x\ y\ R(x,\ y) \rightarrow \exists z\ S(z,\ y)$$

$$\varphi = \forall x\ y\ R(x,\ y) \rightarrow \exists z\ S(x,\ z)$$

$$\varphi = \forall x\ y\ R(x,\ y) \rightarrow \exists z\ S(z,\ x)$$

We will now argue by contradiction. Suppose that the two sets are CQ-equivalent, but there exists a source instance D_0 such that at least one of the necessary conditions does

[1]LTGDs with only one atom in the head : $\forall\ \vec{x},\vec{y}\ \mathrm{R}(\vec{x},\vec{y}) \rightarrow \exists\ \vec{z}\ \mathrm{S}(\vec{y},\vec{z})$

not hold, say chase$_{\Sigma_2} \not\models \Sigma_1$. We denote chase$_{\Sigma_i}$($D_0$) as CH_i, for i ∈ {1,2}. Therefore, there must exist constraint $\varphi \in \Sigma_1$ such that $CH_2 \not\models \varphi$. We will say that φ is of the form ∀x y R(x,y) → ∃ z S(y,z), but all other cases can be treated the same way. There must exist a witness R(a,b) in CH_2 such that there is no c with $CH_2 \models$ S(b,c). Let $F_0 = D_0$, F_1, F_2, ..., F_{k-1}, F_k = R(a,b) be the path (in CH_2) up to this witness.

Claim 2: There must exist a path $F^{'}_0 = D_0$, $F^{'}_1$, $F^{'}_2$, ..., $F^{'}_{k-1}$, $F^{'}_k$ in CH_1 that is homomorphic (preserving the constants in D_0) to the given path in CH_2.

Let $R_i(a_i,b_i)$ be the fact corresponding to F_i. We know from the given properties of the constraints that precisely one of a_i and b_i appears in F_{i-1} and the other one is a fresh null. We can form the canonical query Q of the path from CH_2, which will have as free variables all constants from D_0 and k universal variables, one for each fresh null introduced by each fact. The CQ-equivalence between Σ_1 and Σ_2 implies that CH_1 satisfies Q, which in turn implies the existence of facts $F^{'}_1$, $F^{'}_2$, ..., $F^{'}_{k-1}$, $F^{'}_k$ such that $F^{'}_i$ shares exactly one value with $F^{'}_{i-1}$, $1 \leq i \leq k$. We only remain to prove that these facts form a path in CH_1, as the part with the homomorphism is proven by the query satisfaction. We will build this path using strong induction on $0 < i < k$:

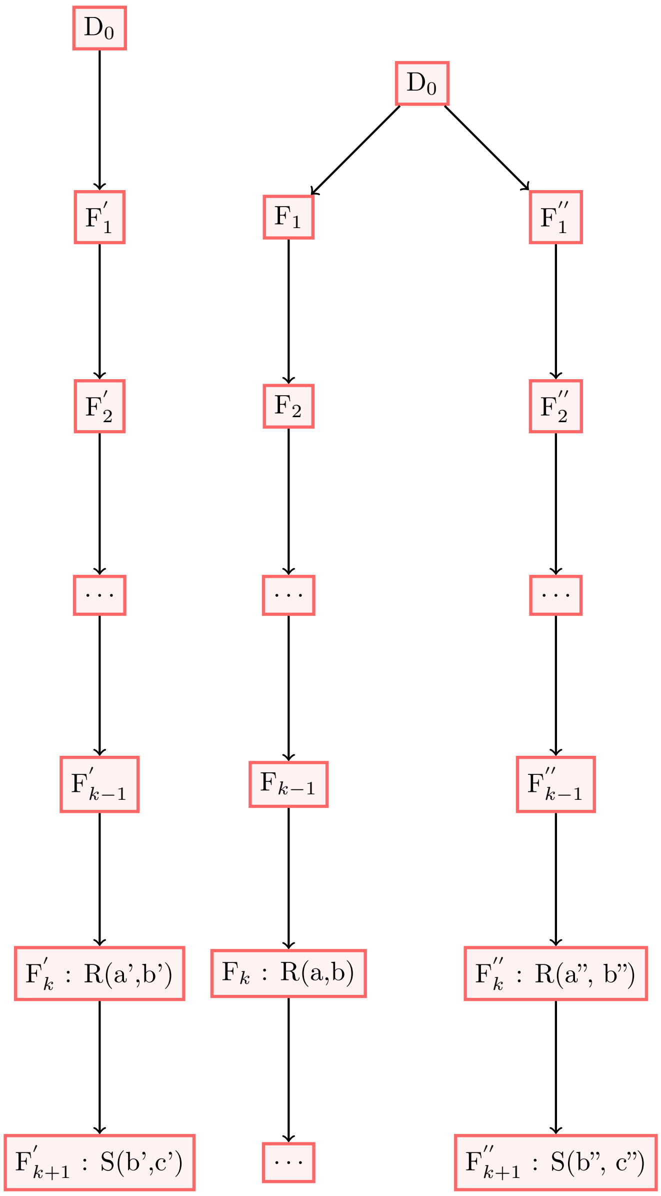
Base case: Because of Claim 1, $F^{'}_1$ must be a child of D_0;

Inductive step: Suppose that $F^{'}_0 = D_0$, $F^{'}_1$, $F^{'}_2$, ..., $F^{'}_{i-1}$, $F^{'}_i$ form a path. Because of Claim 1, $F^{'}_{i+1}$ is either a child of $F^{'}_i$, or a child of $F^{'}_{i-1}$, but using the connected subset property for the chase, the latter case will imply that $F^{'}_{i-1}$ contains the value shared between $F^{'}_i$ and $F^{'}_{i+1}$, which violates rule 4.

Therefore, $F^{'}_k$ will have a fact of the form R(a',b'). Because $\varphi \in \Sigma_1$, the chase procedure guarantees that there exist a child $F^{'}_{k+1}$ of $F^{'}_k$ that contains the fact S(b',c'), for a fresh null c'.

Now, we can use a similar approach for the path from CH_1 (augmented with $F^{'}_{k+1}$) to create the canonical query Q', which must be satisfied by CH_2. Analogously to what we did above and using the same reasoning as for Claim 2, there must exist a path $F^{''}_0 = D_0$, $F^{''}_1$, $F^{''}_2$, ..., $F^{''}_{k-1}$, $F^{''}_k$, $F^{''}_{k+1}$ in CH_1 that is homomorphic (preserving the constants in D_0) to the path from CH_1.

Because of this last homomorphism, $F^{''}_k$ must have a fact of the form R(a",b") and $F^{''}_{k+1}$ a fact of the form S(b",c"). The chase procedure guarantees that whenever we have a parent-child relationship, the child was created due to a constraint being applied, therefore we must have a constraint of the same form as φ in Σ_2, but this leads to a contradiction, as F_k could no longer be a witness that φ does not hold in CH_2.



3.3. Decidability for Equivalences

(A) Logical equivalence:

We will go through a fragment of first-order logic called the *Guarded Fragment*, where the formulas are of the form:

$$\varphi(\vec{x}) ::= True | False | R(\vec{t}) \mid t_i = t_j \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \forall\vec{y}\, R(\vec{x},\vec{y}) \rightarrow \varphi \mid \exists\vec{y}\, R(\vec{x},\vec{y}) \wedge \varphi$$

In the existential and universal cases, the free variables of the inner φ are required to be from $\vec{x} \cup \vec{y}$ and the R($\vec{x}$,$\vec{y}$) atom is called the *guard atom*.

It has been proven that we can construct automata for these formulas in order to decide their satisfiability: see, for example, [1]. With this, we can check if two sets of GF formulas are logically equivalent. We can use this approach for GTGD formulas, even when they are not in GF:

$$\varphi = \forall \text{x, y R(x,y)} \rightarrow \exists \text{ z S(x,z)} \wedge \text{T(y,z)}$$

is not in GF, but we can convert it to GF using a fresh Dummy relation name:

$$\forall \text{x, y R(x,y)} \rightarrow \exists \text{ z Dummy(x,y,z)}$$

$$\forall \text{x, y, z Dummy(x,y,z)} \rightarrow \text{S(x,z)}$$

$$\forall \text{x, y, z Dummy(x,y,z)} \rightarrow \text{T(y,z)}$$

These formulas are all in GF, and we can do this for any GTGD.

Consequently, **logical equivalence for GTGDs is decidable**.

(B) Chase compatibility: This decidability result will be studied in the next chapter, and will require a method of deciding an entailment of the form $\text{chase}_\Sigma(\text{D}_0) \vDash \varphi$.

(C) CQ-equivalence: Under the conditions imposed in Theorem 3.2.1, CQ-equivalence is the same as chase compatibility.

4. Chase Entailment for LTGDs

In order to prove results for chase compatibility introduced previously, we will use the following:

Theorem 4.0.1. For a fixed instance D_0, Σ a set of single-headed LTGDs and φ a single-headed LTGD, we can decide $\text{chase}_\Sigma(D_0) \vDash \varphi$.

The approach inspired from [1] follows the plan:

- Show that each such chase structure can be represented by a tree-like model (a structure that can be "coded" in a tree) - Section 2.7;
- Create an automata that will accept trees that code the given chase structure - Section 4.2;
- Create an automata that will accept only trees that satisfy $\neg\varphi$ - Section 4.4;
- Intersect the two automata and check for non-emptiness - Section 4.5.

4.1. Automata over Infinite Trees: 1-way deterministic Buchi

We define the type of automaton that is capable of representing the chase model of D_0 and Σ: let us fix a maximal outdegree r for the trees that will be considered as input from now on.

Definition 4.1.1: A 1-way deterministic B*ü*chi tree automaton is a 5-tuple
$\mathcal{A}$ = <Q, Alph, δ, I, F>, where:

- Q is a finite set of *states*;
- Alph is an input *alphabet*;
- δ : Q $\times$ Alph $\rightarrow \bigcup_{1 \leq i \leq r}$ Q^i is the *transition function*;
- I $\subseteq$ Q is the set of *initial* states;
- F $\subseteq$ Q is the set of *final* (also called *accepting*) states.

It takes as input an infinite Alph-tree and processes it top-down. A *run* assigns states from Q to the nodes of the tree. A *successful* (or *accepting*) run must satisfy the following conditions:

- The root of the tree must be assigned to an initial state from I;
- If a node v labelled with τ and children $v_1, ..., v_n$ is assigned to state q and we have $\delta(q, \tau) = (q_1, ..., q_n)$, then child i must be assigned to state q_i, for $1 \leq i \leq n$;
- for every path π of the tree, there are infinitely many nodes assigned to accepting states.

The language of such an automaton is formed from the trees for which there exist a successful run. An important property that will be needed from these automata is that the emptiness problem is decidable in PTIME, as it is done via reachability analysis. This is also true for the 1-way nondeterministic case.

4.2. Automata for the Chase

We will use a 1-way deterministic B*ü*chi tree automaton to input tree structures of the form presented at Section 2.7. We fix D_0, Σ and the local names $L = \{n_1, ..., n_{2k}\}$, where k is the maximal arity of any relation name from D_0 and Σ. We define the automaton $\mathcal{A}_{chase}(D_0, \Sigma)$ that will accept only structures that are homomorphic to $chase_\Sigma(D_0)$:

- $Q = \{q_0\} \cup \{q_{F(t_1,...,t_{arity(F)})} \mid F$ is a fact from D_0 or Σ, $t_1, ..., t_{arity(F)} \in Adom(D_0) \cup L\} \cup \{q_{Fail}\}$;
- $Alph = \{D_0\} \cup \{F(t_1, ..., t_{arity(F)}) \mid F$ is a fact from D_0 or Σ, $t_1, ..., t_{arity(F)} \in Adom(D_0) \cup L\}$;
- $\delta : Q \times Alph \to \bigcup_{1 \leq i \leq r} Q^i$ presented below;
- $I = \{q_0\}$;
- $F = Q \setminus \{q_{Fail}\}$.

We created states for each fact that we might see in the coding of a chase structure, and we accept trees based on their structure and labels:
1) At the root, we expect the label to be D_0, otherwise we go to a sink state q_{Fail}:

$$\delta(q_0, \tau) = \left\{ \begin{array}{ll} (q_{F_1}, ..., q_{F_m}), & \text{if } \tau = D_0 \\ (q_{Fail}), & \text{otherwise} \end{array} \right\}$$

The set of facts F_1, ..., F_m will be all the facts that can be generated from D_0, by matching the body of a constraint from Σ. In order to know what local names to expect, we impose a reasonable **policy** that in choosing a local name for a fresh variable, we always pick the first available local name from L, in the order of the indices. Intuitively, we only allow structures that have at the root the label D_0, and we then transition to all states that correspond to new facts derived from D_0 and Σ because this is what we expect to see on the next level.

2) At the "chase states" q_F (the ones that are neither q_0, nor q_{Fail}), we analogously to 1) define the set of new facts that should be generated from matching constraints in Σ with the body F, call it Σ_F = {F_1, ..., F_k}, again using the defined **policy**:

$$\delta(q_F, \tau) = \left\{ \begin{array}{ll} (q_{F_1}, ..., q_{F_k}), & \text{if } \tau = \{F\} \\ (q_{Fail}), & \text{otherwise} \end{array} \right\}$$

3) The final sink state will only transition to itself:

$$\delta(q_{Fail}, \tau) = (q_{Fail})$$

This automaton accepts structures isomorphic to $\text{chase}_\Sigma(D_0)$, because each state hard-codes the expected labels that we require. However, because of the **policy** that we imposed in order to reduce the transition space, we will not accept all structures that are isomorphic to the chase, but there will exist one that is accepted.

4.3. Automata over Infinite Trees: 2-way alternating Buchi

For the second part of the proof, we will require a more powerful automaton, which will have more flexibility in terms of searching over its input trees.

In order to decide whether a structure satisfies a single-headed LTGD, we have existential states that decide that a particular fact holds for at least one tuple and we also have universal states to decide if a particular fact holds for all possible tuples. This motivates the use of an alternating automaton.

Besides that, in the chase case we only traversed the input tree top-down, since only the last level of nodes was involved in the creation of new ones. However, in this case we are interested in traversing nodes in all directions, because we might find facts in other branches that have an impact in our decision. This motivates the use of a 2-way automaton (augmented with actions from $\text{Direction}_r = \{\text{Stay, Up}\} \cup \{\text{Down}_i \mid 1 \leq \text{i} \leq \text{r}\}$).

Our transition function assigns pairs of states and labels to **positive boolean combinations of propositions over Direction$_r$ × Q**. Intuitively, a transition to:

- (dir, q) will tell the automaton to go in direction dir in the input tree and transition to state q (base case);
- $\sigma_1 \wedge \sigma_2$ implies a universal choice, so both σ_1 and σ_2 need to happen;
- $\sigma_1 \vee \sigma_2$ suggests an existential choice, so at least one of σ_1 and σ_2 needs to happen.

As in [1], the set of positive boolean combinations of propositions for a set M is $\text{B}^+(\text{M})$.

Definition 4.3.1: A 2-way alternating B*ü*chi automaton is a 5-tuple
$\mathcal{A} = \langle \text{Q, Alph}, \delta, \text{q}_0, \text{F} \rangle$, where:

- Q is a finite set of *states*;
- Alph is an input *alphabet*;
- $\delta : \text{Q} \times \mathcal{P}(\Sigma) \rightarrow \text{B}^+(\text{Direction}_r \times \text{Q})$ is the *transition function*;
- $\text{q}_0 \in \text{Q}$ is the *initial* state;
- $\text{F} \subseteq \text{Q}$ is the set of *final* states.

The details for a *run* and for an *accepting run* are explained in detail in [1] and are an extension of the conditions from the 1-way deterministic case. The crucial part is that we require the node-child relationship to obey the transition relation, as before. As we did with the 1-way deterministic case, we are interested in the decidability of the non-emptiness problem, and for this type of automaton it has been proven that the problem is decidable in EXPTIME (in the number of states of the automaton).

4.4. Automata for single-headed LTGDs

Our goal is to create an automaton $\mathcal{A}_{\neg\varphi}$ that accepts all tree code chase structures where $\neg\varphi$ holds. For consistency, we will stick to the vocabulary defined at the previous section. Let us fix the single-headed LTGD φ ($\vec{y}$ represents the exported variables):

$$\varphi = \forall \vec{x}\, \vec{y}\, R(\vec{x},\ \vec{y}) \rightarrow \exists \vec{z}\, S(\vec{y},\ \vec{z})$$

$$\neg\varphi = \exists \vec{x}\, \vec{y}\, R(\vec{x},\ \vec{y}) \wedge \forall \vec{z}\, \neg S(\vec{y},\ \vec{z})$$

The automaton $\mathcal{A}_{\neg\varphi}$ will have:

- $\mathrm{Q} = \{\mathrm{q}_0\} \cup \{\mathrm{q}_{\vec{a};\vec{b}} \mid \vec{a},\, \vec{b}$ are tuples with elements in $\mathrm{Adom}(\mathrm{D}_0) \cup \mathrm{L}$ that we can form for $\mathrm{R}\} \cup \{\mathrm{q}_{True},\ \mathrm{q}_{False}\}$;
- Alph is the same as for the chase automaton, without D_0;
- q_0 is the initial state;
- $\mathrm{F} = \{\mathrm{q}_{True}\}$.

The transition function will be:

$$\delta(q_0, \tau) = \bigvee_{R(\vec{a};\vec{b}) \in \tau} (Stay, q_{\vec{a};\vec{b}}) \ \vee \bigvee_{d \in Direction_r} (d, q_0)$$

$$\delta(q_{\vec{a};\vec{b}}, \tau) = \left\{ \begin{array}{ll} (Stay, q_{True}), & \text{if } \vec{b} \cap \tau = \emptyset \\ (Stay, q_{Fail}), & \text{if } S(\vec{b}, \vec{c}) \in \tau \\ \bigwedge_{d \in Direction_r} (d, q_{\vec{a};\vec{b}}), & \text{otherwise} \end{array} \right\}$$

$$\delta(q_{True}, \tau) = (Stay, q_{True})$$

$$\delta(q_{False}, \tau) = (Stay, q_{False})$$

In order for $\neg\varphi$ to be satisfied by one of our coded structures, we will traverse the tree on nodes until we find a node where we can guess that the given pair is $(\vec{a},\ \vec{b})$, since the node is labelled with $\mathrm{R}(\vec{a},\ \vec{b})$. After this guess, we want to show that $\forall \vec{z}\, \neg S(\vec{b},\ \vec{z})$ holds in the tree structure. Starting from the node where we made the guess, we search for $\vec{b}$ in the tree and we have three cases:

1. If $\vec{b}$ is represented by the node and we have the label $S(\vec{b}, \vec{c})$, for some $\vec{c}$, then we fail, and our guess is wrong;

2. If no element from $\vec{b}$ is present in the tuple corresponding to the node, then we can accept, since if we keep searching for $\vec{b}$ from here, we will either go back to already visited nodes, or reach nodes where no element in $\vec{b}$ is represented; this can be explained with the connect subset property of the tree chase structures;

3. If some elements from $\vec{b}$ are present, we recurse on each direction of the tree.

4.5. Time Complexity

Intersecting the two automata that we defined, we obtain a 2-way alternating B$\ddot{u}$chi automaton that only accepts structures isomorphic to $\mathrm{chase}_\Sigma(D_0)$ that do not accept φ. Therefore, we can decide if φ holds in $\mathrm{chase}_\Sigma(D_0)$ by checking if the language of our automaton is $\emptyset$.

Let d be the cardinality of $\mathrm{Adom}(D_0)$ and w be the maximal arity. When we fix the arity w, the number of states becomes polynomial in w, and the non-emptiness check for the automaton will be done in EXPTIME. When this arity is not fixed, then the number of states is exponential in w, therefore the non-emptiness procedure will be 2EXPTIME.

4.6. Relation to Chase Compatibility

So far we proved that we can decide for a fixed D_0 the condition from the definition of chase compatibility, by applying Theorem 4.0.1. However, there are infinitely many possible instances. In order to overcome this issue, we claim:

Proposition 4.6.1: If for all single-fact instances D_0 we have that $\mathrm{chase}_{\Sigma_1}(D_0) \models \Sigma_2$ and $\mathrm{chase}_{\Sigma_2}(D_0) \models \Sigma_1$, then Σ_1 and Σ_2 are chase compatible.

Proof: Suppose the contrary, for instance that $\mathrm{chase}_{\Sigma_1}(D_0) \not\models varphi$, for $\varphi \in \Sigma_2$. Then, there is an witness atom that matches the body of φ such that no corresponding head exists. This atom was obtained from one fact F_0 from D_0 because of the nature of LTGDs, therefore we will also have that $\mathrm{chase}_{\Sigma_1}(F_0) \not\models \varphi$.

This result means that we only need to check chase compatibility of finitely many instances, which makes chase compatibility decidable in 2EXPTIME.

5. Weaker notions of CQ-equivalence

The goal of this section is to study the decidability of CQ-equivalence into more detail.

5.1. Query entailment for the chase

Let us fix a source instance D_0 and a set of LTGDs Σ. Let $M = \text{chase}_\Sigma(D_0)$ and we will use the notation M_k to represent the substructure of M such that in the tree representation of M presented at Section 2.7, we chop off the subtrees of M of depth bigger than k.

Definition 5.1.1: Two atoms of the form $R(c_1 \ldots c_n)$ and $R'(c'_1 \ldots c'_n)$ are *similar* with respect to D_0 if

- they use the same relation: $R = R'$;
- they have the same equalities between terms: $c_i = c_j$ iff $c'_i = c'_j$, for $1 \leq i, j \leq n$;
- they use the same constants from D_0 in the same positions: $c_i = d_0$ iff $c'_i = d_0$, for every $1 \leq i \leq n$ and $d_0 \in \text{Adom}(D_0)$.

Intuitively, there is a renaming for the non-constant terms in one atom that leads to the other atom. We can extend the notion to sets of facts $\{F_1, ..., F_n\}$ and $\{F'_1, ..., F'_n\}$ to mean that F_i and F'_i are similar, for $1 \leq i \leq n$. Two nodes in the tree representation are similar if their labels are similar.[1]

Observation 5.1.2: If two nodes from M are similar, then their subtrees are isomorphic. This holds because the isomorphism between the two subtrees is determined by the isomorphism between the two nodes. Additionally, the same rules will be applied to corresponding nodes.

Observation 5.1.3: For a sufficiently large k, every path of size k in M will contain two nodes that are similar with respect to D_0.

The proof of this relies on the fact that there are finitely many "similarity types", meaning that if we start with a set containing an atom $R(c_1, ..., c_n)$ that uses constants from $\text{Adom}(D_0)$ and local names, we can only add finitely many new atoms to the set until two of them will be similar. This bound k on the set of dissimilar facts will be exponential in

[1] in the LTGD case we will talk about nodes that are labelled with exactly one fact, excluding the case of the root.

the maximal arity of a relation from D_0 (called it w before). Therefore, if we have a path longer than k in the tree, by the Pigeonhole Principle we will encounter two similar nodes.

Proposition 5.1.4: If a Boolean CQ Q is entailed by M, there there exists an integer k such that Q holds in M_k. This integer k will be exponential in the sizes of Q and D_0.

Proof: We will begin with the proof for Q constituting of a single atom. We claim that if we choose the k from Observation 5.1.3, we will obtain in M_k all atoms that can be entailed by M. This holds as if we allow a path longer than k, we will encounter two similar nodes, and by Observation 5.1.2 the subtree coming from the nodes that follows later in the path will be isomorphic to the one from the node that comes earlier, so there will be no new atom in the subtree after this later node. Therefore, we can decide if Q holds in M by checking if the single atom in Q appears in any node from M_k.

For the proof where Q has a number of $q > 1$ atoms, let S_Q be a witness to that i.e. a set of q atoms in M and let h_Q be the corresponding homomorphism of Q into S_Q. We define the *closure* of S_Q in the tree structure $cl(S_Q)$ to be formed by closing S_Q under least common ancestors. Our goal will be to truncate $cl(S_Q)$ in a similar way as for the single-atom case. We say that two elements in $cl(S_Q)$ are *neighbors* if there is no element from S_Q in the path between them. We will repeat the following process:

1. Find two neighbors n_1 and n_2 (n_1 ancestor of n_2) such that there exists two distinct nodes c_1 and c_2 along their path that are similar with respect to the elements from the fact of n_1;

2. Replace every node from S_Q that comes after c_2 with the corresponding node that comes after c_1 to obtain a set $S_Q^{'}$ with homomorphism $h_Q^{'}$;

3. Repeat until we cannot find any candidates at 1 anymore.

Claim: After each step 2, we have that $h_Q^{'}$ is also a homomorphism of Q, where the distance between neighbors has decreased.

This is a result of Observation 5.1.2: the function $h_Q^{'}$ will be a homomorphism of Q because the nodes that come after c_2 will have similar nodes after c_1, thus by replacing them we still remain with a witness for Q. The distance clearly decreases since c_2 is a node that comes strictly after c_1 in the tree representation.

$$\boxed{n_1} \longrightarrow \boxed{\cdots} \longrightarrow \boxed{c_1} \longrightarrow \boxed{\cdots} \longrightarrow \boxed{c_2} \longrightarrow \boxed{\cdots} \longrightarrow \boxed{n_2}$$

Using Observation 5.1.3, we can limit the number of nodes between any two facts in the final S_Q, therefore the final size of k will grow proportionally to q, but will still be exponential in w.

5.2. Weaker notions of CQ-equivalence

Let us define new versions of CQ-equivalence for sets of LTGD constraints Σ_1 and Σ_2, which constrain the sizes of the source instances D_0 or the sizes of the CQs Q:

1. **j-CQ-equivalence**: $\forall$ instances D_0, CQ $Q(\vec{x})$ consisting of j atoms, tuple $\vec{t}$ from D_0, $D_0 \wedge \Sigma_1 \vDash Q(\vec{t})$ iff $D_0 \wedge \Sigma_2 \vDash Q(\vec{t})$;
2. **CQ-k-instance-equivalence**: $\forall$ instances D_0 containing k atoms, CQ $Q(\vec{x})$, tuple $\vec{t}$ from D_0, $D_0 \wedge \Sigma_1 \vDash Q(\vec{t})$ iff $D_0 \wedge \Sigma_2 \vDash Q(\vec{t})$;
3. **j-CQ-k-instance-equivalance**: $\forall$ instances D_0 of k atoms, CQ $Q(\vec{x})$ consisting of j atoms, tuple $\vec{t}$ from D_0, $D_0 \wedge \Sigma_1 \vDash Q(\vec{t})$ iff $D_0 \wedge \Sigma_2 \vDash Q(\vec{t})$.

Clearly, we get the following implications, for any j and k:

CQ-equivalence $\Rightarrow$ j-CQ-equivalence $\Rightarrow$ j-CQ-k-instance-equivalence

CQ-equivalence $\Rightarrow$ CQ-k-instance-equivalence $\Rightarrow$ j-CQ-k-instance-equivalence

Proposition 5.2.1: For sets of LTGDs Σ_1 and Σ_2, j-CQ-k-instance-equivalence is decidable.

Proof: Enumerate all possible queries $Q_j(\vec{x})$ of size j and instances D_k of size k, fix a tuple $\vec{t}$ from the fixed source instance, and then we require to decide if the Boolean CQ $Q_j(\vec{t})$ is entailed by D_k and Σ_i, i $\in$ {1,2}, which by the Universality Theorem of the chase can be reduced to instances of Proposition 5.1.4.

Proposition 5.2.2: The sets of LTGDs Σ_1 and Σ_2 are CQ-equivalent iff they are CQ-1-instance-equivalent.

Proof: We will only prove the non-trivial implication. Suppose that Σ_1 and Σ_2 are CQ-1-instance-equivalent and let D be an arbitrary instance, $Q(\vec{x})$ an arbitrary CQ, and $\vec{t}$ a tuple with constants from D. Suppose $D \wedge \Sigma_1 \vDash Q(\vec{t})$, i.e. $\text{chase}_{\Sigma_1}(D) \vDash Q(\vec{t})$. Our goal is to show that $\text{chase}_{\Sigma_2}(D) \vDash Q(\vec{t})$. Let us write explicitly the form of the query:

$$Q(\vec{t}) = \exists \vec{y}\ R_1(\vec{y_1},\vec{t_1}) \wedge ... \wedge R_n(\vec{y_n},\vec{t_n})$$

where $\vec{y_i} \subseteq \vec{y}$ and $\vec{t_i} \subseteq \vec{t}$, $1 \leq i \leq n$. Since $\text{chase}_{\Sigma_1}(D) \vDash Q(\vec{t})$, there must exist a valuation $\vec{a}$ for the existential variables in Q such that $\text{chase}_{\Sigma_1}(D)$ contains all facts $R_i(\vec{a_i},\vec{t_i})$. Each such fact was generated from exactly one fact from D (and its creation solely depends on that fact, independently on the other branches created from D), as the bodies of all LTGDs have a single atom. For each such fact $R_i(\vec{a_i},\vec{t_i})$, let D_i be the substructure of D containing just the fact that generated it. Also, for each atom in $Q(\vec{t})$, let

$$Q_i(\vec{t_i}) = \exists \vec{y_i}\ R_i(\vec{a_i},\vec{t_i})$$

Then, $\text{chase}_{\Sigma_1}(D_i) \vDash Q_i(\vec{t_i})$ and by CQ-1-instance-equivalence, $\text{chase}_{\Sigma_2}(D_i) \vDash Q_i(\vec{t_i})$. Thus, since $\text{chase}_{\Sigma_1}(D)$ is formed from all the chases executed on the single-fact instances, we can combine all the homomorphisms for the smaller queries to create a homomorphism for $Q(\vec{t})$, which proves that $\text{chase}_{\Sigma_2}(D) \vDash Q(\vec{t})$.

This result limits the space we need to search for CQ-equivalence, as now we only need to check for single-atom instances D_0.

6. Conclusions

6.1. Summary

The project has largely achieved the desired goals outlined in the project description. We have gradually introduced the reader to the background necessary for stating our contributions, which are:

- two new types of equivalences for schema mappings, together with their relation with respect to the already existing ones;
- study and prove decidability of chase compatibility for special classes of constraints;
- use automata approach to obtain lower complexity bounds for the decidability checking mentioned previously;
- study relaxed versions of CQ-equivalence and prove decidability results on them.

6.2. Related work

The closest reference to what is discussed in this project is [6], which analyses the decidability of the equivalence types defined in Section 3.1 (without the new ones). Another closely related work comes from [7].

6.3. Future directions

One extension for the decidability results for would be to relax the constraints imposed on the class of dependencies that we work with. It would be challenging to also generalize the use of automata in Section 4.4 for multi-headed LTGDs. Another question would be about the necessity of all the rules imposed in Theorem 3.2.1.

References

[1] Michael Benedikt, Michael Vanden Boom. *Decidable Logics via Automata* (pg. 30-46).

[2] Michael Benedikt, Balder ten Cate. *Modern Database Dependency Theory* (slides 79-84).

[3] D.S. Johnson, A. Klug. *Testing containment of conjunctive queries under functional and inclusion dependencies.*

[4] Ronald Fagin, Phokion G. Kolaitis, Renée J. Millerc, Lucian Popa. *Data exchange: semantics and query answering.*

[5] Maurizio Lenzerini. *Data Integration: A Theoretical Perspective.*

[6] Reinhard Pichler, Emanuel Sallinger, Vadim Savenkov. *Relaxed Notions of Schema Mapping Equivalence Revisited.*

[7] Ronald Fagin, Phokion Kolaitis, Alan Nash , Lucian Popa. *Towards a theory of schema-mapping optimization.*

[8] Efthymia Tsamoura, David Carral, Enrico Malizia, Jacopo Urbani.*Materializing Knowledge Bases via Trigger Graphs.*