Rheinisch-Westfälische Technische Hochschule Aachen

**Lehr- und Forschungsgebiet**
**Mathematische Grundlagen der Informatik**
**Prof. Dr. Grädel**

# Descriptive Complexity Theory for Constraint Databases

**Diplomarbeit**
**Stephan Kreutzer**

Januar 1999

Hiermit versichere ich, daß ich die Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 18.1.1999

I have many people to thank and acknowledge. Especially I would like to thank my advisor, Eric Rosen, for all the time he spent in supporting my thesis work.

# Contents

# Chapter 1

# Introduction

In the last 30 to 40 years databases have experienced an enormous development from mere extended file systems to the powerful systems we have today. One of the most important steps in this development was the introduction of the relational data model by Codd in 1970 [Cod70]. The advantages of this data model lie both within theory and practice. The intuitive and set-at-a-time user interface that relational database management systems supply makes them user friendly enough to be used even by non-programmers and non-specialists. On the other hand, there is a close relationship between relational databases and logical structures. This results in a solid theoretical foundation for relational databases in the form of finite model theory and descriptive complexity theory.

The relational data model proved adequate for most kinds of data and became standard in the 1980s. But with the increased power of modern computer hardware the need to store advanced data types like multimedia or geometrical data arose. To meet the new demands, other kinds of data models like nested relations or object oriented databases have been proposed. One of the disadvantages almost all of the new data models share, is the restriction to finite databases. Although this may seem to be sufficient and natural for most kinds of databases, it is unnatural for the storage of geometrical objects like circles or rectangles. For these kind of objects the said restriction leaves the user with the task of finding a way to code the objects and to formulate queries for the chosen encoding. This problem was the main motivation behind the introduction of constraint databases.

Constraint databases were first introduced by Kanellakis, Kuper, and Revesz in 1990 [KKR90]. They searched for a data model that allows the user to store an infinite amount of data. Of course this infinite set of objects has to be stored in a computer system and has therefore to be finitely representable. The main difference between this data model and most other data models is that the user does not have to be concerned with the way the data is represented but can think of the database as containing infinitely many objects.

Before giving the exact definition of constraint databases in the next chapter, their usefulness is demonstrated by an example. Imagine a database where geometrical shapes are stored. Databases like this occur, for example, in the field of geographical information systems where the shapes may represent the concentration of chemicals in a certain area. For simplicity suppose that the shapes consist only of circles. To store the said information in a relational database one could for example store the centre of the circle together with its radius, the chemical in consideration and its concentration. A user who wants to ask if there is a point where two special chemicals occur has to query the centers and radii of the two circles and calculate whether the sum of the radii exceeds the distance between their centers. This is very impractical for ad-hoc queries or queries invoked from script languages with limited expressive power.

Using constraint databases we would be able to store the circles as the set of points inside them. The query whether two of them intersect could then be formulated by asking if there is a point within both circles. Obviously this is much simpler and intuitive than the query above.

## 1.1 Relational databases from a logical viewpoint

An important field of research about relational databases is the study of database query languages. Questions arising here concern the expressiveness of query languages and the efficiency of their evaluation. To deal with such questions we model a relational database as a finite relational structure whose universe is the active domain of the database, that is the set of elements occurring in at least one database relation. The relations of the structure are the relations of the database. Once databases are modeled as logical structures we can consider logics as query languages. Furthermore, query languages correspond to logics, for example many commonly used query languages like SQL are essentially first-order logic or only slight extensions of it. Thus results about the complexity or expressiveness of logics yield also results about query languages.

Generally there are three possible relations between a logic $\mathfrak{L}$ and a complexity class $\mathcal{C}$. We say that $\mathcal{C}$ *contains* $\mathfrak{L}$, $\mathfrak{L} \subseteq \mathcal{C}$, if for each signature $\tau$ we have that every class of finite $\tau$-structures definable by an $\mathfrak{L}[\tau]$-sentence can be decided in $\mathcal{C}$. On the other hand we say that $\mathfrak{L}$ *contains* $\mathcal{C}$, $\mathcal{C} \subseteq \mathfrak{L}$, if for each signature $\tau$ we have that every class of $\tau$-structures decidable in $\mathcal{C}$ can be defined by an $\mathfrak{L}[\tau]$-sentence. We say that $\mathfrak{L}$ *captures* $\mathcal{C}$ if both is the case, that is $\mathfrak{L} \subseteq \mathcal{C}$ and $\mathcal{C} \subseteq \mathfrak{L}$.

On the one hand, the containment relation $\mathfrak{L} \subseteq \mathcal{C}$ gives an upper bound for the complexity of the evaluation of queries from $\mathfrak{L}$. On the other hand, it can be used to prove that certain properties cannot be queried by formulae from $\mathfrak{L}$. For example it is known that the transitive closure query is not in $\mathrm{AC}_0$. As first-order logic is contained in $\mathrm{AC}_0$, it cannot define transitive closure. The most interesting relation between a logic $\mathfrak{L}$ and a complexity class $\mathcal{C}$ is the capturing relation. If $\mathfrak{L}$ captures $\mathcal{C}$, then exactly the properties decidable in $\mathcal{C}$ can be defined by formulae from $\mathfrak{L}$. In a sense, $\mathfrak{L}$ is a precise logical description of $\mathcal{C}$. Therefore the field of research that investigates the relationship between logics and complexity classes is called *descriptive complexity theory*.

Starting with Fagin's theorem stating that NP is captured by existential second-order logic, logical descriptions have been found for many natural and important complexity classes. Unfortunately, most of these results are true only for the class of ordered structures. For example, Immerman [Imm86] and Vardi [Var82] proved that least fixed-point logic captures PTIME on the class of ordered structures. But on arbitrary structures, least fixed-point logic is strictly contained in PTIME. Similar results have been found for other complexity classes such as PSPACE as well. Table 1.1 gives an overview of important capturing results on ordered structures. Here *FO+DTC* stands for deterministic transitive closure logic, *FO+TC* for transitive closure logic, *FO+LFP* for least fixed-point logic and *FO+PFP* for partial fixed-point logic. *SO* denotes second-order logic and $\Sigma_1^1$ denotes its existential subset. See Chapter 2 for some more explanations and references.

| | | |
|---|---|---|
| *FO+DTC* | $=$ | LOGSPACE |
| *FO+TC* | $=$ | NLOGSPACE |
| *FO+LFP* | $=$ | PTIME |
| $\Sigma_1^1$ | $=$ | NP |
| *SO* | $=$ | PH |
| *FO+PFP* | $=$ | PSPACE |

Table 1.1: An overview of capturing results on finite ordered structures.

If order is omitted, most of the equalities in Table 1.1 change to inclusions. The only exceptions are $\Sigma_1^1$ and *SO*, because these logics allow the definition of order. On arbitrary structures we get the results summarized in Table 1.2.

| | | |
|---|---|---|
| FO | $\subset$ | $AC_0$ |
| *FO+DTC* | $\subset$ | LOGSPACE |
| *FO+TC* | $\subset$ | NLOGSPACE |
| *FO+LFPA* | $\subset$ | PTIME |
| $\Sigma_1^1$ | $=$ | NP |
| *SO* | $=$ | PH |
| *FO+PFP* | $\subset$ | PSPACE |

Table 1.2: An overview of complexity results on arbitrary finite structures.

For detailed information about descriptive complexity theory see for example the books by Immerman [Imm98] and Ebbinghaus and Flum[EF95]. Another source of information about these topics is the book by Abiteboul, Hull, and Vianu [AHV95], which presents the material from the perspective of database theory. An introduction to complexity theory can be found in the book by Papadimitriou [Pap94].

## 1.2 Aims of the diploma thesis

In the previous section we explained how, for finite databases, questions concerning the expressiveness of query languages and the efficiency of their evaluation can be answered using descriptive complexity theory. A natural questions is how the results and techniques developed there extend to classes of infinite but finitely representable databases. As an example of such databases we consider here the constraint databases introduced by Kanellakis, Kuper, and Revesz [KKR90]. In their framework a database consists of a

| | inequality | $(\mathbb{R},<)$ | $(\mathbb{R},<,+)$ | $(\mathbb{R},<,+,*)$ |
|---|---|---|---|---|
| FO | $\subset$ $\mathrm{AC}_0$ | $\subseteq$ $\mathrm{AC}_0$ | ? | NC |
| *FO+DTC* | $\subset$ LOGSPACE | $=$ LOGSPACE | undecidable | undecidable |
| *FO+TC* | $\subset$ NLOGSPACE | $=$ NLOGSPACE | undecidable | undecidable |
| *FO+LFP* | $\subset$ PTIME | $=$ PTIME | undecidable | undecidable |
| *FO+PFP* | $\subset$ PSPACE | $=$ PSPACE | undecidable | undecidable |

Table 1.3: Data complexity bounds for query languages on dense structures.

| | $(\mathbb{N},<_c)$ | $(\mathbb{N},<,+)$ | $(\mathbb{N},<,+,\cdot)$ |
|---|---|---|---|
| FO | $\subseteq$ LOGSPACE | $=$ PH | undecidable |
| *FO+DTC* | ex. pos. *FO+LFP* closed | undecidable | undecidable |
| *FO+TC* | | undecidable | undecidable |
| *FO+LFP* | undecidable | undecidable | undecidable |
| *FO+PFP* | undecidable | undecidable | undecidable |

Table 1.4: Data complexity bounds for query languages on discrete structures.

fixed structure, called the *context structure*, which is expanded by relations defined by quantifier-free formulae which are interpreted in the context structure. We give a formal definition of constraint databases and their query languages in Chapter 2.

There is a difference in the notion of evaluation complexity between finite and constraint databases which will be explained in detail in the next chapter. It turns out that the complexity of query languages for finitely representable databases depends heavily on the complexity of their representation. In the case of constraint databases the complexity of the representation depends on the complexity of the functions and relations which can be used in the formulae defining the database relations. Therefore the complexity and capturing results for logics on constraint databases depend on the context structure.

In the following chapters we present complexity results for many logics and context structures. We restrict our attention to arithmetical structures, that is structures whose relations and functions are among order, addition, and multiplication. On the side of the logics used as query languages the focus of our investigations is put upon extensions of first-order logic by recursion mechanisms like transitive closure or fixed-point induction. We do not consider second-order logics. Tables 1.3 and 1.4 give an overview of the results presented in the sequel. The *closed* entry for existential positive *FO+LFP* on the class of discrete order databases means that existential positive *FO+LFP* queries can be effectively evaluated in closed form. But nothing more is known about their complexity.

The first thing to notice is that, as for finite databases, we only get capturing results for ordered structures. But the complexity of query evaluation also depends heavily on whether the structures are densely or discretely ordered. The evaluation of queries on discretely ordered constraint databases is often much harder than on densely ordered databases. Whereas on dense order databases the various fixed-point extensions of first-order logic capture the same complexity classes as on finite databases, the complexity increases drastically if the order is discrete. Also the incorporation of recursion mechanisms leads to undecidable languages as soon as addition is available.

The NC result for first-order queries on the field of reals seems to be very promising for applications of constraint databases in the field of geometrical or spatial databases. Also the good complexity bounds for dense order databases makes them suitable as a basis for the implementation of, for example, temporal databases.

The following chapters are organized as follows. In the next chapter we give the formal definitions of constraint databases and query languages. In Chapter 3 we present some general methods to obtain complexity bounds for query evaluation. Chapter 4 gives an overview and proofs for most of the results summarized in Table 1.3 and 1.4. Chapters 5 and 6 focus on two particular context structures, namely dense and discrete orders. In Chapter 5 we present a very general method which allows us to extend results on logics capturing complexity classes from the realm of finite ordered structures to constraint databases over dense linear orders. Chapter 6 then gives a detailed proof for the LOGSPACE data complexity bound of first-order queries on discrete order databases. We close with a short summary and a few remarks about open problems and related work.

# Chapter 2

# Preliminaries

In this chapter we give the formal definitions of constraint databases and query languages and of the notion of complexity used in the sequel. Before we give the definitions, we fix some notation used in the following chapters.

A *signature* $\tau := \{R_1, \dots, R_k, f_1, \dots, f_r, c_1, \dots, c_s\}$ is a set of relation symbols $R_i$, function symbols $f_i$, and constant symbols $c_i$. The arity of each relation symbol $R$ and function symbol $f$ is denoted by $ar(R)$ and $ar(f)$ respectively. $\tau$ is called *relational* if it contains only relation symbols. A *$\tau$-structure* $\mathfrak{A}$ consists of a *universe* $A$, a relation $R^{\mathfrak{A}}$ for each relation symbol $R \in \sigma$, a function $f^{\mathfrak{A}}$ for each function symbol $f \in \sigma$, and a constant $c^{\mathfrak{A}}$ for each constant symbol $c \in \sigma$. A $(\sigma \cup \tau)$-structure $\mathfrak{B}$ with universe $B$, where $\tau$ and $\sigma$ are disjoint signatures, is called a *$\sigma$-expansion of* $\mathfrak{A}$, if $A = B$ and for every symbol $s \in \tau$ $s^{\mathfrak{A}} = s^{\mathfrak{B}}$. The expansion is called *relational* if $\sigma$ is relational. We write $\mathfrak{B} := (\mathfrak{A}, R_1^{\mathfrak{B}}, \dots, R_k^{\mathfrak{B}})$ to indicate that $\mathfrak{B}$ is a $\{R_1, \dots, R_k\}$-expansion of $\mathfrak{A}$.

By $FO[\tau]$ we denote the set of first-order formulae over the signature $\tau$. Besides first-order logic we consider some extensions of first-order logic, namely (deterministic) transitive closure logic, *FO+(D)TC*, least or inflationary fixed point logic, *FO+LFP*, *FO+IFP*, and partial fixed point logic, *FO+PFP*. In Chapter 3 we also consider the infinitary finite variable logic $L^{\omega}_{\infty\omega}$. For a detailed description of those logics see for example [EF95]. Unless stated otherwise, whenever we speak of a logic we have one of these logics in mind.

We write $\varphi(x_1, \dots, x_k)$ for a formula with free variables among $x_1, \dots, x_k$. $\mathfrak{A} \models \varphi[a_1, \dots, a_k]$ means that $\mathfrak{A}$ is a model of $\varphi$ where the free variables $x_i$ are interpreted as $a_i$. For each formula $\varphi(x_1, \dots, x_k)$ we write $\varphi^{\mathfrak{A}}$ to denote the set $\{(a_1, \dots, a_k) \in A^k : \mathfrak{A} \models \varphi[a_1, \dots, a_k]\}$ of tuples satisfying $\varphi$ in $\mathfrak{A}$. We say that formulae *use parameters from* $A$ if not only constant symbols from $\tau$ but also constant symbols $c_a$, for each $a \in A$, can occur in them.

## 2.1 Constraint databases

The basic idea in the definition of constraint databases is to allow infinite relations which can be finitely represented. In the framework introduced by Kanellakis, Kuper, and Revesz [KKR90] the relations are represented by quantifier-free first-order formulae.

**Definition 2.1.** Suppose $\mathfrak{A}$ is a $\tau$-structure, called the *context structure*, $\tau' \subseteq \tau$ a signature, and $\varphi(x_1, \dots, x_n) \in FO[\tau']$ is a quantifier-free formula using parameters from $A$. An $n$-ary relation $R \subseteq A^n$ is *represented by $\varphi$ over $\mathfrak{A}$*, if for all $a_1, \dots, a_n \in A$,

$$\mathfrak{A} \models \varphi[a_1, \dots, a_n] \text{ iff } (a_1, \dots, a_n) \in R.$$

The formula $\varphi$ is called a *finite representation* of $R$. The set of finitely representable relations over $\mathfrak{A}$ is denoted by $\mathrm{Rel}_{fr}^{\tau'}(\mathfrak{A})$. $\tau'$ is called the *relation signature* whereas $\tau$ is called the *context signature*.

In most cases the relation signature and the signature of the context structure coincide. Therefore we omit the index $\tau'$ in $\mathrm{Rel}_{fr}^{\tau'}(\mathfrak{A})$ and agree that, unless stated otherwise, the signature of the context structure is used as relation signature.

**Definition 2.2.** Suppose $\mathfrak{A}$ is a $\tau$-structure and $\sigma$ a relational signature disjoint from $\tau$. A *$\sigma$-constraint database* is a $\sigma$-expansion of the context structure, where all $\sigma$-relations are finitely representable over $\mathfrak{A}$. The set of all constraint databases over a context structure $\mathfrak{A}$ is denoted by $\mathrm{Exp}_{fr}(\mathfrak{A})$. The relations in $\tau$ are called *context relations* whereas the relations in $\sigma$ are called *database relations*. $\sigma$ is called the *database signature*.

By definition, constraint databases are relational expansions of a context structure where all database relations are finitely representable. Note that the same relation can be represented in different ways, e.g. $\varphi_1 := x < 10 \wedge x > 0$ and $\varphi_2 := (0 < x \wedge x < 6) \vee (6 < x \wedge x < 10) \vee x = 6$ are different formulae but define the same relation.

**Definition 2.3.** Suppose $\mathfrak{A}$ is a context structure and $\mathfrak{B} := (\mathfrak{A}, R_1^{\mathfrak{B}}, \dots, R_k^{\mathfrak{B}})$ a constraint database over $\mathfrak{A}$. A set of formulae $\Phi := \{\varphi_{R_1}, \dots, \varphi_{R_k}\}$ is a *finite representation of* $\mathfrak{B}$, if each $\varphi_{R_i}$ is a finite representation of $R_i^{\mathfrak{B}}$ over $\mathfrak{A}$. $\mathfrak{B}$ is called the *database represented by* $\Phi$. Two representations $\Phi$ and $\Phi'$ are *$\mathfrak{A}$-equivalent*, if they represent the same database over $\mathfrak{A}$.

In the following we often deal with algorithms taking constraint databases as inputs. The complexity of these algorithms will be measured in terms of the input size. Therefore the size of constraint databases has to be defined. Unlike finite databases, the size of constraint databases cannot be given in terms of the number of elements stored in them but has to be based on a representation of the database. Note that equivalent representations of a database need not to be of the same size. Thus the size of a constraint database cannot be defined independent of a particular representation.

**Definition 2.4.** Suppose $\mathfrak{B}$ is a constraint database and $\Phi$ a finite representation of $\mathfrak{B}$. The *size of* $(\mathfrak{B}, \Phi)$ is defined as the sum of the lengths of the formulae in $\Phi$.

In the following, whenever we speak of a constraint database $\mathfrak{B}$, we mean a constraint database with a particular representation $\Phi$. The size of $\mathfrak{B}$ is then defined as the size of $(\mathfrak{B}, \Phi)$. The definition of the size of a database corresponds to the following encoding of constraint databases on Turing machines.

**Definition 2.5.** Suppose $\mathfrak{B}$ is a constraint database and $\Phi$ a finite representation of $\mathfrak{B}$. Then $\mathfrak{B}$ is encoded on a Turing tape by the formulae of $\Phi$ representing the database relations.

In this framework the database relations are represented by quantifier-free formulae. Although it would be possible to base constraint databases on other logics, e.g. to represent the relations by arbitrary first-order formulae, the close relationship between constraint and finite databases present in this framework is a strong argument for choosing quantifier-free logic. We illustrate this by an example.

A finite relational database consists of a finite collection of relations. A relation again is a finite set of tuples. We can say that each tuple in a relation corresponds to an object of the real world. In analogy to the example given in the introduction, consider a database relation where circles are stored by their centre and radius. Each pair $(p, r)$, where $p$ is the centre and $r$ the radius, corresponds to a circle. In a constraint database, the circles would be represented by formulae of the form $(x+c_1)^2 \cdot (y+c_2)^2 \leq r^2$. Thus a tuple in the finite relation corresponds to a conjunction of atomic formulae in the constraint relation. Because of this correspondence Kanellakis, Kuper and Revesz called such conjunctions of atomic formulae *generalized tuples*. The finite relation as a set of tuples corresponds to the finitely representable relation as a finite set of generalized tuples. The formula for such a set of generalized tuples is a disjunction of conjunctions of atomic formulae, that is a quantifier-free formula in disjunctive normal form.

Some authors require that the formulae defining finitely representable relations are always given in disjunctive normal form. We will not be so strict and allow arbitrary quantifier-free formulae. Clearly, this does not increase the expressiveness of the finite representations but in some cases makes the representation of relations more intuitive and can also decrease the space needed to store the database.

## 2.2 Constraint queries

Now that constraint databases have been defined, we have to explain how they can be queried.

**Definition 2.6.** Suppose $\mathfrak{A}$ is a $\tau$-structure and $\sigma$ a relational signature. A *constraint query* $Q : \mathit{Exp}_{fr}(\mathfrak{A}) \rightarrow \mathit{Rel}_{fr}(\mathfrak{A})$ is a mapping from $\sigma$-constraint databases over $\mathfrak{A}$ to finitely representable relations over $\mathfrak{A}$.

By definition, a constraint query is purely semantical. In order to allow a user to query a database, we have to define query languages, that is, a syntactic way to define queries. To do this we require the context structure $\mathfrak{A}$ to admit quantifier elimination. This means that every first-order formula $\varphi$ is equivalent in $\mathfrak{A}$ to a quantifier-free formula. Since we are not only interested in first-order logic but also in extensions of it, we use the term quantifier elimination in a broader sense.

**Definition 2.7.** Let $\mathfrak{L}$ be a logic and $\mathfrak{A}$ be a $\tau$-structure. $\mathfrak{A}$ admits *quantifier elimination for* $\mathfrak{L}$, if for every formula $\varphi(x_1, \ldots, x_k) \in \mathfrak{L}[\tau]$, there is a quantifier-free first-order formula

$\varphi'(x_1, \dots, x_k)$, so that for all $a_1, \dots, a_k \in A$,

$$\mathfrak{A} \models \varphi[a_1, \dots, a_k] \quad \textit{iff} \quad \mathfrak{A} \models \varphi'[a_1, \dots, a_k].$$

In the next definition it is explained how queries can be defined by logical formulae.

**Definition 2.8.** Suppose $\mathfrak{L}$ is a logic and $\mathfrak{A}$ a $\tau$-structure admitting quantifier elimination for $\mathfrak{L}$. Let $\sigma$ be a relational signature and $\mathfrak{B}$ be a $\sigma$-database over $\mathfrak{A}$. For every formula $\varphi(x_1, \dots, x_k) \in \mathfrak{L}[\tau \cup \sigma]$ the *query* $Q_\varphi$ *defined by* $\varphi$ is defined as

$$\begin{array}{rlcl} Q_\varphi : & \mathrm{Exp}_{fr}(\mathfrak{A}) & \to & \mathrm{Rel}_{fr}(\mathfrak{A}) \\ & \mathfrak{B} & \mapsto & \{\overline{a} \in A^k \ : \ \mathfrak{B} \models \varphi[\overline{a}]\} \end{array}$$

The formulae $\varphi \in \mathfrak{L}[\tau \cup \sigma]$ are called *query formulae.*

The next lemma and its corollary show that the definition of a query $Q_\varphi$ for the query formula $\varphi$ is well-defined. Here and in the following chapters we need a way to "unfold" a database in a query formula, that is to combine the database and the query formula to a single formula. This is made precise in the following definition.

**Definition 2.9.** Suppose $\mathfrak{A}$ is a $\tau$-structure and $\mathfrak{B}$ a $\sigma$-constraint database over $\mathfrak{A}$. Let $\varphi$ be a query formula. The query formula $\varphi' := \mathit{unfold}(\varphi, \mathfrak{B})$ is defined as

$$\varphi' := \varphi[R_i/\varphi_{R_i}^{\mathfrak{B}}],$$

where each occurrence of a database relation symbol $R_i$ in $\varphi$ has been replaced by the formula $\varphi_{R_i}^{\mathfrak{B}}$ representing $R_i^{\mathfrak{B}}$ in $\mathfrak{B}$.

Clearly, $\varphi$ is equivalent in $\mathfrak{B}$ to $\varphi'$ and, as $\varphi'$ is a formula over $\tau$ alone, $\varphi^{\mathfrak{B}} = \varphi'^{\mathfrak{A}}$. The size of $\varphi'$ is $\mathcal{O}(|\varphi| \cdot |\mathfrak{B}|)$.

**Lemma 2.10.** *Suppose $\mathfrak{L}$ is a logic and $\mathfrak{A}$ is a $\tau$-structure. Let $\mathfrak{B}$ be a $\sigma$-database over $\mathfrak{A}$. If $\mathfrak{A}$ admits quantifier elimination for $\mathfrak{L}$ then also $\mathfrak{B}$ admits quantifier elimination for $\mathfrak{L}$.*

*Proof.* For each query formula $\varphi \in \mathfrak{L}[\tau \cup \sigma]$ let $\varphi'$ be the unfolded query $\varphi' := \mathit{unfold}(\varphi, \mathfrak{B})$. Because $\varphi'$ is a formula over the signature $\tau$ and $\mathfrak{A}$ admits quantifier elimination for $\mathfrak{L}$, $\varphi'$ is equivalent in $\mathfrak{A}$ to a quantifier-free formula. Therefore, as $\varphi$ and $\varphi'$ are equivalent, also $\varphi$ is equivalent to a quantifier-free formula. □

As a corollary of the lemma we get that the notion of query formulae defining queries is well-defined.

**Corollary 2.11.** *If $\varphi$ is a query formula, then $Q_\varphi$ defines a query.*

*Proof.* Let $\mathfrak{A}$ be a $\tau$-structure. By definition, $Q_\varphi$ maps a database $\mathfrak{B}$ over $\mathfrak{A}$ to the set $Q_\varphi(\mathfrak{B}) = \{\overline{a} \ : \ \mathfrak{B} \models \varphi[\overline{a}]\}$. To prove the corollary we show that $Q_\varphi(\mathfrak{B})$ is finitely representable. The preceding lemma proved that $\varphi$ is equivalent to a quantifier-free formula $\psi$ over the signature $\tau$. Thus $Q_\varphi(\mathfrak{B})$ equals $\{\overline{a} \ : \ \mathfrak{A} \models \psi[\overline{a}]\}$ and therefore $\psi$ is a finite representation of $Q_\varphi(\mathfrak{B})$ over $\mathfrak{A}$. □

Note the distinction between a query which is a mapping from databases to finitely representable relations and a query formula as a logical formula defining a query. Thus a query can be defined by various query formulae, whereas a query formula uniquely determines a query.

**Definition 2.12.** A *constraint query language* $\mathcal{L}$ for $\sigma$-databases over a $\tau$-structure $\mathfrak{A}$ consists of a set of logical formulae over a signature $\tau' \subseteq \tau \cup \sigma$, so that $\mathfrak{A}$ admits quantifier elimination for $\mathcal{L}$. $\tau'$ is called the *query signature*. Let $\varphi \in \mathcal{L}[\tau']$ be a query formula. The set $Q_\varphi(\mathfrak{B})$ is called the *answer* of the query $Q_\varphi$ over the database $\mathfrak{B}$.

Typically a logic like first-order logic or DATALOG will be used as query language, but also subsets of those logics like the set of positive or existential formulae are used.
Two different problems in connection with the evaluation of queries have been considered in the database literature.

**Definition 2.13.** Let $\mathfrak{A}$ be a context structure and $\mathcal{L}$ be a query language over $\mathfrak{A}$.

- The *evaluation problem* for a query formula $\varphi$ and a database $\mathfrak{B}$ over $\mathfrak{A}$ is defined as the problem of finding a finite representation of $Q_\varphi(\mathfrak{B})$.
- The *recognition problem* for $\varphi$ and $\mathfrak{B}$ is defined as the problem of checking whether a given tuple $\overline{a}$ is in $Q_\varphi(\mathfrak{B})$.

If $\varphi$ is a boolean formula, than the evaluation and the recognition problem coincide, because the answer of the query is either *true* or *false*. But if $\varphi$ is non-boolean, say a query in the variables $\overline{x}$, the two problems differ. In this case the recognition problem can again be seen as solving a boolean query, because checking whether a tuple $\overline{a}$ is in $Q_\varphi(\mathfrak{B})$ can be done by answering *true* or *false* to the query defined by $\varphi[\overline{x}/\overline{a}]$. In contrast, the evaluation problem requires the computation of a formula, namely the representation of the answer of $Q_\varphi(\mathfrak{B})$, and cannot be reduced to a boolean query.

It follows that only structures admitting quantifier elimination are suitable as context structures for constraint databases if the evaluation problem of non-boolean queries is under consideration. If we are not interested in formulae with free variables but only in boolean queries, the structure need not necessarily admit quantifier elimination but its theory must be decidable. An example of such a structure is Presburger arithmetic (see [Pre27]), that is, the natural numbers with addition.

**Note 2.14.** *The theory of any context structure has to admit effective quantifier elimination in order to allow query evaluation. If only boolean queries are considered, it suffices for the theory of the context structure to be effectively decidable.*

Typical questions that arise when dealing with constraint query languages are the complexity of query evaluation for a certain constraint query language and the definability of a query in a given language. We will be especially interested in the former question. Therefore we neglect the distinction between query formulae and the queries they define and call both just queries.

Throughout the following chapters we are mainly interested in the complexity of query evaluation for different logics. The complexity of query evaluation can be measured in different ways.

**Definition 2.15.** Let $\mathfrak{A}$ be a context structure and $\mathcal{L}$ be a query language. The complexity of the evaluation problem can be measured in three different ways.

- For a fixed query formula $\varphi \in \mathcal{L}$, the *data complexity* of the query $Q_\varphi$ is defined as the amount of resources (e.g. time, space, or number of processors) needed to evaluate the function that takes a representation $\Phi$ of a database $\mathfrak{B}$ to a representation of the answer relation $Q_\varphi(\mathfrak{B})$.
- For a fixed constraint database $\mathfrak{B}$ over $\mathfrak{A}$, the *query complexity of* $\mathfrak{B}$ is defined as the amount of resources needed to evaluate the function taking a query formula $\varphi$ to the representation of the answer relation $Q_\varphi(\mathfrak{B})$.
- If both, the database and the query, are variable, we speak of the *combined complexity.* It is defined as the complexity of the function taking the pair $(Q_\varphi, \mathfrak{B})$ to $Q_\varphi(\mathfrak{B})$.

When considering query evaluation for constraint query languages, there are different parameters to vary. The first and most important one is the context structure, which depends on the kind of data to be stored in the database. For example a geographical information system or software to manipulate geometrical figures may need the ordered field of reals $(\mathbb{R}, <, +, *)$ as context, whereas $(\mathbb{R}, <, +)$ might suffice for a CAD system. So far research has concentrated on these two structures and dense linear orders like $(\mathbb{R}, <)$.

The next parameter in consideration is the choice of the query language. To fix the query language the query signature as well as the query logic has to be chosen. Here first-order logic and DATALOG have been the logics of choice so far, where the query and the context signature coincide.

# Chapter 3

# Proof methods for constraint databases

In this chapter we investigate methods for proving complexity bounds or non-definability results for constraint query languages. In the first section we relate the complexity of query evaluation to the complexity of the theory of a context structure. In the section thereafter we show as an application of Ehrenfeucht-Fraïssé-games that the fixed-point logics introduced so far are too weak over the class of databases defined by inequality constraints over countable infinite sets to define all queries computable in LOGSPACE.

## 3.1 Obtaining complexity bounds

In this section we focus our attention on the question of how the complexity of the theory of a context structure and of quantifier elimination are related to the complexity of query evaluation. Throughout the rest of this chapter we consider first-order queries and the first-order theory of a context structure $\mathfrak{A}$.

We begin our investigations with the following lemma relating the complexity of $Th(\mathfrak{A})$ to the query complexity of boolean queries. The question how similar results can be found for data complexity is considered thereafter. The section is closed by the consideration of the data complexity of non-boolean queries.

**Lemma 3.1.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a function and $\mathcal{R}$ be a resource, e.g. time or space.*

*(i) If $\mathcal{R}(f(n))$ is a lower complexity bound for the theory of $\mathfrak{A}$ then it is also a lower bound for the query complexity of boolean queries.*

*(ii) If $\mathcal{R}(f(n))$ is an upper complexity bound of $Th(\mathfrak{A})$ then the query complexity of boolean queries is bounded from above by $\mathcal{R}(f(\mathcal{O}(n)))$.*

*Proof.*

*(i)* Cleary, a lower bound for the complexity of $Th(\mathfrak{A})$ is also a lower bound for the query complexity of boolean queries, because every sentence of the theory is also a boolean query on arbitrary databases over $\mathfrak{A}$.

*(ii)* Suppose $\mathfrak{B} = (\mathfrak{A}, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$ is a database over $\mathfrak{A}$. For each boolean query $\psi$ let $\psi' := \mathit{unfold}(\psi, \mathfrak{B})$ be the be the unfolded query as defined in Definition 2.9. The truth of $\psi'$ can be decided in $\mathcal{R}(f(|\psi'|))$. As the database is fixed, the size of $\psi'$ is $\mathcal{O}(|\psi|)$. Thus the truth of $\psi$ can be checked in $\mathcal{R}(f(\mathcal{O}(|\psi|)))$.

□

The preceding lemma shows that the complexity of the theory of $\mathfrak{A}$ yields the query complexity for boolean queries on constraint databases over $\mathfrak{A}$. It does not generally apply to data complexity, because if data complexity is in consideration the query and therefore the number of quantifiers is fixed, whereas the complexity of the theory often depends heavily on the number of quantifiers. To deal with data complexity we consider sentences with a fixed prefix length.

**Definition 3.2.** Let $k \in \mathbb{N}$ be an integer. The *$k$-dimensional theory $k$-$Th(\mathfrak{A})$ of* $\mathfrak{A}$ is the set of all sentences in prenex normal form with $k$ quantifiers which are true in $\mathfrak{A}$.

The next theorem proves that a lower data complexity bound for query evaluation can be obtained from a lower complexity bound for $k$-$Th(\mathfrak{A})$.

**Theorem 3.3.** *Suppose $k \in \mathbb{N}$ is fixed and $\mathcal{C}$ a lower bound for the complexity of $k$-$Th(\mathfrak{A})$. Then $\mathcal{C}$ is also a lower complexity bound for the data complexity of the query evaluation problem.*

*Proof.* To prove the lemma we reduce the decision problem for $k$-$Th(\mathfrak{A})$ to the query evaluation problem for databases over $\mathfrak{A}$. Every sentence in $k$-$Th(\mathfrak{A})$ has the form $\varphi := Q_1x_1 \ldots Q_kx_k\varphi'$, where $\varphi'$ is quantifier-free. Let $R_k$ be a $k$-ary relation symbol and $\psi_k$ be the query $\psi_k := Q_1x_1 \ldots Q_kx_kRx_1 \ldots x_k$. Obviously the size of the query depends only on $k$ and is therefore fixed.

For each $\varphi \in k$-$Th(\mathfrak{A})$, with $\varphi := Q_1x_1 \ldots Q_kx_k\varphi'$, let $\mathfrak{B}_\varphi$ be an $\{R\}$-database such that $R^{\mathfrak{B}_\varphi}$ is represented by $\varphi'$. The size of $\mathfrak{B}_\varphi$ equals the size of $\varphi'$ and thus the size of $\varphi$ minus a constant $c := 2k$. Clearly, $\varphi \in k$-$Th(\mathfrak{A})$ if and only if $\psi_k^{\mathfrak{B}_\varphi}$ evaluates to true. Thus the lower bound for $k$-$Th(\mathfrak{A})$ is also a lower bound for the data complexity of the evaluation problem. □

The preceding theorem proves that a lower bound for a $k$-dimensional theory also yields a lower bound for the data complexity of the query evaluation problem. An upper complexity bound for the data complexity of boolean queries can also be derived from the complexity of the $k$-dimensional theories, if the complexity of all $k$-dimensional theories is known.

**Theorem 3.4.** *Let $\mathcal{R}$ be a resource, e.g. time or space, and $f_k : \mathbb{N} \to \mathbb{N}$ be a function such that $\mathcal{C}_k := \mathcal{R}(f_k(n))$ is an upper complexity bound for $k$-$Th(\mathfrak{A})$. Then $\mathcal{R}(f_k(\mathcal{O}(n)))$ is an upper bound for the data complexity of all boolean queries with at most $k$ quantifiers.*

*Proof.* We prove the theorem by showing that every boolean query $\psi$ with $k$ quantifiers can be evaluated in $\mathcal{R}(f_k(\mathcal{O}(n)))$. Suppose $\mathfrak{B} := (\mathfrak{A}, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$ is a database. Let

$\psi' := unfold(\psi, \mathfrak{B})$ be the unfolded query as defined in Definition 2.9. As we consider data complexity, the query $\psi$ is fixed and therefore the size of $\psi'$ is $\mathcal{O}(|\mathfrak{B}|)$. Further, $\psi'$ has $k$ quantifiers and therefore the truth of $\psi'$ in $\mathfrak{A}$ and, with it, the truth of $\psi^{\mathfrak{B}}$ can be decided in $\mathcal{R}(f_k(|\psi'|)) = \mathcal{R}(f_k(\mathcal{O}(|\mathfrak{B}|)))$. □

Thus if the complexity of all $k$-dimensional theories is known, we can derive an upper data complexity bound for boolean queries. For example, if the complexity of all $k$-dimensional theories is PTIME, then also the data complexity of boolean queries is PTIME. Note the difference between the upper bound $\mathcal{C}$ for the data complexity of boolean queries and the complexity of the theory of $\mathfrak{A}$. In the calculation of the $\mathcal{C}_k$ the number of quantifiers is considered as being fixed. Thus $\mathcal{C}$ is derived from the complexity of fixed dimension theories whereas in the calculation of the complexity of $Th(\mathfrak{A})$ the number of quantifiers is variable. For example consider databases defined over the ordered field of reals. The complexity of the theory of real closed fields has a non-deterministic exponential lower time bound (see for example [HU79]). But we will see in Section 4.2.3 that the data complexity of first-order queries is NC.

One way to obtain lower bounds for fixed dimension theories is to study the complexity of prefix classes.

**Definition 3.5.** Suppose $\overline{Q} \in \{\forall, \exists\}^*$ is a finite sequence of quantifiers. The *prefix class* $[\overline{Q}]$ of $Th(\mathfrak{A})$ is defined as the set of all sentences in prenex normal form with quantifier prefix $\overline{Q}$ which are true in $\mathfrak{A}$.

Obviously a lower bound for the complexity of a prefix class of $\mathfrak{A}$ with $k$ quantifiers is also a lower bound for the complexity of the $k$-dimensional theory of $\mathfrak{A}$. We will use this in Section 4.3.2 to prove that for each level of the polynomial time hierarchy there is a first-order query in the context of the Presburger arithmetic whose data complexity is complete for this level.

So far all lemmas and theorems in this section dealt only with boolean queries. Clearly, the complexity of non-boolean queries is at least the complexity of boolean queries. Therefore the results for obtaining lower bounds can also be used for non-boolean queries. To obtain upper bounds, the study of the complexity of $Th(\mathfrak{A})$ is not sufficient. Instead we have to consider the complexity of quantifier elimination in $\mathfrak{A}$. The next theorem relates the complexity of quantifier elimination with the data complexity of first-order query evaluation.

**Theorem 3.6.** *A lower or upper complexity bound for the elimination of a fixed number of quantifiers is also a lower or upper bound for the data complexity of first-order queries.*

*Proof.* Clearly, an upper bound for the complexity of eliminating a fixed number of quantifiers is also an upper data complexity bound for first-order queries, because a quantifier elimination algorithm can also be used for query evaluation.
On the other hand, using the method presented in the proof of Theorem 3.3 one can easily show that the problem of eliminating a fixed number of quantifiers can be reduced to the evaluation problem for first-order queries. Therefore a lower complexity bound for the former problem is also a lower bound for the latter. □

In this section we obtained complexity bounds for problems connected to the complexity of the theory of the context structure. In the next section we consider methods to prove that a query language is too weak to define particular properties of databases.

## 3.2 Proving non-definability results

In [GS94] and [GS97a] Grumbach and Su examined which methods used to show non-definability results in classical or finite model theory still work for the class of finitely representable structures. They showed that the finitely representable model theory differs both from classical as well as finite model theory. Among the few methods which still work in the context of finitely representable structures are Ehrenfeucht-Fraïssé games and their variants, the pebble games. As an application of Ehrenfeucht-Fraïssé games we prove that, as in finite model theory, on unordered structures even such expressive logics as *FO+PFP* lack the power to count. Thus there are tasks which can be accomplished by LOGSPACE Turing machines which cannot be defined in *FO+PFP*.

To prove this we need some theorems well known from finite model theory (see for example [EF95]). The first theorem states that the fixed-point logics introduced so far are all contained in the infinitary logic $L^{\omega}_{\infty\omega}$.

**Theorem 3.7.**

*(i)* *FO+LFP* $\subseteq L^{\omega}_{\infty\omega}$.

*(ii)* *FO+PFP* $\subseteq L^{\omega}_{\infty\omega}$.

An implication of the theorem is that we can show non-definability results for the various fixed-point logics by showing that the properties in question are not definable in $L^{\omega}_{\infty\omega}$. To show non-definability results for $L^{\omega}_{\infty\omega}$we use a game theoretical characterization of $L^{\omega}_{\infty\omega}$-equivalence.

**Theorem 3.8.** *Let $\mathfrak{A}$ and $\mathfrak{B}$ be $\tau$-structures. The following is equivalent:*

*(i)* $\mathfrak{A} \equiv_{L^k_{\infty\omega}} \mathfrak{B}$.

*(ii) The duplicator wins the $k$-pebble game $G^k_\infty(\mathfrak{A}, \mathfrak{B})$.*

We use this to prove the following theorem.

**Theorem 3.9.** *On unordered finitely representable structures* LOGSPACE $\not\subseteq L^{\omega}_{\infty\omega}$.

*Proof.* Let $\mathfrak{A}$ be a structure with an empty signature whose universe is countably infinite and $\sigma := \{R\}$ be a database signature containing only one unary relation symbol. We use an Ehrenfeucht-Fraïssé game to show that there is no $L^{\omega}_{\infty\omega}$-sentence $\varphi$ which is true in a database $\mathfrak{B} = (\mathfrak{A}, R^{\mathfrak{B}})$ if and only if $R^{\mathfrak{B}}$ is infinite or of even cardinality. For the sake of contradiction suppose there would be such a sentence $\varphi \in L^{\omega}_{\infty\omega}$. Recall that $L^{\omega}_{\infty\omega}$ is defined as the union of $L^k_{\infty\omega}$ for all $k \in \mathbb{N}$. Thus there is a $k \in \mathbb{N}$ such that $\varphi \in L^k_{\infty\omega}$. Let $P$ be the set of constants occurring in $\varphi$. Consider two databases $\mathfrak{B}_1$ and $\mathfrak{B}_2$ where

$|R^{\mathfrak{B}_1}| = k$ and $|R^{\mathfrak{B}_2}| = k+1$ and no constant from $P$ occurs in $R^{\mathfrak{B}_1}$ or $R^{\mathfrak{B}_2}$. Obviously the duplicator wins the game $G^k_\infty(\mathfrak{B}_1, \mathfrak{B}_2)$, because, essentially, he has to ensure that whenever the spoiler places his pebble on an element of $R$ in one structure, he has to do so in the other structure, and whenever the spoiler places his pebble on an element which occurs as a constant in $P$, he has to do so as well. This is always possible because the spoiler can only place $k$ pebbles and both relations contain at least $k$ elements which occur not as a constant in $P$. Thus the duplicator wins the game $G^k_\infty(\mathfrak{B}_1, \mathfrak{B}_2)$ and, by Theorem 3.8, the structures $\mathfrak{B}_1$ and $\mathfrak{B}_2$ are $L^k_{\infty\omega}$-equivalent. Therefore either $\varphi$ holds in both structures or in none of them. Clearly, one of $R^{\mathfrak{B}_1}$ and $R^{\mathfrak{B}_2}$ is of even and the other of odd cardinality. Thus $\varphi$ cannot distinguish between relations of odd and even cardinality.
Now we show that a LOGSPACE Turing machine can decide whether a relation $R$ is finite and of even cardinality. Note that as $R$ is unary it is represented by a quantifier-free formula $\varphi_R$ with only one variable $x$. We define a set *unbounded*$(\varphi_R)$ by induction on the structure of $\varphi_R$.

- If $\varphi_R := x = c$, where $c$ is a parameter, then *unbounded*$(\varphi_R) := \emptyset$.
- If $\varphi_R := x \neq c$ then *unbounded*$(\varphi_R) := \{x \neq c\}$.
- If $\varphi_R := \varphi_1 \wedge \varphi_2$ then *unbounded*$(\varphi_R) :=$ *unbounded*$(\varphi_1) \cap$ *unbounded*$(\varphi_2)$.
- If $\varphi_R := \varphi_1 \vee \varphi_2$ then *unbounded*$(\varphi_R) :=$ *unbounded*$(\varphi_1) \cup$ *unbounded*$(\varphi_2)$.

By induction it can be shown that $R$ is finite if and only if *unbounded*$(\varphi_R)$ is empty. Although a LOGSPACE Turing machine cannot explicitly generate the set *unbounded*$(\varphi_R)$, it can test for every negated atomic formula $\varphi$ occurring in $\varphi_R$ if $\varphi \in$ *unbounded*$(\varphi_R)$. Thus finiteness of $R$ can be decided in LOGSPACE. Now we show that for every finite $R$ it can be checked in LOGSPACE whether $R$ is of even cardinality. Clearly, if $R$ is finite, then all elements satisfying $\varphi_R$ must occur as parameter in it. Thus the Turing machine can test for every parameter if it satisfies $\varphi_R$ and decide whether the number of parameters for which the test succeeded is even. Thus for every finite relation $R$, the test whether $R$ is of even cardinality can be done in LOGSPACE. This proves the lemma.

□

In this chapter we considered proof methods which are independent of a particular context structure. In contrast we consider in the following chapters special context structures and the data complexity of query languages defined for them.

# Chapter 4

# An overview of data complexity bounds

In this chapter we give an overview of results about the complexity of query evaluation. The chapter is organized as follows. In each of the following sections we cover the results found for a certain context structure. We begin with the simplest case, the extension of finite databases to databases with an infinite domain. In the sections thereafter we add successively more arithmetical functions to the context structure. As we will see, there is a significant difference between discrete and dense structures.

## 4.1 Inequality constraints over an infinite domain

One important restriction of finite databases and their query languages is the restriction to safe queries. Safe queries are those which are guaranteed to produce a finite output on a finite database. To guarantee the safety of a query language for finite databases, its syntax is restricted so that all queries which can be formulated in the restricted language are safe. For example predicate calculus is restricted to the safe calculus by prohibiting the use of universal quantifiers and allow negation only for certain "guarded" expressions. In the context of constraint databases, all these restrictions are unnecessary. The obvious extension of finite databases to constraint databases is to allow an infinite domain and consider all first-order queries on the database, that is to allow negation and universal quantification. Kanellakis, Kuper and Revesz proved in [KKR90] that the data complexity of first-order queries is still LOGSPACE.

**Theorem 4.1.** *First-order logic with equality constraints over an infinite domain has* LOGSPACE *data complexity.*

We give the proof explicitly because it demonstrates a common proof technique for constraint databases. Consider a query with $k$ free variables. The idea is to partition the $k$-dimensional space into a finite number of sets, such that the points in one set cannot be distinguished by first-order formulae and each set can be defined by a quantifier-free formula. A query can be evaluated by testing for each partition if one representative point in it satisfies the query. The answer of the query consists of the union of all partitions for which the test succeeded. This answer can be represented by the disjunction of the formulae defining the partitions in the union.

In the rest of this section we assume a database $\mathfrak{B} = (\mathbb{N}, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$, where $\mathfrak{A} := \mathbb{N}$ is the context structure over the empty signature whose universe is the set of integers. Further, we assume a query $\psi$ where all occurrences of database relation symbols $R$ have been replaced by their representations $\varphi_R^{\mathfrak{B}}$ in $\mathfrak{B}$. Let $P$ be the set of parameters used in $\psi$ or in the representation of $\mathfrak{B}$. Now we use the method outlined above to prove Theorem 4.1. We define a so called *e-configuration* to represent a partition.

**Definition 4.2.** A *$k$-dimensional e-configuration* $\xi = (\sim, \overline{v})$ consists of an equivalence relation $\sim$ on $\{1, \ldots, k\}$ and a sequence $\overline{v} = (v_1, \ldots, v_k)$, where each $v_i$ is in $P \cup \{\cdot\}$, such that for all $1 \leq i, j \leq k$,

*(i)* if $i \sim j$, then $v_i = v_j$, and

*(ii)* if $v_i = v_j$ and $v_i, v_j \neq \cdot$, then $i \sim j$.

The idea behind the e-configurations is as follows. Consider two points $\overline{a}$ and $\overline{b}$ in $A^k$. The two points are distinguishable using inequality constraints and parameters from $P$ if and only if two elements of one tuple are equal and the corresponding elements of the other tuple are not or one element of a tuple equals a constant in $P$ and the corresponding element in the other tuple does not. We prove in Lemma 4.9 below that an e-configuration defines a set of indistinguishable points.

The following two definitions relate points to e-configurations, the first defining the set of points contained in an e-configuration and the second defining for a given point $p$ an e-configuration $\xi$ containing it.

**Definition 4.3.** Suppose $\xi = (\sim, \overline{v})$ is a $k$-dimensional e-configuration. The formula $F(\xi)$ corresponding to $\xi$ is defined as the conjunction of

1. $x_i = x_j$, if $i \sim j$,
2. $x_i \neq x_j$, if $i \not\sim j$,
3. $x_i = v_i$, if $v_i \neq \cdot$, and
4. $x_i \neq p$ for all $p \in P$, if $v_i = \cdot$.

The set of points *contained in* $\xi$ is defined as the set of points satisfying $F(\xi)$.

The next definition explains how an e-configuration $\xi$ can be found for a given point $p$ such that $p$ is contained in $\xi$.

**Definition 4.4.** Let $\overline{a} \in A^k$ be a $k$-tuple. The e-configuration *e-conf*$(\overline{a}) := (\sim, \overline{v})$ is defined as follows.

- $i \sim j$ if and only if $a_i = a_j$.
- If $a_i \in P$ then $v_i = a_i$. Otherwise $v_i = \cdot$.

We illustrate the previous definition by an example. Let $P$ be the set $\{1, 2\}$ and $\overline{a} = (1, 1, 2, 4, 2, 4, 3)$. The e-configuration *e-conf*$(\overline{a})$ consists of the equivalence relation $\overline{e} = \{\{1, 2\}, \{3, 5\}, \{4, 6\}, \{7\}\}$ and the sequence $\overline{v} = (1, 1, 2, \cdot, 2, \cdot, \cdot)$.
The next lemma proves that for a given point $\overline{a} \in A^k$ the e-configuration *e-conf*$(\overline{a})$ contains $\overline{a}$. Moreover, it is the unique e-configuration containing $\overline{a}$.

**Lemma 4.5.** *Let $(a_1, \dots, a_k)$ be a point in $A^k$. There exists a unique e-configuration $\xi$ such that $\mathfrak{A} \models F(\xi)(a_1, \dots, a_k)$.*

*Proof.* Let $\xi :=$ *e-conf*$(\overline{a})$ be the e-configuration according to Definition 4.4. Clearly, $F(\xi)$ is satisfied by $\overline{a}$. Now suppose that $\xi'$ is an e-configuration such that $F(\xi')$ is satisfied by $\overline{a}$ as well. We show that $\xi$ and $\xi'$ must be equal by proving that if $\xi \neq \xi'$ then $F(\xi) \wedge F(\xi')$ is not satisfiable. For the sake of contradiction suppose that $\xi \neq \xi'$ but $(a_1, \dots, a_k)$ satisfies $F(\xi) \wedge F(\xi')$. If $\xi \neq \xi'$, then $\sim \neq \sim'$ or $\overline{v} \neq \overline{v}'$.

If $\sim \neq \sim'$ then there are $i, j$ with $i \sim j$ and $i \not\sim' j$ or $i \not\sim j$ and $i \sim' j$. Suppose w.l.o.g. that $i \sim j$ and $i \not\sim' j$. Thus $F(\xi)$ contains $x_i = x_j$ whereas $F(\xi')$ contains $x_i \neq x_j$ and therefore $F(\xi) \wedge F(\xi')$ is not satisfiable.

On the other hand, if $\sim = \sim'$ but $\overline{v} \neq \overline{v}'$, then there is an $1 \leq i \leq k$ such that $v_i \neq v'_i$. If neither $v_i$ nor $v'_i$ equals $\cdot$ then $F(\xi)$ contains $x_i = v_i$ whereas $F(\xi')$ contains $x_i = v'_i$. As $v_i \neq v'_i$, $F(\xi) \wedge F(\xi')$ is not satisfiable. Now suppose w.l.o.g. that $v_i = p \in P$ and $v'_i = \cdot$. Then $F(\xi)$ contains $x_i = p$ whereas $F(\xi')$ contains $x_i \neq p$ and again $F(\xi) \wedge F(\xi')$ is not satisfiable. □

Before we can define the evaluation algorithm for first-order queries on inequality constraint databases, we need some more technical definitions and lemmas. The first definition and the following lemma explain how a $k$-dimensional e-configuration can be extended to a $(k+1)$-dimensional one.

**Definition 4.6.** Suppose $\xi = (\sim, v_1, \dots, v_n)$ is a $k$-dimensional e-configuration. A $(k+1)$-dimensional e-configuration $\xi' = (\sim', \overline{v}')$ is an *extension* of $\xi$ if and only if for all $1 \leq i, j \leq k$ $i \sim j$ iff $i \sim' j$ and $\overline{v}' = (v_1, \dots, v_k, v'_{k+1})$ for some $v'_{k+1} \in A$.

**Lemma 4.7.** *Let $\xi$ be a $k$-dimensional e-configuration and $\xi'$ be a $(k+1)$-dimensional extension of $\xi$. Then for all $a_1, \dots, a_k \in A$,*

$$\mathfrak{A} \models F(\xi)(\overline{a}) \textit{ iff there is an } a \in A \textit{ such that } \mathfrak{A} \models F(\xi')(\overline{a}, a).$$

*Proof.* For the backward direction note that $F(\xi)$ is a conjunction of formulae occurring also in $F(\xi')$. Thus $\mathfrak{A} \models F(\xi')(\overline{a}, a)$ implies $\mathfrak{A} \models F(\xi)(\overline{a})$.
To prove the forth direction we show that there is an $a \in A$ such that $(a_1, \dots, a_k, a)$ satisfies $F(\xi')$. If $(k+1) \sim i$ for some $1 \leq i \leq k$ then set $a = a_i$. Otherwise if $v_{k+1} = p \in P$ set $a = p$. If none of both is the case set $a = c$ for some $c \in A \backslash P$. This is possible as $A$ is infinite whereas $P$ is always finite. In all cases $(a_1, \dots, a_k, a)$ satisfies $F(\xi')$. □

The next lemma states that every e-configuration defines a non-empty set of elements.

**Lemma 4.8.** *For every $k$-dimensional e-configuration $\xi$ there are elements $a_1, \dots, a_k \in A$ such that $\mathfrak{A} \models F(\xi)(\overline{a})$.*

*Proof.* The proof follows easily by induction on $k$ using Lemma 4.7 and the fact that if $\xi$ is 0-dimensional, then $F(\xi) = true$. □

The last lemma needed to define the evaluation algorithm shows that to decide whether the points contained in an e-configuration satisfy a formula $\psi$ it suffices to test whether one of them satisfies $\psi$.

**Lemma 4.9.** *Suppose $\xi = (\sim, \overline{v})$ is a $k$-dimensional e-configuration and $\psi$ is a formula with at most $k$ free variables $x_1, \dots, x_k$ using only parameters from $P$.*

*(i) If $\mathfrak{A} \models F(\xi)(\overline{a})$ and $\mathfrak{A} \models F(\xi)(\overline{a}')$ then $\mathfrak{A} \models \psi(\overline{a})$ iff $\mathfrak{A} \models \psi(\overline{a}')$.*

*(ii) $F(\xi) \wedge \psi$ is satisfiable in $\mathfrak{A}$ if and only if $F(\xi) \to \psi$ is valid in $\mathfrak{A}$.*

*Proof.*

*(i)* The first part of the lemma is proved by induction on the structure of $\psi$. Suppose $\mathfrak{A} \models F(\xi)(\overline{a})$ and $\mathfrak{A} \models F(\xi)(\overline{a}')$.

- Suppose $\psi := x_i = c$, where $c \in P$. Obviously $\mathfrak{A} \models \psi(\overline{a})$ if and only if $a_i = c$ and therefore $v_i = c$. As $\mathfrak{A} \models F(\xi)(\overline{a}')$, also $a'_i = c$ and thus $\mathfrak{A} \models \psi(\overline{a}')$.
- The proof of the boolean cases is straightforward.
- Now suppose $\psi := \exists x \psi'$. Then $(a_1, \dots, a_k)$ satisfies $\psi$ if and only if for some $a \in A$ $(a_1, \dots, a_k, a)$ satisfies $\psi'$. By Lemma 4.5 there is a $(k+1)$-dimensional e-configuration $\xi'$ such that $\mathfrak{A} \models F(\xi')(\overline{a}, a)$. Obviously $\xi'$ extends $\xi$ and because $(a'_1, \dots, a'_k)$ satisfies $F(\xi)$ there is by Lemma 4.7 an element $a' \in A$ such that $\mathfrak{A} \models F(\xi')(\overline{a}', a')$. As $\psi'$ is a sub-formula of $\psi$ we get by induction that $\psi'$ is satisfied by $(a'_1, \dots, a'_k, a')$ and thus $\psi$ is satisfied by $(a'_1, \dots, a'_k)$.

*(ii)* If $F(\xi) \to \psi$ is valid then, by Lemma 4.8, $F(\xi) \wedge \psi$ is satisfiable. For the backward direction suppose that $\mathfrak{A} \models F(\xi) \wedge \psi(\overline{a})$ for some $\overline{a} \in A^k$. By the first part of this lemma we get that all tuples $\overline{a}'$ satisfying $F(\xi)$, satisfy $\psi$ as well. Thus $F(\xi) \to \psi$ is valid.

□

Now we are ready to define the evaluation algorithm. The algorithm works as follows. Given a query formula $\psi$ with $k$ free variables as input it tests for every $k$-dimensional e-configuration $\xi$ whether $F(\xi) \to \psi$ is valid. This is done by a sub-algorithm *TEST*. It then outputs the disjunction of the formulae $F(\xi)$ for every $\xi$ such that $F(\xi) \to \psi$ is valid. Recall from the beginning of the section that the database relation symbols occurring in the query have been replaced by the formulae representing the relations in the database.

**Algorithm 4.10.** *TEST*$(\psi, \xi)$

**Input:** *A formula $\psi$ with $k$ free variables and a $k$-dimensional e-configuration $\xi = (\sim, \overline{v})$.*

**Output:** true *if $F(\xi) \rightarrow \psi$ is valid,* false *otherwise.*

*The algorithm is defined inductively on the structure of $\psi$.*

- *Suppose $\psi$ is atomic of the form $x_i = x_j$.*
  *If $i \sim j$ return* true *else return* false.
- *Suppose $\psi$ is atomic of the form $x_i = c$.*
  *If $v_i = c$ return* true *else return* false.
- *Suppose $\psi$ is of the form $\psi_1 \vee \psi_2$.*
  *If* $\mathrm{TEST}(\psi_1, \xi)$ *returns* true *return* true, *else return* $\mathrm{TEST}(\psi_2, \xi)$.
- *Suppose $\psi$ is of the form $\neg\psi'$.*
  *If* $\mathrm{TEST}(\psi', \xi)$ *returns* false *then return* true *else return* false.
- *Suppose $\psi$ is of the form $\exists x \psi'$.*
  *For every extension $\xi'$ of $\xi$ do* $\mathrm{TEST}(\psi, \xi')$. *If one of these returns* true *then return* true *as well. Otherwise return* false.

The algorithm *TEST* is used in the following evaluation algorithm $evaluate_\psi$.

**Algorithm 4.11.** $evaluate_\psi(\mathfrak{B})$

**Input:** *A constraint database $\mathfrak{B}$.*

**Output:** *A finite representation of $\psi^{\mathfrak{B}}$.*

*We assume in the algorithm that the occurrences of the database relation symbols in $\psi$ have been replaced by the formulae representing the relations in $\mathfrak{B}$. Further, let $k$ be the number of free variables in $\psi$ and $P$ be the set of parameters used in $\mathfrak{B}$ and $\psi$.*

**for each** *$k$-dimensional e-configuration $\xi$ using parameters from $P$* **do**
  *execute* $\mathrm{TEST}(\psi, \xi)$
**od**
**output** *The disjunction of all $F(\xi)$ so that* $\mathrm{TEST}(\psi, \xi)$ *returned* true.

Having defined the evaluation algorithm we have to prove its correctness. This is done in the following two lemmas, the first proving the correctness of the sub algorithm *TEST* and the second proving the correctness of *evaluate.*

**Lemma 4.12.** *The algorithm* $\mathrm{TEST}(\psi, \xi)$ *returns* true *if and only if $F(\xi) \rightarrow \psi$ is valid.*

*Proof.* The proof is by induction on the structure of $\psi$.

- The atomic cases are trivial.

- Suppose $\psi$ is of the form $\psi_1 \vee \psi_2$. To prove the forth direction recall that *TEST* returns *true* if at least one of the sub algorithms called for $\psi_1$ and $\psi_2$ returns *true*. Thus, by induction, $F(\xi) \rightarrow \psi_1$ or $F(\xi) \rightarrow \psi_2$ and therefore also $F(\xi) \rightarrow (\psi_1 \vee \psi_2)$ is valid.

  Now suppose $F(\xi) \rightarrow (\psi_1 \vee \psi_2)$ is valid. Then, by Lemma 4.9, $F(\xi) \wedge (\psi_1 \vee \psi_2)$ is satisfiable. Thus $F(\xi) \wedge \psi_1$ or $F(\xi) \wedge \psi_2$ is satisfiable and by Lemma 4.9 $F(\xi) \rightarrow \psi_1$ or $F(\xi) \rightarrow \psi_2$ is valid. By induction we get that at least one of $TEST(\psi_1, \xi)$ or $TEST(\psi_2, \xi)$ and therefore also $TEST(\psi, \xi)$ returns *true*.

- Suppose $\psi$ is of the form $\neg\psi'$. To prove the correctness of this case we show that $F(\xi) \rightarrow \neg\psi'$ is valid if and only if $F(\xi) \rightarrow \psi'$ is not. $F(\xi) \rightarrow \neg\psi'$ is equivalent to $\neg(F(\xi) \wedge \psi')$ and, by Lemma 4.9, to $\neg(F(\xi) \rightarrow \psi')$. Thus $F(\xi) \rightarrow \neg\psi'$ is valid if and only if $F(\xi) \rightarrow \psi'$ is not.

- Suppose $\psi$ is of the form $\exists x\psi'$. To prove this case we show that $F(\xi) \rightarrow \psi$ is valid if and only if $F(\xi') \rightarrow \psi'$ is valid for some extension $\xi'$ of $\xi$. By Lemma 4.9 it suffices to show that $F(\xi) \wedge \psi$ is satisfiable in $\mathfrak{A}$ if and only if $F(\xi') \wedge \psi'$ is satisfiable for some extension $\xi'$ of $\xi$.
  For the forth direction suppose that $\mathfrak{A} \models F(\xi) \wedge \psi[\overline{a}]$. Because $\overline{a}$ satisfies $\psi$ there is an $a \in A$ such that $\mathfrak{A} \models \psi'[\overline{a}, a]$. By Lemma 4.5 there is an e-configuration $\xi'$ containing $(\overline{a}, a)$. Clearly, $\xi'$ is an extension of $\xi$ and therefore $\mathfrak{A} \models F(\xi') \wedge \psi'[\overline{a}, a]$.
  For the backward direction suppose that $\mathfrak{A} \models F(\xi') \wedge \psi'[\overline{a}, a]$ for some extension $\xi'$ of $\xi$ and elements $\overline{a}, a \in A$. As $\xi'$ extends $\xi$, Lemma 4.7 gives $\mathfrak{A} \models F(\xi)[\overline{a}]$ and, because $(\overline{a}, a)$ satisfies $\psi'$, $\mathfrak{A} \models \psi[\overline{a}]$. Thus $\mathfrak{A} \models F(\xi) \wedge \psi[\overline{a}]$.
  This finishes the proof.

□

We use the previous result to show the correctness of the evaluation algorithm.

**Lemma 4.13.** *The result of* evaluate$_\psi(\mathfrak{B})$ *is equivalent to* $\psi^{\mathfrak{B}}$.

*Proof.* Let $C = \{\xi_1, \ldots, \xi_n\}$ be the set of all e-configurations such that $F(\xi_i) \rightarrow \psi$ is valid. Thus the result of *evaluate*$_\psi$ is $\bigvee_{1 \leq i \leq n} F(\xi_i)$. To prove the lemma we show that a tuple $\overline{a}$ satisfies $\psi$ in $\mathfrak{B}$ if and only if it satisfies $\bigvee_{1 \leq i \leq n} F(\xi_i)$.
Clearly, $(\bigvee_{1 \leq i \leq n} F(\xi_i)) \rightarrow \psi$ is valid and therefore every tuple $\overline{a}$ satisfying $(\bigvee_{1 \leq i \leq n} F(\xi_i))$ satisfies $\psi$ as well. For the converse let $\overline{a}$ be a tuple with $\mathfrak{A} \models \psi(\overline{a})$. By Lemma 4.5 there is an e-configuration $\xi$ containing $\overline{a}$. Thus $\overline{a}$ satisfies $F(\xi) \wedge \psi$ and, by Lemma 4.9, $F(\xi) \rightarrow \psi$ is valid. Therefore $\xi \in C$ and $\overline{a}$ satisfies $(\bigvee_{1 \leq i \leq n} F(\xi_i))$. □

Having proven the correctness of the algorithm we still have to show that it runs in logarithmic space.

**Lemma 4.14.** evaluate$_\psi$ *is a* LOGSPACE-*algorithm.*

*Proof.* Let $\psi$ be a first-order query with $k$ free variables and $\mathfrak{B} := (\mathfrak{A}, R_1^{\mathfrak{B}}, \ldots, R_s^{\mathfrak{B}})$ be the input database. To prove the lemma we show that both algorithms, *TEST* and *evaluate*, are LOGSPACE-algorithms. We begin with the sub-algorithm *TEST*. *TEST* is called by *evaluate* with the query $\psi$ and a $k$-dimensional e-configuration $\xi$ as parameter. Recall that we assumed in the definition of the algorithms that the database relation symbols occurring in $\psi$ have been replaced by the formulae defining the relations in $\mathfrak{B}$. Let $\psi' := \psi[R_i/\varphi^{\mathfrak{B}}_{R_i}]$ be the result of the substitution. Clearly, $\psi'$ cannot be stored on the working tape as it consumes more than logarithmic space. Instead of explicitly creating $\psi'$, the algorithm operates on the original query $\psi$ and each time it comes to a database relation symbol $R$ it remembers its current position in the query and continues the evaluation with the formula $\varphi^{\mathfrak{B}}_R$ on the input tape. The Turing machine can remember the current position in the query in its states. Thus no extra space is used for the representation of $\psi'$.

We now show that a $k$-dimensional e-configuration $\xi$ can be stored on the working tape in logarithmic space. As $\psi$ is fixed, the number $k$ of free variables is independent of the input. Recall that an e-configuration essentially consists of a $k$ tuple of coefficients from the query or the database and the equivalence relation $\sim$. Clearly, the size of the equivalence relation depends only on $k$ and is therefore fixed. The coefficients in $\xi$ occurring in the query are independent of the input and therefore fixed in size whereas the coefficients occurring in the database can be stored by pointers to their occurrence on the input tape. Thus the e-configuration $\xi$ can be stored by at most $k^2$ pointers, and with this, in space logarithmic to the size of the input. We now show that *TEST* does not use more than logarithmic space. The algorithm operates by induction on the structure of the query. Whenever it comes to a database relation symbol $R$ it has to test whether $F(\xi) \wedge \varphi^{\mathfrak{B}}_R$ is satisfiable in $\mathfrak{A}$. This can clearly be done in LOGSPACE.

The only other interesting case is existential quantification. Here *TEST* calls itself for every extension $\xi'$ of $\xi$. Clearly there are only finitely many extensions and an extension can also be stored in logarithmic space. As the space occupied by an extension can be reused for the next one, the algorithm has only to store one of the extensions at a time. By induction we get that the recursive calls with one extension can also be done in LOGSPACE. This proves that *TEST* is a LOGSPACE-algorithm.

Now consider the algorithm *evaluate*. All it does is to generate all possible e-configurations of dimension $k$ with parameters form the query or the database and to execute *TEST* for each of it. As the space used for the e-configurations can be reused and *TEST* works in LOGSPACE, *evaluate* is itself a LOGSPACE-algorithm. This finishes the proof. □

The last two lemmas together prove Theorem 4.1 giving the LOGSPACE upper complexity bound for first-order queries.
Using the methods we develop in Chapter 5, complexity bounds for other logics like transitive closure or least fixed-point logic can be proven. We already saw in Section 3.2 that these logics cannot express all LOGSPACE computable queries. Table 4.1 gives an overview of the complexity bounds for databases defined over countably infinite sets.

In the next sections we switch from unordered to ordered structures. As we will see, there is a significant difference between structures with a discrete universe and dense structures. The next section deals with dense structures whereas the discrete structures

| | | |
|---|---|---|
| *FO+DTC* | $\subset$ | LOGSPACE |
| *FO+TC* | $\subset$ | NLOGSPACE |
| *FO+LFP* | $\subset$ | PTIME |
| *FO+PFP* | $\subset$ | PSPACE |

Table 4.1: Logics and complexity classes in the context of inequality constraints.

will be covered in the section thereafter.

## 4.2 Structures with a densely ordered universe

A straightforward extension of the databases considered in the previous section are dense order constraint databases, that is databases over a context structure which is a dense order. We consider such databases in the next section. In the sections thereafter we enrich the context signature first by addition and then by addition and multiplication.

### 4.2.1 Dense order databases

In this section we consider databases over dense orders. Kanellakis, Kuper and Revesz considered dense order databases in [KKR90] and proved that first-order queries can be evaluated in LOGSPACE.

**Theorem 4.15.** *First-order queries over dense order databases have* LOGSPACE *data complexity.*

Their argument is very similar to the one presented above to prove the LOGSPACE upper bound for inequality constraints over an infinite domain. The proof is based on r-configurations instead of the e-configurations used above. Again let $P$ be the set of parameters occurring in the query or the database.

**Definition 4.16.** A *$k$-dimensional r-configuration* $\xi = (\overline{f}, \overline{l}, \overline{u})$ consists of a sequence $\overline{f} = (f_1, \ldots, f_k)$, where $\{f_1, \ldots, f_k\} = \{1, \ldots, j\}$ for some $j \leq k$, and two sequences $\overline{l} = (l_1, \ldots, l_k)$ and $\overline{u} = (u_1, \ldots, u_k)$, where the $l_i$'s are in $P \cup \{-\infty\}$ and the $u_i$'s are in $P \cup \{\infty\}$, such that for all $1 \leq i, j \leq k$:

- $l_i \leq u_i$.
- There is no constant $c \in P$ with $l_i < c < u_i$.
- Whenever $f_i < f_j$, then $l_i < u_j$.
- Whenever $f_i = f_j$, then $l_i = l_j$ and $u_i = u_j$.

The idea is that two points $\overline{x}$ and $\overline{y}$ can be distinguished using order constraints and the available parameters if the relative order of the $x_i$'s and the $y_i$'s differ or if for some $i$, $x_i$ and $y_i$ are in a different relation to some element of $P$. Using r-configurations instead

of e-configurations the proof of Theorem 4.15 is very similar to the proof given in the previous section. We don't go into details here because dense order databases will be considered in detail in Chapter 5, where not only first-order logic but also its various fixed-point extensions will be considered.
In other work, Kanellakis and Goldin [KG94] give an extension of Codd's relational algebra to dense order databases. This constraint algebra can be used to implement evaluation algorithms for first-order queries on dense order databases which are based on circuits. By doing this an $\mathrm{AC}_0$ upper bound can be shown.

### 4.2.2 Linear constraints

In this section we consider linear constraint databases, that is, databases defined over the context structure $(\mathbb{R}, <, +)$. An example for the application of linear constraint databases are databases used by CAD-systems. The data complexity of linear constraint queries has been studied by Grumbach, Su, and Tollu in [GS97b] and [GST95]. In [GS97b] Grumbach and Su claim that "first-order queries on linear constraint databases have a $\mathrm{NC}_1$ data complexity." Unfortunately the proof is given only for the two dimensional case and cannot be extended to higher dimensions.

We briefly investigate the data complexity of more expressive logics than first-order. It turns out that adding a recursion mechanism to first-order logic leads to non-closed or undecidable languages. All fixed-point extensions of first-order logic considered so far are not closed. For example, the following *FO+DTC* formula

$$nat(x) := [DTC_{x,y}(x + 1 = y)](0, x)$$

defines the natural numbers. As the natural numbers are not even definable in the field of reals the result of the query *nat* cannot be represented by a quantifier-free formula over $(\mathbb{R}, <, +)$. The following theorem shows that there is no way to enrich the context structure by functions or relations such that *FO+DTC*-queries would be decidable and closed.

**Theorem 4.17.** *Every query language over the context structure $(\mathbb{R}, <, +)$ which is at least as expressive as existential FO+DTC is undecidable.*

*Proof.* To prove the theorem we reduce the decision problem for the existential theory of arithmetic, which is known to be undecidable (see [EFT94]), to the evaluation problem of boolean existential *FO+DTC*-queries. As we saw above the natural numbers are definable in *FO+DTC*. The graph of the multiplication for natural numbers can be defined by the *FO+DTC*-formula

$$mult(a, b, c) := [DTC_{x,y,x',y'}(y + 1 = y' \wedge x + a = x')](0, 0, c, b).$$

Now we reduce the decision problem for the existential theory of arithmetic to the query evaluation problem for boolean *FO+DTC*-queries. For each sentence $\varphi$ of the existential theory of arithmetic let $\varphi'$ be the result of relativizing each quantifier in $\varphi$ to *nat* and replacing each occurrence of multiplication by the formula *mult*. Note that as

multiplication occurs in $\varphi$ as a function, we have to introduce new existential quantifiers to replace it by the formula *mult*. These new quantifiers are also relativized to *nat*.
The formulae *nat* and *mult* are both existential *FO+DTC*-formulae. Therefore $\varphi'$ is also existential. Clearly, $\varphi$ is true in the existential theory of arithmetic if and only if $\varphi'$ evaluates to true. Thus the undecidability of the evaluation problem follows from the undecidability of the existential theory of arithmetic. □

### 4.2.3 Real closed fields

In this subsection we consider databases defined over the field of reals. Databases like this can be used to store geometrical objects, one of the most promising applications of constraint databases.

In their seminal paper [KKR90], Kanellakis, Kuper and Revesz considered databases over the field of reals and proved an NC upper data complexity bound.

**Theorem 4.18.** *First-order queries in the context of the field of reals have NC data complexity.*

The proof of the theorem is based on the following result by Ben-or, Kozen and Reif [BoKR86].

**Theorem 4.19.** *The theory of real closed fields can be decided in deterministic exponential space or parallel exponential time. In fixed dimension, the theory can be decided in NC.*

The main result of the previous section immediately implies that the various fixed-point extensions of first-order logic are undecidable over the field of reals.

## 4.3 Structures with a discretely ordered universe

In the previous sections we analyzed the complexity of query languages in the context structures with a dense universe. Now we shift our attention to structures with a discrete universe. As we will see, there is a serious gap in complexity between dense and discrete order query languages. We proceed as in the previous sections, beginning with a discrete order as context structure and successively enriching the signature in the succeeding sections by addition and addition and multiplication.

### 4.3.1 Discrete linear orders

In Section 4.2.1 we saw that extending finite databases to dense order databases still yields efficient query languages. As we will see in this section the situation changes drastically if discrete instead of dense orders are considered. We will show that *FO+LFP* is already Turing complete. Throughout this section $(\mathbb{N}, <)$ serves as an example of discrete linear orders.

Grumbach and Su [GS97b] stated that the techniques used by Kanellakis and Goldin in [KG94] to prove an $\mathrm{AC}_0$ bound for dense linear orders could be extended to discrete orders. Unfortunately this is not the case as the theory of discrete linear orders does not admit quantifier elimination. For example the formula $\varphi(x, y) := \exists z\ x < z \wedge z < y$ is not equivalent to any quantifier-free formula. To obtain quantifier elimination for $(\mathbb{N}, <)$, the language has to be enriched by a countable set of relations $\{<_c :\ c \in \mathbb{N} \cup \{-1\}\}$. The intended meaning of $x <_c y$ is $x + c < y$. These relations are called *gap-orders*. Using this signature the formula $\varphi$ above can be translated to an equivalent formula $\psi(x, y) := x <_1 y$ which is quantifier-free. Note that $<_0$ is the usual order and $<_{-1}$ corresponds to $\leq$. In the following we consider the context structure $(\mathbb{N}, (<_c)_{c \in \mathbb{N} \cup \{-1\}})$.

The complexity of first-order queries on gap-order databases will be considered in some detail in Chapter 6. Here we consider the data complexity of more expressive logics. We will see in Chapter 5 that, because it captures PTIME, *FO+LFP* is an important logic for the class of dense order databases. The next theorem by Revesz [Rev93] shows that it is already too powerful over the class of discrete orders because it can express any Turing computable function.

**Theorem 4.20.** *Any Turing computable function is expressible by an FO+LFP formula on discrete gap-order databases.*

*Proof.* The class of Turing computable functions coincides with the class of $\mu$-recursive functions. Thus is suffices to show that every $\mu$-recursive function is expressible by an *FO+LFP* formula. These functions are built up from the initial functions *zero*, *proj*, and *succ* by the operations of composition, primitive recursion, and the $\mu$-operator. In the following we show that the graph of these (partial) functions is definable by *FO+LFP*-formulae.
The functions $zero(x)$, $proj_i(x_1, \ldots, x_n, y)$, and $succ(x, y)$ can be defined by

$$\begin{aligned}
&\varphi_{zero}(x) := \neg\exists y\ y < x,\\
&\varphi_{proj_i}(x_1, \ldots, x_n, y) := x_i = y, \text{ and}\\
&\varphi_{succ}(x, y) := x < y \wedge \neg\exists z\ x < z \wedge z < y.
\end{aligned}$$

Given formulae $\varphi_g, \varphi_{h_1}, \ldots, \varphi_{h_k}$ corresponding to $\mu$-recursive functions $g$ and $h_i$, the function $f = g(h_1, \ldots, h_k)$ derived by composing $g$ and $h_1, \ldots, h_k$ can be expressed by

$$\varphi_f(\overline{x}, y) := \exists \overline{u}\ \varphi_g(\overline{u}, y) \wedge \bigwedge_{1 \leq i \leq k} \varphi_{h_i}(\overline{x}, u_i).$$

Recall that a function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is obtained by primitive recursion on $g : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ if

$$\begin{aligned}
f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n)\\
f(x_1, \ldots, x_n, i+1) &= h(x_1, \ldots, x_n, f(x_1, \ldots, x_n, i), i).
\end{aligned}$$

Let $\varphi_g$ and $\varphi_h$ be formulae corresponding to $g$ and $h$. We define an $(n+2)$-ary relation $R$ such that $R\overline{a}, c, b$ holds if $f(\overline{a}, c) = b$. The relation is used in the fixed-point formula

$$\begin{aligned}
\varphi_f(\overline{x}, i, y) := [LFP_{R,\overline{x},i,y}((i = 0 \wedge \varphi_g(\overline{x}, y)) \vee\\
(i \neq 0 \wedge \exists y' R\overline{x}, i-1, y' \wedge \varphi_h(\overline{x}, y', i-1, y)))](\overline{x}, i, y)
\end{aligned}$$

defining the graph of the function $f$. It states that "$i = 0$ and $g(\overline{x}) = y$ or $i \neq 0$ and there is $y'$ such that $f(\overline{x}, i-1) = y'$ and $h(\overline{x}, y', i) = y$". The abbreviation $i-1$ used in the definition of $\varphi_f$ can be defined using the successor function.
The last operation we have to consider is the $\mu$-operator. Recall that the $\mu$-operator is defined as follows. Let $g : \mathbb{N}^{n+1} \to \mathbb{N}$ be a $\mu$-recursive function. The function $f : \mathbb{N}^n \to \mathbb{N}, f := \mu y g$ is derived from $g$ by an application of the $\mu$-operator, if

$$f(\overline{x}) := \begin{cases} \text{the smallest } y \text{ such that} & g(\overline{x}, y) = 0 \text{ and for all } z < y \\ & g(\overline{x}, z) \text{ is defined and not } 0 \\ \text{undefined,} & \text{if no such } y \text{ exists.} \end{cases}$$

Let $\varphi_g(x_1, \dots, x_n, x_{n+1}, y)$ be a formula corresponding to the function $g$. The formula $\varphi_f(x_1, \dots, x_n, y)$ can be defined as

$$\varphi_f(x_1, \dots, x_n, y) := g(\overline{x}, y, 0) \wedge \forall z < y \exists y' > 0 \ (\varphi_g(\overline{x}, z, y')).$$

Thus all $\mu$-recursive functions can be expressed by *FO+LFP*-formulae. □

We get from the theorem that *FO+LFP* is too expressive in the context of discrete orders. The reason for this is the combination of recursion and the definability of the successor function. Leaving out recursion leads to first-order queries which are considered in detail in Chapter 6, where it is shown that their data complexity is LOGSPACE.

If the formulae are restricted to positive ones, the successor function becomes undefinable. Revesz considered in [Rev90] and [Rev93] positive DATALOG queries on gap-order databases. He used a representation of constraints by graphs, so called *gap-graphs*, to set up an evaluation algorithm and proved the algorithm to terminate on every query. But nothing more is known of its complexity. Since positive DATALOG is equivalent to existential positive fixed-point logic, we get the following theorem.

**Theorem 4.21 ([Rev93]).** *Existential positive FO+LFP queries can be evaluated bottom up in closed form.*

In this section we saw that the combination of gap-orders as context structure and recursive query languages leads to high data complexity. In the next section we add addition to the natural numbers and see that even boolean first-order queries have a very high data complexity.

### 4.3.2 Presburger arithmetic

In the last section we saw that the structure $(\mathbb{N}, <)$ does not admit quantifier elimination but can be enriched by certain relations so that the expanded structure admits quantifier elimination. The same happens if we consider the structure $(\mathbb{N}, <, +)$, also known as Presburger arithmetic. An example that shows that the Presburger arithmetic does not admit quantifier elimination is the set of even numbers which can be defined by $\varphi(x) := \exists z \ z + z = x$ but which cannot be defined without quantifiers.

Grumbach and Su claimed in [GS97b] that first-order queries on $(\mathbb{N}, <, +)$ as context structure could be evaluated in $NC_1$ data complexity. We will prove this to be false using the methods of Section 3.1.

The proof is based on the following theorem by Grädel [Grä88] and Schöning [Sch97].

**Theorem 4.22.** *(Grädel [Grä88], Schöning [Sch97])*
*If $m$ is odd, then*

*(a)* $[\exists_1\forall_2\ldots\exists_m\forall^3]$ *is* $\Sigma^p_m$*-complete.*

*(b)* $[\forall_1\exists_2\ldots\forall_m\exists^3]$ *is* $\Pi^p_m$*-complete.*

*If $m$ is even, then*

*(a)* $[\exists_1\forall_2\ldots\forall_m\exists^3]$ *is* $\Sigma^p_m$*-complete.*

*(b)* $[\forall_1\exists_2\ldots\exists_m\forall^3]$ *is* $\Pi^p_m$*-complete.*

We use this to prove the following theorem.

**Theorem 4.23.** *For each level $\Sigma^p_k$, resp. $\Pi^p_k$, of the polynomial time hierarchy there is a fixed query such that the data complexity of the query is $\Sigma^p_k$-complete, resp. $\Pi^p_k$-complete.*

*Proof.* A direct implication of Theorem 4.22 and Theorem 3.3 is that for each level $\Sigma^p_k$, or $\Pi^p_k$ resp., there are queries such that $\Sigma^p_k$, or $\Pi^p_k$ resp., is a lower bound for the data complexity of the queries. In the proof of Theorem 3.3 we reduced the decision problem for a $k$-dimensional theory to the evaluation problem for boolean queries. We can therefore reduce the decision problem for the prefix classes mentioned above to the evaluation problem for boolean queries. This proves that for each level of the polynomial time hierarchy there are queries whose data complexity is complete for that level. □

As a corollary of the theorem we get that FO captures PH on the class of constraint databases defined over the Presburger arithmetic.

**Corollary 4.24.** *First-order logic captures the polynomial time hierarchy on the class of constraint databases defined over the Presburger arithmetic.*

*Proof.* Clearly, FO $\subseteq$ PH because every first-order query either already has a prefix of the form as in Theorem 4.22 or it can be converted to such a form by adding quantifiers. The corollary now follows immediately from Theorem 4.23. □

As we can see, first-order logic can express quite complex queries. A natural question is to ask for sub-classes of first-order logic which can still be evaluated in polynomial time. We will see that existential and universal boolean queries are such classes. The following lemma is due to Lenstra [Len83] and Scarpellini [Sca84].

**Lemma 4.25.** *For all fixed dimensions $t \in \mathbb{N}$, $[\exists^t]$ and $[\forall^t]$ are in* PTIME.

As an implication of this lemma and Theorem 3.4 we get the following theorem.

**Theorem 4.26.** *Existential and universal boolean queries have* PTIME *data complexity.*

Although first-order queries have a very high data complexity, boolean existential or universal queries are still tractable. Schöning [Sch97] proved that the prefix class $[\exists\forall] \cap PA$, that is the set of all sentences with quantifier prefix $\exists\forall$ true in the Presburger arithmetic, is NP-complete. Thus, unless PTIME = NP, the above two classes are the only sub-classes which can be evaluated in PTIME.

In Section 4.2.2 we saw that the various fixed-point extensions of first-order logic lead to undecidable query languages over the context structure $(\mathbb{R}, <, +)$, because the natural numbers with addition and multiplication become definable. The same happens in the context of the Presburger arithmetic because the formula *mult* given in Section 4.2.2 can be used to define multiplication in this context as well. Thus also with the Presburger arithmetic as context structure, adding fixed-points to first-order logic leads to undecidable languages.

### 4.3.3 The theory of arithmetic

We proved in the last section that using Presburger arithmetic as context structure results in query languages with very high data complexity. Adding multiplication to the Presburger arithmetic leads to undecidable query languages as the first-order theory of $(\mathbb{N}, <, +, *)$ is undecidable. In contrast, when the universe of the context structure is densely ordered, we have been able to add multiplication to the signature and still end up in tractable query languages.

## 4.4 Summary

The results presented in this chapter are summarized in Table 4.2 and 4.3.

| | inequality | $(\mathbb{R}, <)$ | $(\mathbb{R}, <, +)$ | $(\mathbb{R}, <, +, *)$ |
|---|---|---|---|---|
| FO | LOGSPACE | LOGSPACE | ? | NC |
| ex. *FO+DTC* | - | - | undecidable | undecidable |

Table 4.2: Overview, Part *I*, of the results presented in Chapter 4.

| | $(\mathbb{N}, <_c)$ | $(\mathbb{N}, <, +)$ | $(\mathbb{N}, <, +, \cdot)$ |
|---|---|---|---|
| bool. univ. or ex. FO | - | PTIME | - |
| boolean FO | - | = PH | undecidable |
| ex. pos. *FO+LFP* | closed | - | - |
| *FO+LFP* | Turing complete | undecidable | undecidable |

Table 4.3: Overview, Part *II*, of the results presented in Chapter 4.

# Chapter 5

# Dense linear orders

The focus of this chapter is the complexity of query evaluation in the context of dense linear orders. We prove a general result which allows us to give precise complexity bounds for the data complexity of various logics such as transitive closure or fixed-point logic and to extend results on logics capturing complexity classes from the realm of finite ordered structures to constraint databases over dense linear orders. Given a fixed query, its evaluation in a database can be done by transforming the database into a finite structure, called its invariant, evaluating a slightly modified version of the query in it, and transforming the result of the evaluation to an answer of the original query. Although this evaluation method may seem to be a long way round, it will actually prove to be a shortcut.

## 5.1 Evaluating queries

In this chapter the context structure $\mathfrak{A} := (\mathbb{R}, <)$ serves as an example of a dense linear order without endpoints, but the results hold for arbitrary dense linear orders, because no special features of the reals will be used. Throughout the chapter we consider a fixed query $\psi$ with a set $P^\psi$ of parameters. The query has to be transformed so that it can be evaluated in the invariant. This transformation is independent of a particular database and can be seen as a compilation or preprocessing step. To set up the evaluation method outlined above, we define two mappings. The first, *inv*, maps databases to their corresponding invariants; the second, $\pi$, maps the answer of the query on the invariant to the answer of the original query. Before the mappings are defined we fix some notation and prove a few facts about dense linear order databases.

**Definition 5.1.** Let $\sigma := \{R_1, \dots, R_k\}$ be a signature, $\mathfrak{B}$ be a $\sigma$-database over $(\mathbb{R}, <)$, $P \subset \mathbb{R}$ a set of elements, and $\bar{b}$ a tuple of real numbers.

- The *complete atomic type of $\bar{b}$ with respect to $\mathfrak{B}$*, written as $atp^{\mathfrak{B}}(\bar{b})$, is the set of all atomic and negated atomic formulae $\varphi(\bar{x})$ over the signature $\{<, R_1, \dots, R_k\}$ such that $\mathfrak{B} \models \varphi[\bar{b}]$.
- The *complete atomic type of $\bar{b}$ over $P$ with respect to $\mathfrak{B}$*, $atp_P^{\mathfrak{B}}(\bar{b})$, is defined in the same way as $atp^{\mathfrak{B}}(\bar{b})$, but with formulae that may also use the parameters from $P$.

- The *complete order type of* $\overline{b}$ *with respect to* $\mathfrak{B}$, $otp^{\mathfrak{B}}(\overline{b})$, is defined as the complete atomic type of $\overline{b}$ over the signature $\{<\}$. The definition of a *complete order type of* $\overline{b}$ *over* $P$ is analogous.

- A maximally consistent set of atomic and negated atomic $\sigma \cup \{<\}$-formulae $\varphi(\overline{x})$ is a *complete atomic type (over* $P$*) in the variables* $\overline{x}$, if it is a complete atomic type (over $P$) of a tuple $\overline{b}$ with respect to a $\sigma$-expansion of $\mathfrak{A}$. We write $atp^{\mathfrak{B}}(\overline{x})$, resp. $atp_P^{\mathfrak{B}}(\overline{x})$, for a complete atomic type (over $P$) in the variables $\overline{x}$ over the database signature $\sigma$ of $\mathfrak{B}$.

A type is an $n$-type if it has $n$ free variables. We omit $\mathfrak{B}$ if it is clear from the context. When speaking about types we always mean complete atomic types throughout this chapter.

We call complete atomic types over $\sigma \cup \{<\}$ also *complete database types*. Database types are of special interest here because the database type of a tuple $\overline{b}$ determines everything we can say about $\overline{b}$ in terms of the database, especially in which database relations $\overline{b}$ stands.

Suppose $\mathfrak{B}$ is a database and $P$ the set of parameters used in its definition. Recall from the introduction that there are different ways to represent the database $\mathfrak{B}$. The set of parameters used in these representations will generally differ from $P$. We define a set of parameters, called the canonical parameters, which can be extracted from $\mathfrak{B}$ independent of its representation.

**Definition 5.2.** Suppose $\mathfrak{B} = (\mathbb{R}, <, R_1^{\mathfrak{B}}, \dots, R_k^{\mathfrak{B}})$ is a database. The set $cp(\mathfrak{B}) \subset \mathbb{R}$ of *canonical parameters of* $\mathfrak{B}$ is the set of elements $p$ satisfying the following condition.
For at least one $n$-ary relation $R \in \{R_1^{\mathfrak{B}}, \dots, R_k^{\mathfrak{B}}\}$ there are $a_1, \dots, a_n \in \mathbb{R}$, an $\varepsilon \in \mathbb{R}, \varepsilon > 0$, and an $\varepsilon$-neighbourhood $\delta = (p - \varepsilon, p + \varepsilon)$ of $p$ such that one of the following holds.

- For all $q \in \delta, q < p$ and for no $q \in \delta, q > p$ we have $R\overline{a}[p/q]$.

- For all $q \in \delta, q > p$ and for no $q \in \delta, q < p$ we have $R\overline{a}[p/q]$.

- $R\overline{a}[p/q]$ holds for all $q \in \delta \backslash \{p\}$ but not for $q = p$.

- $R\overline{a}[p/q]$ holds for $q = p$ but not for any $q \in \delta \backslash \{p\}$.

$R\overline{a}[p/q]$ means that all components $a_i = p$ are replaced by $q$.

One important property of $cp(\mathfrak{B})$ we need in the sequel is that $cp(\mathfrak{B})$ is finite for all dense order constraint databases. This is proved in the following lemma.

**Lemma 5.3.** *The set* $cp(\mathfrak{B})$ *of canonical parameters is finite for all databases* $\mathfrak{B}$.

*Proof.* We claim that all canonical parameters occur explicitly as constants in every representation of the database. The proof of the lemma then follows easily because as all representations are finite, only finitely many parameters can occur in them and thus the set of canonical parameters must also be finite.

To prove the claim let $p$ be a canonical parameter. There exists a relation $R$, an $n$-tuple $\overline{a}$, and an $\varepsilon$-neighbourhood $\delta$ of $p$ such that at least one of the four conditions of Definition 5.2 is met. We give the proof explicitly for the first condition. The other cases can be treated analogously.
As the first condition is met, for all $q \in \delta, q < p$ and for no $q \in \delta, q > p$, we have $R\overline{a}[p/q]$. Thus there is a boolean combination of atomic and negated atomic formulae in the representation of $R$ distinguishing between the two sets $\{\overline{a}[p/q] \ : \ q \in \delta, q < p\}$ and $\{\overline{a}[p/q] \ : \ q \in \delta, q > p\}$. As the universe $\mathbb{R}$ is dense, we can always find an $\varepsilon' > 0$ such that no $a_i \neq p$ is an element of the $\varepsilon'$-neighbourhood $\delta' := (p - \varepsilon', p + \varepsilon')$ of $p$. Clearly, no formula of the form $x_i < x_j$ or $x_i = x_j$ can distinguish between any two points $\overline{b}, \overline{b}'$, such that $\overline{b} \in A^- := \{\overline{a}[p/q] \ : \ q \in \delta', q < p\}$ and $\overline{b}' \in A^+ := \{\overline{a}[p/q] \ : \ q \in \delta', q > p\}$. As for all $\overline{b} \in A^-$ and for no $\overline{b} \in A^+$ we have $R\overline{b}$, there must be a boolean combination of formulae of the form $x_i < c$ or $x_i = c$, where $c$ is a parameter, in the representation of $R$, distinguishing between $A^-$ and $A^+$. Obviously, the parameter $p$ must occur in these formulae. This proves the claim. □

The parameters in the previous definition have been called canonical, because they can be defined in the database independent of a particular representation. We show in the next lemma that an atomic order type over $cp(\mathfrak{B})$ uniquely determines a complete database type. It follows that every two tuples realizing the same atomic order type over $cp(\mathfrak{B})$ occur in the same database relations and thus the set $cp(\mathfrak{B})$ is sufficient to define a representation of $\mathfrak{B}$.

**Lemma 5.4.** *Suppose $\mathfrak{B}$ is a database and $\overline{a}, \overline{b} \in \mathbb{R}^k$ are two $k$-tuples.*

*(i) If $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{b})$, then $atp^{\mathfrak{B}}(\overline{a}) = atp^{\mathfrak{B}}(\overline{b})$.*

*(ii) If $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(a_i) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(b_i)$ for all $1 \leq i \leq k$ and $otp^{\mathfrak{B}}(\overline{a}) = otp^{\mathfrak{B}}(\overline{b})$, then $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{b}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a})$.*

*Proof.*

*(i)* For the sake of contradiction suppose that $atp^{\mathfrak{B}}(\overline{b})$ and $atp^{\mathfrak{B}}(\overline{a})$ differ. Then there is an atomic or negated atomic formula $\varphi$ such that $\mathfrak{B} \models \varphi[\overline{a}]$ but $\mathfrak{B} \not\models \varphi[\overline{b}]$. If $\varphi$ is of the form $x_i < x_j$, then $a_i < a_j$ but not $b_i < b_j$, which contradicts the assumption that $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{b}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a})$.
Now suppose $\varphi$ is of the form $Rx_1, \dots, x_r$, where $r := ar(R)$. Let $\mathcal{C} := (\overline{c}_0, \overline{c}_1, \dots, \overline{c}_k)$ be a sequence of points in $\mathbb{R}^k$, such that for $0 \leq i \leq k$, $c_{ij} := b_j$ for all $1 \leq j \leq i$ and $c_{ij} := a_j$ for all $i < j \leq k$. Thus $\overline{c}_0 = \overline{a}$, $\overline{c}_k = \overline{b}$, $\overline{c}_1 = (b_1, a_2, \dots, a_k)$, $\overline{c}_2 = (b_1, b_2, a_3, \dots, a_k)$, and so on. Further, let $L := (l_1, \dots, l_k)$ be a sequence of lines such that for all $1 \leq i \leq k$ the endpoints of $l_i$ are $c_{i-1}$ and $c_i$. As $\mathfrak{B} \models \varphi[\overline{a}]$ but $\mathfrak{B} \not\models \varphi[\overline{b}]$, there is $1 \leq j \leq k$ such that $l_j$ intersects both $R^{\mathfrak{B}}$ and $\mathbb{R}^k \backslash R^{\mathfrak{B}}$. Assume w.l.o.g. that $a_j < b_j$. Let $\overline{q} := \overline{c}_{j-1}$. Then there is $p \in \mathbb{R}$ with $a_j < p \leq b_j$ such that $R^{\mathfrak{B}}\overline{q}$ but not $R^{\mathfrak{B}}q_1, \dots, q_{j-1}, p, q_{j+1}, \dots, q_k$. We claim that there is at least one canonical parameter $d$ with $a_j \leq d \leq p$. Thus $\overline{a}$ and $\overline{b}$ do not satisfy the same

complete order type over $cp(\mathfrak{B})$ which contradicts the assumption.
To prove the claim, let $A$ be a set of real numbers defined as

$$A := \{a \in \mathbb{R} \ : \ a_j \leq a \text{ and } R^{\mathfrak{B}} q_1, \ldots, q_{j-1} a' q_{j+1}, \ldots, q_k \text{ for all } a_j \leq a' \leq a\}.$$

If $A$ is closed let $d$ be the biggest element in $A$, otherwise let $d$ be the upper boundary of $A$. Then, by Definition 5.2, $c$ is a canonical parameter and $a_j \leq d \leq p$. This proves the claim.

The case where $\varphi$ is negated atomic can be proven analogously.

*(ii)* The proof follows from the fact that all formulae $\varphi$ occurring in $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{x})$ for variables $x_1, \ldots, x_k$ either occur in $otp^{\mathfrak{B}}(\overline{x})$, if they are of the form $x_i < x_j$ for some $1 \leq i, j \leq k$, or occur in $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(x_i)$, if they are of the form $x_i = p$ for some $p \in cp(\mathfrak{B})$. Thus, because $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(a_i) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(b_i)$ for all $1 \leq i \leq k$ and $otp^{\mathfrak{B}}(\overline{a}) = otp^{\mathfrak{B}}(\overline{b})$, also $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a})$.

□

One implication of the lemma is the following. Suppose we want to decide if $R\overline{a}$ holds for a tuple $\overline{a} := a_1, \ldots, a_k$ and a $k$-ary database relation $R$. The question can be answered if we know whether $R\overline{b}$ holds for a tuple $\overline{b} := b_1, \ldots, b_k$ such that $\overline{a}$ and $\overline{b}$ realize the same order type and each $b_i$ realizes the same 1-order type over $cp(\mathfrak{B})$ as $a_i$. This will be the central idea in the definition of the invariant.

The lemma also has some other useful corollaries.

**Corollary 5.5.** *Suppose $\mathfrak{B}$ is a $\sigma$-database and $\overline{a}, \overline{b} \in \mathbb{R}$ are $k$-tuples. If $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{b})$ then $\overline{a}$ and $\overline{b}$ cannot be distinguished by a first-order formula using only parameters from $cp(\mathfrak{B})$.*

*Proof.* Let $\varphi \in FO[\sigma \cup \{<\}]$ be a first-order formula and $\varphi' := unfold(\varphi, \mathfrak{B})$ be the unfolded query according to Definition 2.9. As $\varphi'$ is a first-order formula over the signature $\{<\}$ and the theory of dense linear orders admits quantifier elimination, $\varphi'$ is equivalent to a quantifier-free formula $\psi$. It follows from Lemma 5.4($i$) that $\psi$ cannot distinguish between $\overline{a}$ and $\overline{b}$. As also $\varphi$ is equivalent to $\psi$, $\overline{a}$ and $\overline{b}$ cannot be distinguished by $\varphi$ either. □

In Definition 5.9 below the canonical parameters of a database will be used to define its invariant. Not only the parameters of the database but also the parameters $P^{\psi}$ used in the query are needed. Thus it has to be shown that the preceding lemma holds even if we extend the set $cp(\mathfrak{B})$.

**Proposition 5.6.** *Suppose $\mathfrak{B}$ is a database, $\overline{a}, \overline{b} \in \mathbb{R}^k$ are two $k$-tuples, and $P := cp(\mathfrak{B}) \cup P^{\psi} \cup \{0, 1\}$. Then $otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{a}) = otp^{\mathfrak{B}}_{cp(\mathfrak{B})}(\overline{b})$ if $otp^{\mathfrak{B}}_{P}(\overline{b}) = otp^{\mathfrak{B}}_{P}(\overline{a})$.*

*Proof.* The proof of the proposition follows immediately from the fact that all atomic formulae occurring in the atomic order type over $cp(\mathfrak{B})$ also occur in the atomic order type over $P$. □

For the rest of this chapter we define $P$ to be the set $\{0,1\} \cup cp(\mathfrak{B}) \cup P^{\psi}$. The constants 0 and 1 are included because they are needed in the definition of the invariant. The last thing we need for the definition of *inv* is that the set $P$ is uniformly first-order definable over $(\mathbb{R}, <, 0, 1, P^{\psi})$.

**Lemma 5.7.** *Suppose $\mathfrak{B} = (\mathfrak{A}, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$ is a database. For each $1 \leq i \leq k$ let $r_i$ be the arity of $R_i$. Then $P$ equals $\{a \ : \ \mathfrak{B} \models \delta[a]\}$, where $\delta(x)$ is defined as*

$$\begin{aligned}
\delta(x) := \bigvee_{i=1}^{k} (\exists y_1 \ldots y_{r_i} \bigvee_{j=1}^{r_i} & \exists u \exists v \ u < x \wedge x < v \wedge \\
& \forall z \exists y_1' \ldots \exists y_{r_i}' (\textstyle\bigwedge_{l=1}^{r_i} (y_i = x \to y_i' = z) \wedge (y_i \neq x \to y_i' = y_i)) \wedge \\
& \ [\ (u \leq z < x \to \neg R_i y_1 \ldots y_{j-1} z y_j \ldots y_{r_i} \\
& \quad \wedge x < z \leq v \to R_i y_1 \ldots y_{j-1} z y_j \ldots y_{r_i}) \\
& \ \vee (u \leq z < x \to R_i y_1 \ldots y_{j-1} z y_j \ldots y_{r_i} \\
& \quad \wedge x < z \leq v \to \neg R_i y_1 \ldots y_{j-1} z y_j \ldots y_{r_i})] \\
& \ \vee R_i y_1 \ldots y_{j-1} x y_j \ldots y_{r_i} \wedge \\
& \quad (u \leq z \wedge z \leq v \wedge \neg z = x) \to \neg R_i y_1 \ldots y_{j-1} z y_j \ldots y_{r_i} \\
& \ \vee \neg R_i y_1 \ldots y_{j-1} x y_j \ldots y_{r_i} \\
& \quad (\wedge u \leq z \wedge z \leq v \wedge \neg z = x) \to R_i y_1 \ldots y_{j-1} z y_j \ldots y_{r_i}\ ] \\
\vee \textstyle\bigvee_{p \in P^{\psi}} x = p \vee x = 0 \vee x = 1. &
\end{aligned}$$

*Proof.* The proof should be clear as the formula $\delta$ is essentially a first-order formalization of Definition 5.2 augmented with the formula including 0, 1, and the parameters from $P^{\psi}$. □

We are now ready to define the invariant. Define an equivalence relation $\sim$ on $\mathbb{R}$ such that two elements $a$ and $b$ are $\sim$-equivalent if and only if they realize the same 1-order type over $P$. As $P$ is first-order definable the equivalence relation $\sim$ is first-order definable as well. The set of equivalence classes $\mathbb{R}_{\sim}$ serves as the universe of the invariant. To complete the definition we have to specify the database relations.

Before we give the detailed definition of the relations in the invariant, we illustrate the idea by an example. Consider a database $\mathfrak{B}$ with a single binary relation $S$ represented by $\varphi_S(x,y) := x > 4 \wedge x < 8 \wedge y > 3 \wedge y < 6 \wedge y < x$. The relation is shown in Figure 5.1. The set $cp(\mathfrak{B})$ consists of the four elements $\{3,4,6,8\}$. Thus there are nine different $\sim$-equivalence classes, namely the intervals $(-\infty,3), \{3\}, (3,4), \{4\}, (4,6), \{6\}, (6,8), \{8\}$, and $(8,\infty)$. Recall that these equivalence classes form the universe of the invariant. Thus the relation $S$ has somehow to be defined in terms of these classes. Obviously it is not enough to factorize $S$ by $\sim$, because as $5 \sim 5.1$, the equivalence classes $[5]$ and $[5.1]$ are equal, but $([5.1],[5]) \in S$ and $([5],[5]) \notin S$. Thus $S_{/\sim}$ would not be well-defined.

Instead of simply factorizing a $k$-ary relation $R$ by $\sim$ we consider the set $C_R$ of $(k+1)$-tuples $([a_1], \ldots, [a_k], \rho)$, where $[a_i] \in \mathbb{R}_{/\sim}, 1 \leq i \leq k$ and $\rho$ denotes a $k$-order type, such that $([a_1], \ldots, [a_k], \rho) \in C_R$ if and only if there is a $\overline{b} \in \mathbb{R}^k$ realizing $\rho$ such that $R\overline{b}$ holds and $a_i \sim b_i$ for all $1 \leq i \leq k$. In the example above, the set $C_S$ consists of the set of all triples $([a_1],[a_2],\rho)$ such that $[a_1] \times [a_2]$ is in the rectangle marked by the dashed line in Figure 5.1 and $\rho$ is the order type $x < y$.

The idea behind the definition of the relation in the invariant is to use the set $C_R$ as a finite relation carrying all the information necessary to restore the original database

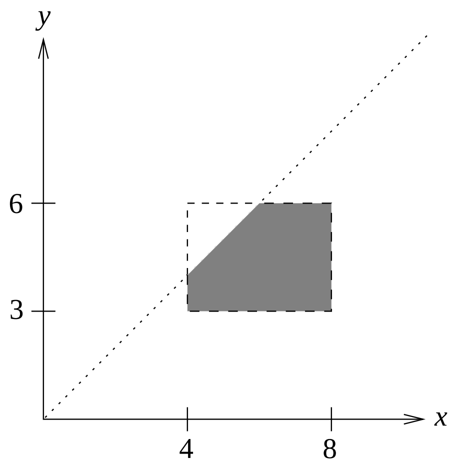



Figure 5.1: Relation $S$ represented by $x > 4 \wedge x < 8 \wedge y > 3 \wedge y < 6 \wedge y < x$.

relation $R$. Note that the number of different $k$-order types is finite for all $k$. Thus we can assign to each order type a finite word $\xi(p)$ over $\{0, 1\}$. Once such an injection $\xi$ is fixed, the set $C_R$ can be rewritten to a set $C'_R := \{([a_1], \dots, [a_k], \bar{t}) \ : \ ([a_1], \dots, [a_k], \rho) \in C_R$ and $\xi(\rho) = \bar{t}\}$. This gives the definition of the relations in the invariant.

**Definition 5.8.** For each $k \in \mathbb{N}$ let $ord(k)$ be the set of $k$-order types and $on(k) = \lceil \log(|ord(k)|) \rceil$. Fix for each $k \in \mathbb{N}$ an injection $\xi_k$ taking $ord(k)$ to the set $\{0, 1\}^{on(k)}$. For $k = 2$ we define $\xi_2$ to be the injection taking $x < y$ to 00, $x = y$ to 01, and $y < x$ to 10.

**Definition 5.9.** Let $\sigma := \{R_1, \dots, R_k\}$ be a relational signature where each $R_i$ is of arity $r_i$. Suppose $\mathfrak{B}$ is a $\sigma$-database over $\mathfrak{A}$. The *invariant* $\mathfrak{B}'$ of $\mathfrak{B}$ is a finite structure with universe $U$ over the signature $\{<, R'_1, \dots, R'_k\}$, where

- $U := \mathbb{R}_{/\sim}$,
- $[x] < [y]$ if and only if $x < y$ and $x \not\sim y$, and
- $R'_i$ is of arity $r_i + on(k)$ and defined as: $R_i'^{\mathfrak{B}'}[a_1] \dots [a_k] t_1 \dots t_{on(k)}$ iff there are $b_1, \dots, b_k \in \mathbb{R}$ with $\xi_k(otp(\bar{b})) = t_1, \dots, t_{on(k)}$ so that $R_i^{\mathfrak{B}} b_1 \dots b_k$ and $[a_i] = [b_i]$ for $1 \leq i \leq k$.

The mapping *inv* is defined as the function taking databases to their invariants.

Note that this idea of a finite encoding of finitely representable relations is similar to the encoding given by Belegradek, Stolboushkin and Taitslin in [BST98]. The main difference is that they only coded the relations but kept the infinite universe, whereas here both, the relations and the universe are finite.

Having defined the invariant of a database, we have to explain how the query has to be transformed for evaluation in the invariant. This translation of the formulae follows the same ideas described above, namely to increase the arity of the relations to store the order type. While translating a formula with free variables $\{x_1, \dots, x_k\}$ we introduce new free variables $\bar{i}$ to hold the order type.

In Definition 5.11 below we define the transformation of formulae according to their structure. In this transformation we need to compare order types over a different number of variables. In particular we need to check if for two order types $\rho_1$ and $\rho_2$, $\rho_1 \subseteq \rho_2$ holds.

**Definition 5.10.** Suppose $\rho_1$ is a $k_1$-order type in the variables $x_1, \dots, x_{k_1}$ and $\rho_2$ a $k_2$-order type in the variables $x_1, \dots, x_{k_2}$, where $k_1 < k_2$. $\rho_2$ *extends* $\rho_1$, if $\rho_1 \subseteq \rho_2$.

This means that the order type $\rho_2$ behaves on $x_1, \dots, x_{k_1}$ in the same way as $\rho_1$. In the query transformation we need a formula $extends_{k_1 k_2}(\bar{i}, \bar{j})$ stating that $\bar{i} := i_1, \dots, i_{on(k_1)}$ codes a $k_1$-order type $\rho_1$, $\bar{j} := j_1, \dots, j_{on(k_2)}$ a $k_2$-order type $\rho_2$, and $\rho_2$ extends $\rho_1$. The formula is defined as

$$extends_{k_1 k_2}(\bar{i}, \bar{j}) := \bigvee_{\rho_2 \in ord(k_2)} (\xi_{k_2}(\rho_2) = \bar{j} \rightarrow \bigvee_{\substack{\rho_1 \in ord(k_1) \\ \rho_2 \text{ extends } \rho_1}} \xi_{k_1}(\rho_1) = \bar{i}).$$

We are now ready to define the transformation of queries.

**Definition 5.11.** Suppose $\sigma$ is a database schema and $\tau$ the signature of the invariants corresponding to $\sigma$-databases. Further, let $\mathcal{L}$ be a logic from $\{$*FO*, *FO+DTC*, *FO+TC*, *FO+LFP*, *FO+PFP*$\}$. Suppose $\psi(x_1, \dots, x_k) \in \mathcal{L}[\sigma]$ is a query. The map $f : \mathcal{L}[\sigma] \rightarrow \mathcal{L}[\tau]$ is defined inductively as follows.

- $\psi(x, y) := x < y$.
  $(f\psi)(x, y, i_1, i_2) := x \leq y \wedge i_1 = 0 \wedge i_2 = 0.$
- $\psi(x, y) := x = y$.
  $(f\psi)(x, y, i_1, i_2) := x = y \wedge i_1 = 0 \wedge i_2 = 1.$
- $\psi(x) := x = c$.
  $(f\psi)(x, i) := x = [c] \wedge i = 0.$
- $\psi(x) := x < c$.
  $(f\psi)(x, i) := x < [c] \wedge i = 0.$
- $\psi(x_1, \dots, x_l) := R_i u_1 \dots u_k$, where $k = ar(R_i)$. The $u_i$ are either constants or variables from $\{x_1, \dots, x_l\}$ and all $x_i$ occur in $\{u_1, \dots, u_k\}$.
  $(f\psi)(x_1, \dots, x_l, i_1, \dots, i_{on(l)}) := R_i v_1 \dots v_k \bar{i}$, where $v_r := \begin{cases} x_s & \text{if } u_r = x_s, \\ [c] & \text{if } u_r = c. \end{cases}$
- $\psi(x_1, \dots, x_k) := \psi_1(y_1, \dots, y_{k_1}) \wedge \psi_2(z_1, \dots, z_{k_2})$, where all $y_i$ and $z_i$ occur in $\bar{x}$. Let $\bar{i} := i_1, \dots, i_{on(k)}$, $\bar{j} := j_1, \dots, j_{on(k_1)}$, and $\bar{j}' := j'_1, \dots, j'_{on(k_2)}$. Then
  $(f\psi)(\bar{x}, \bar{i}) := \exists \bar{j} \exists \bar{j}'\ extends_{k_1 k}(\bar{j}, \bar{i}) \wedge extends_{k_2 k}(\bar{j}', \bar{i}) \wedge (f\psi_1)(\bar{y}, \bar{j}) \wedge (f\psi_2)(\bar{z}, \bar{j}').$
- The disjunction case is defined analogously.

- $\psi := \neg\varphi$.
  $(f\psi) := \neg(f\varphi)$.

- $\psi(x_1, \ldots, x_k) := \exists y\ \varphi(\overline{x}, y)$.
  $(f\psi)(x_1, \ldots, x_k, \overline{i}) := \exists y \exists j_1, \ldots, \exists j_{on(k+1)} extend_{k(k+1)}(\overline{i}, \overline{j}) \wedge (f\varphi)(\overline{x}, y, \overline{j})$.

- $\psi(\overline{u}, \overline{v}) := [DTC_{\overline{x},\overline{y}}\varphi(\overline{x}, \overline{y})](\overline{u}, \overline{v})$.
  $(f\psi)(\overline{u}, \overline{v}, \overline{i}) := [DTC_{\overline{x},\overline{y},\overline{j}}(f\varphi)(\overline{x}, \overline{y}, \overline{j})](\overline{u}, \overline{v}, \overline{i})$.

- The rule for the $TC$-operator is defined analogously.

- $\psi(\overline{u}) := [LFP_{R,\overline{x}}\varphi(R, \overline{x})](\overline{u})$.
  $(f\psi)(\overline{u}, \overline{i}) := [LFP_{R',\overline{x},\overline{j}}(f\varphi)(R', \overline{x}, \overline{j})](\overline{u}, \overline{i})$.

- The rules for the $IFP$- and $PFP$-operators are defined analogously.

Now almost all parts of the query evaluation method are defined. A query formula $\varphi$ on a database $\mathfrak{B}$ can be evaluated by evaluating $f(\varphi)$ in $inv(\mathfrak{B})$ using the methods for finite databases. What is left to be done is to define the way back.

**Definition 5.12.** Suppose $S$ is a $(k + on(k))$-ary answer of a query on an invariant. The map $\hat{\pi}$ maps $S$ to a formula $\varphi_S$ representing the corresponding relation on the original database. $\hat{\pi}$ is defined as

$$\hat{\pi} : S \mapsto \varphi_S(x_1, \ldots, x_k) := \bigvee_{\overline{a}\overline{t} \in S} (\sigma_k(\overline{x}, \overline{t}) \wedge \bigwedge_{j=1}^{k} (x_j \sim a_j)),$$

where $\sigma_k(\overline{x}, \overline{i})$ is a formula stating that $\overline{x}$ satisfies the order type specified by $\overline{i}$. The map $\pi$ mapping relations to finitely representable relations is defined as

$$\pi : S \mapsto \{\overline{a} \ : \ \mathfrak{A} \models \hat{\pi}(S)[\overline{a}]\}.$$

All parts of the evaluation algorithm have now been defined. The next theorem proves its correctness.

**Theorem 5.13.** *Suppose $\mathfrak{B}$ is a database over $\mathfrak{A}$ and $\psi$ is a query. Let $\mathfrak{B}' := inv(\mathfrak{B})$ be the invariant corresponding to $\mathfrak{B}$. Then*

$$\psi^{\mathfrak{B}} = \pi((f\psi)^{\mathfrak{B}'}).$$

*Proof.* The proof is by induction on the structure of the query.

- Suppose $\psi(x, y) := x < y$. $\psi^{\mathfrak{B}}$ is the set of pairs $(a, b) \in \mathbb{R}^2$ such that $a < b$. By Definition 5.11, $f(\psi)$ is defined as $x \leq y \wedge i_1 = 0 \wedge i_2 = 0$. Evaluating $(f\psi)^{\mathfrak{B}'}$ results in the set $C := \{(a, b, i_1, i_2) \ : \ a \leq b, i_1 = 0, i_2 = 0\}$. Transforming this set with the mapping $\hat{\pi}$ yields the formula $\varphi_C(x, y) := \bigvee_{(a,b,i_1,i_2) \in C}(\sigma_2(x, y, i_1, i_2) \wedge x \sim a \wedge y \sim b)$. As $i_1$ and $i_2$ are 0 for all tuples $(a, b, i_1, i_2) \in C$, $\sigma_2(x, y, i_1, i_2)$ reduces to $x < y$ and thus $\pi(C)$ equals $\{(a, b) \in \mathbb{R}^2 \ : \ a < b\}$.

- The case where $\psi$ is of the form $\psi(x, y) := x = y$ can be treated analogously.

- Suppose $\psi(x) := x = c$. Then $(f\psi)(x, i) := x = [c] \wedge i = 0$ and $(f\psi)^{\mathfrak{B}'}$ evaluates to the set $C := \{([c], 0)\}$. Thus $\hat{\pi}(C)$ results in the formula $\varphi(x) := \sigma_1(x, 0) \wedge x \sim c$. $\varphi$ is satisfied only by $c$ because $c \in P$ and therefore the only member of $[c]$ is $c$ itself. We get $\pi(C) := \{c\} = \psi^{\mathfrak{B}}$.

- The case $\psi(x) := x < c$ can be treated in the same way.

- Suppose $\psi(x_1, \ldots, x_l) := R_s u_1 \ldots u_k$ as in Definition 5.11. We assume w.l.o.g. that the first arguments of the relation are the variables and the parameters come thereafter, that is $u_1 = x_1, \ldots, u_l = x_l$ and $u_{l+1} = c_1, \ldots, u_k = c_{k-l}$. The transformed query is $(f\psi)(x_1, \ldots, x_l, \bar{i}) := R'_s x_1 \ldots x_l [c_1] \ldots [c_{k-l}]\bar{i}$. Evaluating $f(\psi)$ in $\mathfrak{B}'$ yields the set $C := \{([a_1], \ldots, [a_l], [c_1], \ldots, [c_{k-l}], \bar{i}) \in R_s'^{\mathfrak{B}'}\}$. Now we have to show that $\pi(C) = \psi^{\mathfrak{B}}$.
For the forth direction suppose that $(a_1, \ldots, a_k) \in \pi(C)$. Then there is a disjunct $\varphi := \sigma_k(x_1, \ldots, x_k, \bar{i}) \wedge \bigwedge_r (x_r \sim b_r)$ in $\hat{\pi}(C)$ with $(\bar{b}, \bar{i}) \in C$ and $\mathfrak{B} \models \varphi(\bar{a})$. As $(\bar{b}, \bar{i}) \in R'^{\mathfrak{B}'}$ and therefore, by Definition 5.9, $(a_1, \ldots, a_k) \in R^{\mathfrak{B}}$ we get $\bar{a} \in \psi^{\mathfrak{B}}$.
For the back direction suppose that $(a_1, \ldots, a_k) \in R^{\mathfrak{B}}$. Then $([a_1], \ldots, [a_k], \bar{i})$ is in $R'^{\mathfrak{B}'}$, where $\xi_k(otp(\bar{a})) = \bar{i}$, and $\sigma_k(\bar{x}, \bar{i}) \wedge \bigwedge_r a_r \sim x_r$ occurs as a disjunct in $\hat{\pi}(C)$. Obviously this formula is satisfied by $\bar{a}$ and therefore $\bar{a} \in \pi(C)$.

- Suppose $\psi(\bar{x}) := \psi_1(\bar{y}) \wedge \psi_2(\bar{z})$, where all $y_i$ and $z_i$ occur in $\bar{x}$. The set $\psi^{\mathfrak{B}}$ consists of all tuples $\bar{a}$ so that the corresponding parts of $\bar{a}$ are in $\psi_1^{\mathfrak{B}}$ and $\psi_2^{\mathfrak{B}}$. By induction we have $\pi((f\psi_1)^{\mathfrak{B}'}) = \psi_1^{\mathfrak{B}}$ and $\pi((f\psi_2)^{\mathfrak{B}'}) = \psi_2^{\mathfrak{B}}$. It follows that also $\psi^{\mathfrak{B}} = \pi((f\psi)^{\mathfrak{B}'})$.

- The other boolean cases can be proven analogously.

- Suppose $\psi(x_1, \ldots, x_k) := \exists y\, \varphi(\bar{x}, y)$. The transformed formula $(f\psi)$ is defined as $(f\psi)(\bar{x}, \bar{i}) := \exists y \exists j_1, \ldots, j_{on(k+1)} extend_{k(k+1)}(\bar{i}, \bar{j}) \wedge (f\varphi)(\bar{x}, y, \bar{j})$.
Suppose that $(a_1, \ldots, a_k) \in \psi^{\mathfrak{B}}$. This is the case if and only in there is an $a_{k+1}$ with $(a_1, \ldots, a_k, a_{k+1}) \in \varphi^{\mathfrak{B}}$. By induction $\varphi^{\mathfrak{B}} = \pi((f\varphi)^{\mathfrak{B}'})$. Thus there is a tuple $([a_1], \ldots, [a_{k+1}], \bar{j}) \in (f\varphi)^{\mathfrak{B}'}$ and $(a_1, \ldots, a_{k+1})$ satisfies the $(k+1)$-order type $\rho$ denoted by $\bar{j}$. This is the case if and only if there is a tuple $([a_1], \ldots, [a_k], \bar{i}) \in (f\psi)^{\mathfrak{B}'}$ such that $\rho$ extends the order type denoted by $\bar{i}$. Thus we get that $(a_1, \ldots, a_k) \in \psi^{\mathfrak{B}}$ if and only if $([a_1], \ldots, [a_k], \bar{i}) \in (f\psi)^{\mathfrak{B}'}$, where $(a_1, \ldots, a_k)$ satisfies the order type denoted by $\bar{i}$. This implies $\psi^{\mathfrak{B}} = \pi((f\psi)^{\mathfrak{B}'})$.

- $\psi(\bar{u}) := [LFP_{R,\bar{x}}\varphi(R, \bar{x})](\bar{u})$.
$f(\psi)(\bar{u}, i) := [LFP_{R',\bar{x},j}(f\varphi)(R', \bar{x}, j)](\bar{u}, i)$.
We assume in the proof that $\varphi$ does not include an LFP-operator. This is no restriction of generality because the number of LFP-operators occurring in the query is fixed. Thus they can be eliminated one by one starting with the innermost.
The proof is by induction on the steps of the fixed-point evaluation. We denote by $R_i$ the value of $R$ after the $i$th induction step and show that after each step $R_i = \pi(R'_i)$ holds.
The proof for the 0-th induction step is straightforward, because $\varphi$ can be seen as a

query on an expansion of $\mathfrak{B}$ by the empty relation $R$. It follows from the induction on the structure of $\varphi$ that $R_0 = \pi(R'_0)$.
Now suppose that after the $n$-th step $R_n = \pi(R'_n)$. Again $\varphi$ can be seen as a query on the expansion of $\mathfrak{B}$ by $R_n$ and $f(\varphi)$ as a query on the invariant expanded by $R'_n$. By induction assumption we have $R_n = \pi(R'_n)$ and therefore the expanded invariant is again the invariant of the expanded database. Now the induction hypothesis follows by the induction on the structure of $\varphi$.

- The other cases can be proven in the same way.

□

Now that we have given an evaluation method for queries on dense linear order databases we are interested in the data complexity of such queries. In the next section we will prove upper complexity bounds; we show them to be optimal in the section thereafter.

## 5.2 Data complexity

The complexity of the evaluation method defined above depends on the three different tasks involved in the algorithm. First of all the invariant is created. We will show this to be in LOGSPACE. This done, the query can be evaluated in the invariant. The complexity of this task is well understood as it is just a query on a finite ordered database. Thus we can use the results of descriptive complexity theory for this part of the evaluation process. The third task in the algorithm is to transform the answer of the evaluation in the invariant back to the answer on the database. We will show this to be in LOGSPACE as well. Taken together, the complexity of the complete algorithm is determined by the complexity of the second task, because all the logics we are interested in have a data complexity on finite databases of at least LOGSPACE. We get that the change from finite to dense order databases does not increase the evaluation complexity. The next lemmas prove the LOGSPACE results for the first and the last part. The following theorem gives the complexity of the whole evaluation process.

**Lemma 5.14.** *The map inv can be calculated in* LOGSPACE, *that is given a representation of a constraint database, its invariant can be constructed in logarithmic space.*

*Proof.* We split the construction of the invariant in two separate algorithms. In the first the invariant's universe and in the second the relations are constructed.

In Lemma 5.7 it is proven that the set $P$ can be defined by a first-order formula. The set of canonical parameters is extracted from the input database in the following way. For every parameter $p$ occurring in the input we use the formula $\delta(x)[x/p]$ as a boolean query on the database. If the query evaluation returns *true* then $p$ is a canonical parameter. As first-order queries can be evaluated in LOGSPACE (see [KKR90]), we get that the set $P$ can be constructed in LOGSPACEas well. To set up the universe $U$ of the invariant we assume that our Turing machines have an operation *mean* taking the mean of two elements. Besides this we need two other operations $+1$ and $-1$ with the obvious semantic. Now we use these operations to construct the invariant's universe. The Turing machine takes every pair $u, v$ from $P$ such that $v$ is the successor of $u$ in $P$ and uses *mean* to find an element $m_{u,v}$ between $u$ and $v$. The universe of the invariant consists of all these new elements $m_{u,v}$ together with the set $P$ and the elements $l-1$ and $b+1$, where $l$ is the smallest and $b$ the biggest element in $P$. Obviously all the operations involved, and thus the construction of the universe, can be done in LOGSPACE.

Now that the universe has been constructed, the relations have to be built up. A relation $R'$ in the invariant of arity $k + on(k)$ can be constructed by taking the set of tuples $\overline{a}, \overline{i}$, where $\overline{a} \in U^k$ and $\overline{i}$ codes a $k$-order type, such that the query

$$\exists \overline{x} (\bigwedge_{i=1}^{k} x_i \sim a_i \wedge \sigma_k(\overline{x}, \overline{i}) \wedge R\overline{x})$$

on the database evaluates to *true*. Each query is first-order and can be evaluated in LOGSPACE. As the space can be reused, the whole relation can be constructed in LOGSPACE. Thus the second sub-algorithm a LOGSPACE-algorithm.

As the composition of LOGSPACE-algorithms is also in LOGSPACE, we get that the combined algorithm constructing the invariant is also in LOGSPACE. □

**Lemma 5.15.** *The map $\hat{\pi}$ can be calculated in* LOGSPACE.

*Proof.* Suppose $\mathfrak{B}'$ is the invariant of a dense linear order database and $R$ an answer relation obtained by a query $\varphi$ on $\mathfrak{B}'$. All the algorithm to calculate $\hat{\pi}(R)$ has to do is to output the disjunction of the formulae $(\sigma_k(\overline{x}, \overline{i}) \wedge \bigwedge_{j=1}^{k}(x_j \sim a_j))$ for every tuple $\overline{a}\overline{i} \in R$. Clearly, this can be done in LOGSPACE. □

Now we have the complexity of the three parts of the evaluation algorithm. The following theorem puts the parts together and gives an upper bound for the complexity of various dense order query languages.

**Theorem 5.16.** *Suppose $\mathcal{L} \in \{FO, FO{+}DTC, FO{+}TC, FO{+}LFP, FO{+}IFP, FO{+}PFP\}$ is a logic and $\mathfrak{C}$ a complexity class so that the evaluation problem for $\mathcal{L}$ on finite databases is in $\mathfrak{C}$. Then the evaluation problem for $\mathcal{L}$ on dense linear order databases is also in $\mathfrak{C}$.*

The theorem above gives us the upper data complexity bounds for query evaluation. In the next section we show them to be optimal.

## 5.3 Capturing complexity classes

As in the previous sections we use the invariant to lift the capturing results for finite structures from descriptive complexity theory to dense linear order databases. In the proof of the capturing results a transformation of formulae over the invariant to formulae over the database is needed. In a way this forms the opposite direction we took in the first section of this chapter. We will define this mapping by means of a first-order interpretation $\Gamma$ of the invariant in the database (see [Hod97]).

**Definition 5.17.** To ease notation we denote by $\mathfrak{B}$ the database $\mathfrak{B} := (\mathfrak{A}, R_1^{\mathfrak{B}}, \dots, R_k^{\mathfrak{B}})$, by $\sigma$ its signature, by $\mathfrak{B}'$ its invariant $inv(\mathfrak{B})$, and by $\tau$ the signature of the invariant. The interpretation $\Gamma$ interpreting $\mathfrak{B}'$ in $\mathfrak{B}$ is given by

- the domain formula $\delta_\Gamma(x) := true$,
- a surjective map $f_\Gamma : \mathbb{R} \to U$ defined as $f_\Gamma(x) := [x]$, and
- for each atomic formula $\psi \in FO[\tau]$ a formula $\psi_\Gamma \in FO[\sigma]$.
  A formula $u = v \in FO[\tau]$ corresponds to $u \sim v$, where $u, v$ denote either variables or parameters from $P$. The formulae for all other atomic formulae can be given according to Definition 5.9. That is, a formula $u < v \in FO[\tau]$ corresponds to $u < v \wedge \neg u \sim v$ and $R'_s\overline{x}\overline{i}$ to $\exists \overline{y}\, R_s\overline{y} \wedge \sigma_{ar(R_s)}(\overline{y}, \overline{i}) \wedge \bigwedge_j (x_j \sim y_j)$. Recall the definition of $\sigma_k$ from Definition 5.12.

Clearly, for all atomic formulae $\psi \in FO[\tau]$, $\mathfrak{B}' \models \psi[f_\Gamma\overline{a}]$ if and only if $\mathfrak{B}' \models \psi_\Gamma[\overline{a}]$, because the correspondence between formulae given in the previous definition is only a formalization of Definition 5.9. The last thing needed to prove the capturing results is a generalization of the reduction lemma given in [Hod97] to transitive closure and fixed-point logic.

**Lemma 5.18 (generalized reduction lemma).** *Suppose $\mathfrak{B}$ is a $\sigma$-structure, $\mathfrak{B}'$ a $\tau$-structure and $\Gamma$ a 1-dimensional interpretation of $\mathfrak{B}'$ in $\mathfrak{B}$. Let $\mathcal{L}$ be a logic from {FO, FO+DTC, FO+TC, FO+LFP, FO+IFP, FO+PFP}. Then for every formula $\psi(\overline{y})$ of the language $\mathcal{L}[\tau]$ there is a formula $\psi_\Gamma(\overline{x})$ of the language $\mathcal{L}[\sigma]$, such that for all $\overline{a} \in (\delta_\Gamma(A))^n$,*

$$\mathfrak{B}' \models \psi(f_\Gamma\overline{a}) \Longleftrightarrow \mathfrak{B} \models \psi_\Gamma(\overline{a}).$$

*Proof.* We extend $\psi_\Gamma$ given in Definition 5.17 by the following rules:

- $(\neg\psi)_\Gamma = \neg(\psi_\Gamma)$.
- $(\psi_1 \wedge \psi_2)_\Gamma = (\psi_1)_\Gamma \wedge (\psi_2)_\Gamma$.
- $(\exists y \psi)_\Gamma = \exists x(\delta_\Gamma(x) \wedge \psi_\Gamma)$.
- $([LFP_{R,x_1,\dots,x_k}\psi)_\Gamma = [LFP_{R,x_1,\dots,x_k}\psi_\Gamma]$.

The rules for the other operators like the $DTC$ or $PFP$ operator are analogous to the $LFP$ case. The first-order cases are already proven in the reduction lemma, whereas the $LFP$ case follows from the same argument as in Theorem 5.13. ☐

We are now ready to prove the theorem lifting the capturing results over finite structures to dense linear order databases. We state the theorem explicitly only for *FO+LFP* but the proof works exactly the same way for *FO+DTC*, *FO+TC*, *FO+IFP*, and *FO+PFP*.

**Theorem 5.19.** *FO+LFP captures* PTIME *in the context of dense linear orders.*

| | | |
|---|---|---|
| *FO+DTC* | = | LOGSPACE |
| *FO+TC* | = | NLOGSPACE |
| *FO+LFP* | = | PTIME |
| *FO+PFP* | = | PSPACE |

Table 5.1: Logics and complexity classes in the context of dense linear orders.

*Proof.* We have already proven that $FO{+}LFP \subseteq$ PTIME. To show the other direction suppose that $\mathfrak{B}$ is a dense linear order database and $Q$ a PTIME query. Then there is a PTIME-algorithm $M_Q$ which takes a database $\mathfrak{B}$ as input and returns $Q(\mathfrak{B})$ as answer. Again let $\sigma$ be the database signature and $\tau$ be the signature of the corresponding invariants. We now show that there is an $FO{+}LFP[\sigma]$-formula $\psi_Q$ defining $Q$.

We define an algorithm $M$ which takes an invariant $inv(\mathfrak{B})$ of a database $\mathfrak{B}$ as input and returns $Q(\mathfrak{B})$ as output. The algorithm $M$ operates as follows. First it reconstructs a representation of the database $\mathfrak{B}$ whose invariant is given as input. Afterwards it executes the algorithm $M_Q$ on the representation. $M$ operates in PTIME, because the representation of the database can be constructed in polynomial time and the algorithm $M_Q$ is by assumption a PTIME-algorithm. Note that in contrast to the algorithm of the previous section this algorithm constructs the database from the invariant and evaluates the query in the database, whereas the algorithm in the previous section constructs the invariant from the database and then operates on the invariant.

Note that $M$ takes a finite structure as input and is itself a PTIME-algorithm. Thus, by the result of Immerman [Imm86] and Vardi [Var82], there is an $FO{+}LFP[\tau]$-formula $\varphi$ equivalent to $M$. Further, we proved in Lemma 5.18 that there is a formula $\varphi_\Gamma \in FO{+}LFP[\sigma]$ such that for all $\overline{a} \in \mathbb{R}^n$ $inv(\mathfrak{B}) \models \varphi([\overline{a}])$ iff $\mathfrak{B} \models \varphi_\Gamma(\overline{a})$. Thus $\mathfrak{B} \models \varphi_\Gamma(\overline{a})$ if and only if $\overline{a} \in Q(\mathfrak{B})$. This proves the theorem. $\square$

As stated above the proof of the theorem works in exactly the same way for the other logics mentioned above. Thus we get the relations between logics and complexity classes in the context of dense linear orders summarized in Table 5.1.

# Chapter 6

# Discrete linear orders

In this chapter we examine the complexity of first-order queries on discrete linear order databases. Recall from Section 4.3.1 that $(\mathbb{N}, <)$ does not admit quantifier elimination and is therefore not suitable as a context structure. As explained there, quantifier elimination can be obtained by adding a countable set of relations $\{<_c : c \in \mathbb{N}\}$, where the intended interpretation of $x <_c y$ is $x + c < y$. For convenience we add the relation $<_{-1}$ as an abbreviation for $\leq$. Note that $<_0$ is simply the usual order relation.

In Section 4.3.1 we examined the evaluation complexity for different logics on gap-order databases but left open the first-order case which is the topic of this chapter. We will prove a LOGSPACE upper bound for the evaluation complexity of first-order queries.

## 6.1 First-order query evaluation

Let $\tau := \{<_c : \ c \in \mathbb{N} \cup \{-1\}\}$ be the signature of gap-orders. For the rest of this chapter assume a fixed query $\psi(x_1, \dots, x_s)$ and a database $\mathfrak{B} = (\mathfrak{A}, R_1^{\mathfrak{B}}, \dots, R_k^{\mathfrak{B}})$ over the $\tau$-structure $\mathfrak{A} = (\mathbb{Z}, (<_c)_{c \in \mathbb{N} \cup \{-1\}})$. Further, we assume that the representation of $\mathfrak{B}$ is in disjunctive normal form whereas the query does not have to be in some normal form. We consider databases over the integers instead of the natural numbers because it makes the notation for the algorithms easier. But the main result also holds for databases with the natural numbers as universe because $\mathbb{N}$ is definable in $\mathbb{Z}$ by the formula $x \geq 0$.

We assume w.l.o.g. that every atomic formula occurring in the representation of $\mathfrak{B}$ is of the form $x <_c y$, $x = y$, $x < a$, $a < x$, or $x = a$, where $a$ is a parameter. Formulae of the kind $x \leq a$, $x <_c a$, or $a <_c x$, where $c \neq 0$, can easily be converted to equivalent formulae of the type $x < a'$ or $a' < x$.

The goal of this chapter is to prove the following theorem.

**Theorem.** *First-order queries on gap-order constraint databases can be evaluated in* LOGSPACE.

At the end of this section we define an algorithm to evaluate first-order queries inductively over the structure of the query. The algorithm evaluates the query from the inside out. As explained above it is only called on databases whose representations are in disjunctive normal form. While evaluating the query it keeps the evaluation results in dnf.

Therefore the following algorithms dealing with the different logical operations suppose that their input is also in disjunctive normal form. As we will see, the only complicated operations are existential quantification and negation. We first give two sub-algorithms dealing with these cases.

### 6.1.1 Handling quantification

The next algorithm will be used to eliminate one existential quantifier. It takes a conjunction $\Psi$ of atomic or negated atomic formulae together with a variable $z$ as input and returns a quantifier-free formula equivalent to $\exists z\Psi$ in which $z$ does not occur. The algorithm is only called on conjunctions of literals because the evaluation algorithm deals only with formulae in dnf and, as $\exists z(\bigvee_i \varphi_i) \equiv \bigvee_i(\exists z\varphi_i)$, the quantifier can independently be eliminated in the disjuncts. The resulting formula is also a conjunction of literals. To ease notation we neglect the difference between a conjunction of literals and the set of all literals occurring in it.

**Algorithm 6.1. eliminate**$(\Psi, z)$

**Input:** *A variable $z$ and a conjunction $\Psi$ of atomic or negated atomic formulae over the signature $\tau$.*

**Output:** *A conjunction $\Psi'$ of atomic or negated atomic formulae equivalent to $\exists z\Psi$.*

- *If there is no literal in $\Psi$ containing $z$ return $\Psi' := \Psi$.*
- *If there is a formula of the type $z = a$ in $\Psi$, where $a \in \mathbb{Z}$, then do the following. If there is a formula $z = b$ with $a \neq b$, a formula $z < b$ with $b \leq a$, or a formula $b < z$ with $a \leq b$ in $\Psi$, then return false. Otherwise return $\Psi' := \Psi[z/a]$.*
- *If there is only one literal $\varphi \in \Psi$ containing $z$, then check whether $\varphi$ is satisfiable in $\mathfrak{A}$. Return $\Psi' :=$ false if not and $\Psi \backslash \{\varphi\}$ if it is.*
- *If there is more than one literal in $\Psi$ containing $z$ do the following three steps.*
  - *(i) Let $\Phi'$ be the set of formulae of the type $z < a$ or $a < z$. If $\exists z\Phi'$ is* false *in $\mathfrak{A}$ then return $\Psi' :=$ false.*
  - *(ii) Let $\Phi_z$ be the set of literals in $\Psi$ containing $z$. We build a set $\Phi$ containing the literals of $\Psi$ together with some new literals generated as follows. For each pair of formulae $(\varphi, \psi) \in \Phi_z$ do the following steps.*
    - *If $\varphi = z <_c x$ then do one of the following.*
      - *If $\psi$ is of the form $y <_{c'} z$ add a new formula $y <_{c+c'+1} x$ to $\Phi$.*
      - *If $\psi = \neg z <_{c'} y$ then if $c' > c$ add a formula $\neg x <_{c'-c-2} y$ to $\Phi$. Otherwise add $y <_{c'-c} x$.*
    - *If $\varphi = x <_c z$ and $\psi = \neg y <_{c'} z$ add a formula $\neg y <_{c'-c-2} x$ to $\Phi$ if $c' > c$. If $c' \leq c$ add $x <_{c-c'+1} y$.*

- *If* $\varphi = \neg z <_c x$ *and* $\psi = \neg y <_{c'} z$ *add a formula* $\neg y <_{c+c'-1} x$ *to* $\Phi$.
- *If* $\varphi = z < a$, *where* $a \in \mathbb{Z}$, *then do one of the following. If* $\psi = x <_c z$ *add* $x < a - c - 1$ *to* $\Phi$. *If* $\psi = \neg z <_c x$ *add* $x < a + c$ *to* $\Phi$.
- *If* $\varphi = a < z$ *then if* $\psi = z <_c x$ *add* $a + c + 1 < x$ *to* $\Phi$. *Otherwise if* $\psi = \neg x <_c z$ *add* $a - c < x$.
- *If* $\varphi$ *and* $\psi$ *are not of one of the above forms, do nothing.*

*(iii) For every* $\varphi \in \Phi - \Psi$ *is satisfiable in* $\mathfrak{A}$ *return* $\Psi' := \Phi - \Phi_z$. *Otherwise return* $\Psi' :=$ false.

The steps in part *(ii)* of the algorithm are used to eliminate the atomic formulae containing $z$. In the sequel they are therefore called elimination steps. Note that the elimination steps do not cover all pairs of formulae $(\varphi, \psi) \in \Phi_z$, that is there are pairs of formulae where no elimination step is defined for. We explain later why this is not necessary.

Having defined the algorithm we have to prove its correctness. This is done in the following lemma.

**Lemma 6.2.** *Suppose* $\Psi$ *is a conjunction of atomic or negated atomic formulae in the variables* $\overline{x}, z$ *and* $\Psi' :=$ *eliminate*$(\Psi, z)$. *Then*

$$\exists z \Psi \equiv_{\mathfrak{A}} \Psi',$$

*where* $\equiv_{\mathfrak{A}}$ *means equivalence in* $\mathfrak{A}$.

To prove the lemma we need a few technical sub-lemmas. The first two lemmas prove that the elimination steps are correct. This is done in two steps. In the first we prove that if $\psi_1$ and $\psi_2$ are eliminated to $\varphi$ then $\exists z(\psi_1 \wedge \psi_2) \equiv_{\mathfrak{A}} \varphi$. In the second step we show that if no elimination step is defined for the pair $(\psi_1, \psi_2)$ then all pairs of elements satisfy $\exists z(\psi_1 \wedge \psi_2)$.

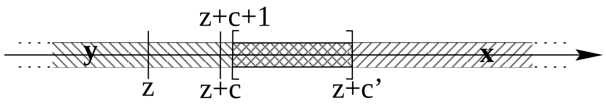
**Lemma 6.3.** *Suppose* $\psi_1$ *and* $\psi_2$ *are two literals containing* $z$ *and* $\varphi$ *is the result of an elimination step on* $\psi_1$ *and* $\psi_2$. *Then for all* $a_1, a_2 \in \mathbb{Z}$,

$$\mathfrak{A} \models \exists z(\psi_1 \wedge \psi_2)[a_1, a_2] \text{ iff } \mathfrak{A} \models \varphi[a_1, a_2].$$

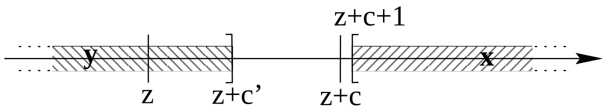
*Proof.* We demonstrate the proof for the various sub-cases by two examples. The other cases can be proven analogously.

- If $\psi_1 := z <_c x$ and $\psi_2 := y <_{c'} z$ then the new formula $\varphi = y <_{c+c'+1} x$ is added. Suppose $\mathfrak{A} \models \exists z(\psi_1 \wedge \psi_2)[x/a_1, y/a_2]$. Then there is an element $a \in \mathbb{Z}$ such that $\mathfrak{A} \models \psi_1[x/a_1, z/a]$ and $\mathfrak{A} \models \psi_2[y/a_2, z/a]$. Thus, by definition, we have $a_2 + c' < a$ and $a + c < a_1$ which is equivalent to $a_2 + c' < a$ and $a \leq a_1 - c - 1$. This implies $a_2 + c' < a_1 - c - 1$ which gives $a_2 + c' + c + 1 < a_1$. Thus $\mathfrak{A} \models \varphi[x/a_1, y/a_2]$.
  For the converse suppose $\mathfrak{A} \models \varphi[x/a_1, y/a_2]$. Then, by definition, $a_2 + c' + c + 1 < a_1$. Let $a := a_2 + c' + 1$. Clearly, $a_2 + c' < a$ and $a + c < a_1$ and therefore $\mathfrak{A} \models \psi_1[x/a_1, z/a]$ and $\mathfrak{A} \models \psi_2[y/a_2, z/a]$. Thus $\mathfrak{A} \models \exists z(\psi_1 \wedge \psi_2)[x/a_1, y/a_2]$.

- Suppose $\psi_1 := z <_c x$ and $\psi_2 := \neg z <_{c'} y$. We consider two sub-cases. For the first case suppose that $c' > c$. We illustrate this in the following figure.

  The figure illustrates the sets of numbers valid for $x$ and $y$ for a particular $z$. The values for $y$ are marked by the lines from the upper left to the lower right, whereas the values for $x$ are marked by the lines from the lower left to the upper right. As we can see the formula is satisfied if $y < x$ or if $y \geq x$ but $x + c' - c - 2 \geq y$. Thus $\exists z(\psi_1 \wedge \psi_2)$ is satisfied by all pairs $(a, b)$ so that $a + c' - c - 2 \geq b$, which is the set of pairs satisfying the formula $\neg x <_{c'-c-2} y$.
  For the second sub-case suppose that $c' \leq c$. Again we illustrate this by a figure.

  As we can see, the set of pairs $(a, b)$ such that $\mathfrak{A} \models \exists z(\psi_1 \wedge \psi_2)[x/a, y/b]$ is exactly the set $\{(a, b) \ : \ b + c - c' < a\}$. Thus the formula is equivalent to $y <_{c-c'} x$.

□

The next lemma proves that the elimination steps cover all pairs of formulae $\psi_1, \psi_2 \in \Psi$ such that $\exists z(\psi_1 \wedge \psi_2)$ restricts the set of tuples satisfying $\Psi$.

**Lemma 6.4.** *Suppose $\psi_1$ and $\psi_2$ are two literals containing $z$ such that no elimination step is defined for the pair $(\psi_1, \psi_2)$. Then for all $a_1, a_2 \in \mathbb{Z}$,*

$$\mathfrak{A} \models \exists z(\psi_1 \wedge \psi_2)[a_1, a_2].$$

*Proof.* To prove the lemma we consider all pairs $(\psi_1, \psi_2)$ of literals such that no elimination step is possible on them. Let $a_1, a_2$ be two arbitrary elements of $\mathbb{Z}$. Obviously if $\psi_1$ and $\psi_2$ are both of the same type, say for example, $\psi_1 := x_1 <_c z$ and $\psi_2 := x_2 <_{c'} z$, then we can always choose $z$ so that $a_1$ and $a_2$ satisfy both formulae. In the example we choose $z$ so that $a_1 + c < z$ and $a_2 + c' < z$.
The case where both formulae are of the types $z < a$ or $a < z$ is dealt with at the beginning of the algorithm. Now suppose that $\psi_1 := z <_c x_1$ and $\psi_2 := \neg x_2 <_{c'} z$. In this case we can choose $z$ so that it is both smaller than $a_1 - c$ and $a_2$.
What is left to show are the cases where one formula is of the type $a < z$ and the other is either of the type $x <_c z$ or $\neg z <_c x$, or one formula is of the type $z < a$ and the other is either $\neg x <_c z$ or $z <_c x$. If $\psi_1 := a < z$ and $\psi_2 := x_i <_c z$ then we choose $z$ so that it is larger than the maximum of $a$ and $a_i + c$. The other three cases can be proven analogously. All cases together give the proof of the lemma. □

The last lemma needed for the proof of Lemma 6.2 proves that $\exists z\Phi$ and $\Psi'$ are satisfied in $\mathfrak{A}$ by the same tuples $\overline{a}$, where $\Phi$ and $\Psi$ are defined as in Algorithm 6.1.

**Lemma 6.5.** *Let $\Psi$ be a conjunction of atomic or negated atomic formulae in the variables $\overline{x}, z$ and $\Psi' :=$ eliminate$(\Psi, z)$. Further, let $\Phi$ be as defined in Algorithm 6.1. Then $\Phi$ and $\Psi'$ are equivalent in $\mathfrak{A}$. In other words, for all $\overline{a}$,*

$$\mathfrak{A} \models \exists z\Phi[\overline{a}] \textit{ iff } \mathfrak{A} \models \Psi'[\overline{a}].$$

*Proof.* The forth direction is trivial because $\Phi$ contains all formulae occurring in $\Psi'$. For the opposite direction let $\Phi_z := \{\varphi_1, \dots, \varphi_n\}$ be defined as in Algorithm 6.1. Suppose $\mathfrak{A} \models \Psi'[\overline{a}]$. Let $Z$ and $Z_i$ for $1 \leq i \leq n$ be sets of elements defined as $Z := \{a : \mathfrak{A} \models (\bigwedge_{i=1}^{n} \varphi_i)[\overline{x}/\overline{a}, z/a]\}$ and $Z_i := \{a : \mathfrak{A} \models \varphi_i[\overline{x}/\overline{a}, z/a]\}$. We show that $Z$ is not empty. Obviously $Z = \bigcap_{i=1}^{n} Z_i$. Note that each $Z_i$ either consists of a single point, if $\varphi_i$ is $z = a$, a closed interval of elements, or an interval open on one side. Therefore $Z$ is empty if and only if there are two formulae $\varphi_r, \varphi_s \in \Phi_z$ such that $Z_r \cap Z_s = \varnothing$.
Now suppose that there exist two such formulae $\varphi_r, \varphi_s \in \Phi_z$ with $Z_r \cap Z_s = \varnothing$. Lemma 6.4 proves that if no elimination step is defined for $(\varphi_r, \varphi_s)$, then the intersection of $Z_r$ and $Z_s$ is not empty. Thus, as $\varphi_r$ and $\varphi_s$ contain $z$, there is an elimination step defined for the pair $(\varphi_r, \varphi_s)$ and the result $\psi$ of the elimination step is a sub-formula of $\Psi'$. By Lemma we get 6.3 that $\mathfrak{A} \models \exists z(\varphi_r \wedge \varphi_s)[\overline{a}]$ iff $\mathfrak{A} \models \psi[\overline{a}]$. Therefore if $Z_r \cap Z_s$ is empty then $\overline{a}$ does not satisfy $\psi$ which contradicts the assumption that $\mathfrak{A} \models \Psi'[\overline{a}]$. Thus no such pair $(\varphi_r, \varphi_s)$ exists. This proves the lemma. ☐

Now we are ready to prove Lemma 6.2.
*Proof of Lemma 6.2.* We consider the different cases of the algorithm *eliminate*. If there is no literal in $\Psi$ containing $z$, then $\exists z\Psi$ is equivalent to $\Psi$ which is the output of the algorithm.
To prove the second sub-case suppose there is a formula of the type $z = a$ in $\Psi$. Clearly, if there is also a formula $z = b$ with $b \neq a$, a formula $z < b$ with $b \leq a$, or a formula $b < z$ with $a \leq b$ in $\Psi$, then $\exists\Psi$ is not satisfiable and therefore equivalent to *false*. If no formula of these kinds occurs in $\Psi$, then $\exists z\Psi$ is equivalent to $\Psi' := \Psi[z/a]$.
The proof of the third sub-case is straightforward, because if $\Psi$ contains only one literal $\varphi$ containing $z$, then $\exists z\Psi$ is *false* if $\varphi$ is not satisfiable, for example $\varphi := z < z$, or it is equivalent to $\Psi' := \Psi \backslash \{\varphi\}$.
The proof of the fourth sub-case is slightly more complicated. The first step in this sub-case is to test whether the set $\Phi'$ of formulae $z < a$ or $a < z$ occurring in $\Psi$ is satisfiable. Clearly, if $\exists\Phi'$ is not satisfiable, then also $\exists z\Psi$ is not satisfiable and therefore equivalent to $\Psi' :=$ *false*. Now suppose that $\Phi'$ is satisfiable. The next step in the algorithm is to eliminate the formulae containing $z$. In each elimination step new formulae are added to $\Phi$. Clearly, as $\Phi$ is a conjunction of formulae, the set of tuples satisfying $\Phi$ cannot increase. Let $\varphi$ be a formula added to $\Phi$ by an elimination step on $\psi_1$ and $\psi_2$. By Lemma 6.3 we get that $\varphi$ is satisfied by all pairs $(a_1, a_2) \in \mathbb{Z} \times \mathbb{Z}$ satisfying $\exists z(\psi_1 \wedge \psi_2)$. Therefore $\exists z(\Psi \wedge \varphi)$ is equivalent in $\mathfrak{A}$ to $\exists z\Psi$ and by induction on the number of elimination steps we get $\exists z\Phi \equiv_{\mathfrak{A}} \exists z\Psi$. In the third step of this sub-case the set $\Psi'$ is obtained from $\Phi$

by removing the formulae in $\Phi$ containing $z$. We proved in Lemma 6.5 that $\exists z\Phi \equiv_{\mathfrak{A}} \Psi'$. Thus we have $\exists z\Phi \equiv_{\mathfrak{A}} \Psi'$ and $\exists z\Phi \equiv_{\mathfrak{A}} \exists z\Psi$ and therefore $\exists z\Psi \equiv_{\mathfrak{A}} \Psi'$. □

### 6.1.2 Handling negation

The next algorithm *negate* takes a quantifier-free formula as input and returns a quantifier-free formula which is equivalent to the negation of the input formula. Note that we cannot simply convert the input formula first to negation normal form and then back to disjunctive normal form again, because the size of the resulting formula could be exponential in the size of the input formula. Instead we define an algorithm based on complete atomic types to obtain LOGSPACE complexity.

**Definition 6.6.** Let $P$ and $C$ be non-empty sets of elements such that $c \geq -1$ for all $c \in C$. Let $\overline{a}$ be a tuple of elements.

- The signature $\tau_C$ *induced by* $C$ is defined as $\tau_C := \{<_c :\ c \in C\}$.
- The *complete atomic type of* $\overline{a}$ *over* $P$ *and* $C$ *with respect to* $\mathfrak{A}$, written as $atp^{\mathfrak{A}}_{P,C}(\overline{a})$, is defined as the set of all atomic or negated atomic formulae $\varphi(\overline{x})$ over the signature $\tau_C$ such that $\mathfrak{A} \models \varphi[\overline{a}]$.
- A maximally consistent set of atomic and negated atomic $\tau_C$-formulae is a *complete atomic type over* $P$ *and* $C$ *in the variables* $\overline{x}$, $atp^{\mathfrak{A}}_{P,C}(\overline{x})$, if it is the complete atomic type of a tuple $\overline{a}$ over $P$ and $C$.

Clearly, each tuple $\overline{a}$ realizes exactly one complete atomic type over $P$ and $C$ and elements realizing the same complete atomic type over $P$ and $C$ cannot be distinguished by quantifier-free formulae over the signature $\tau_C$ using only parameters from $P$.
We use this to define the algorithm *negate*. The idea is to successively generate all different complete atomic types over $P$ and $C$, where $P$ is the set of parameters occurring in the input formula $\psi$ and $C := \{c\ : <_c \text{ occurs in } \psi\}$, and test for each type whether a point realizing it satisfies $\psi$. The algorithm then outputs the disjunction of all types the test failed for.

To implement the algorithm sketched above we need some technical definitions and lemmas.

**Definition 6.7.** Let $\rho$ be a complete atomic type over $P$ and $C$ in the variables $\overline{x} := x_1, \ldots, x_k$. The *reduced type representation* $rtr(\rho)$ *of* $\rho$ is defined as follows.

*(i)* If there is a formula $\varphi := x_i = p$ in $\rho$, where $1 \leq i \leq k$ and $p \in P$, then $\varphi \in rtr(\rho)$.

*(ii)* For each $1 \leq i \leq k$, if there is a formula $\varphi := x_i < p$ and no formula $x_i < p'$ in $\rho$, where $p, p' \in P$ and $p' < p$, then $\varphi \in rtr(\rho)$.

*(iii)* For each $1 \leq i \leq k$, if there is a formula $\varphi := p < x_i$ and no formula $p' < x_i$ in $\rho$, where $p, p' \in P$ and $p < p'$, then $\varphi \in rtr(\rho)$.

*(iv)* For every $1 \leq i \neq j \leq k$, if there is a formula $\varphi := x_i <_c x_j$ and no formula $x_i <_{c'} x_j$ in $\rho$, where $c, c' \in C$ and $c < c'$, then $\varphi \in rtr(\rho)$.

*(v)* For every $1 \leq i \neq j \leq k$, if there is a formula $\varphi := \neg x_i <_c x_j$ and no formula $\neg x_i <_{c'} x_j$ in $\rho$, where $c, c' \in C$ and $c' < c$, then $\varphi \in rtr(\rho)$.

*(vi)* For every $1 \leq i, j \leq k$, if there is a formula $x_i = x_j$ or $\neg x_i = x_j$ in $\rho$, then it is also in $rtr(\rho)$.

*(vii)* $rtr(\rho)$ is the smallest set of formulae satisfying $(i) - (vi)$.

The set $k\text{-}rtr(P, C)$ is defined as $\{rtr(\rho) \ : \ \rho$ is a complete atomic type over $P$ and $C$ in the variables $x_1, \ldots, x_k\}$.

The next lemma proves that each complete atomic type $\rho$ is equivalent to its reduced type representation $rtr(\rho)$.

**Lemma 6.8.** *Let $\rho$ be a complete atomic type over $P$ and $C$ in the variables $x_1, \ldots, x_k$. Then for all $\overline{a} := a_1, \ldots, a_k \in \mathbb{Z}^k$,*

$$\mathfrak{A} \models \rho[\overline{a}] \quad \textit{iff} \quad \mathfrak{A} \models rtr(\rho)[\overline{a}].$$

*Proof.* The forth direction is straightforward, because every formula in $rtr(\rho)$ occurs also in $\rho$. To prove the back direction let $\overline{a}$ be a tuple such that $\mathfrak{A} \models rtr(\rho)[\overline{a}]$. We show that $\overline{a}$ satisfies every atomic and negated atomic formulae occurring in $\rho$. Let $\varphi \in \rho$ be atomic or negated atomic. We consider the different possible cases for $\varphi$.

*(i)* If $\varphi := x_i = p$, where $1 \leq i \leq k$ and $p \in P$, then $\varphi$ occurs also in $rtr(\rho)$ and is therefore satisfied in $\mathfrak{A}$ by $\overline{a}$.

*(ii)* If $\varphi := x_i < p$, where $1 \leq i \leq k$ and $p \in P$, then there is a formula $\varphi' := x_i < p'$ in $rtr(\rho)$ with $p' \leq p$. Clearly, $\varphi'$ implies $\varphi$ and therefore $\varphi$ is satisfied by $\overline{a}$.

*(iii)* The proof for $\varphi := p < x_i$ is analogous.

*(iv)* If $\varphi := x_i <_c x_j$, where $1 \leq i \neq j \leq k$, then there is a formula $\varphi' := x_i <_{c'} x_j$ in $rtr(\rho)$ with $c' \geq c$. Again, $\varphi'$ implies $\varphi$ and therefore $\overline{a}$ satisfies $\varphi$.

*(v)* The case where $\varphi := \neg x_i <_c x_j$, $1 \leq i \neq j \leq k$, can be treated analogously.

*(vi)* Formulae of the type $x_i = x_j$ or $\neg x_i = x_j$ occurring in $\rho$ also occur in $rtr(\rho)$ and are therefore satisfied by $\overline{a}$.

Thus every formula occurring in $\rho$ is satisfied by $\overline{a}$ and therefore $\mathfrak{A} \models \rho[\overline{a}]$. $\square$

Let $\rho$ be a complete atomic type and $rtr(\rho)$ be its reduced type representation. To find a tuple realizing $\rho$, we find a tuple satisfying the set of equalities and inequalities contained in $rtr(\rho)$. The (in)equalities occurring there can be rewritten according to the following equivalences.

$$\begin{array}{rcll}
p < x_i & \equiv & -x_i \leq -p-1 & \\
x_i < p & \equiv & x_i \leq p-1 & \\
x_i = p & \equiv & x_i \leq p \ \wedge -x_i \leq -p & \\
x_i = x_j & \equiv & x_i \leq x_j \wedge -x_i \leq -x_j & \\
x_i <_c x_j & \equiv & x_i + c + 1 \leq x_j & \equiv \ x_i - x_j \leq -c-1 \\
\neg x_i <_c x_j & \equiv & x_i + c \geq x_j & \equiv \ x_j - x_i \leq c
\end{array}$$

Thus we get that in order to find a tuple realizing $\rho$ we have to solve the following set of inequalities. For all $1 \leq i \neq j \leq k$ and $p \in P$,

$$\begin{array}{rl}
-x_i \leq -p-1, & \text{for every formula } p < x_i \in \rho, \\
x_1 \leq p-1, & \text{for every formula } x_i < p \in \rho, \\
\left.\begin{array}{r} x_i \leq p, \\ -x_i \leq -p, \end{array}\right\} & \text{for every formula } x_i = p \in \rho, \\
\left.\begin{array}{r} x_i \leq x_j, \\ -x_i \leq -x_j, \end{array}\right\} & \text{for every formula } x_i = x_j \in \rho, \\
x_i - x_j \leq -c-1, & \text{for every formula } x_i <_c x_j \in \rho, \\
-x_i + x_j \leq c, & \text{for every formula } \neg x_i <_c x_j \in \rho.
\end{array} \tag{6.1}$$

Let $m$ be the number of inequalities in (6.1). The set can be written in the form $A\overline{x} \leq \overline{b}$, where $A$ is a $(m \times k)$-matrix, all of whose coefficients are either 0, 1, or $-1$. We use the Fourier-Motzkin elimination method to solve the inequality $A\overline{x} \leq \overline{b}$ (see [Sch86]). In the method the variables are eliminated one after another. To do this we write the system of inequalities in the following form, where $\overline{x}' := (x_2, \dots, x_k)$ and $\overline{a}_i'$ is the $i$-th row of $A$ with the first entry removed.

$$\begin{array}{rll}
x_1 + (\overline{a}_i')^T\overline{x}' \leq b_i, & 1 \leq i \leq m_1 \\
-x_1 + (\overline{a}_j')^T\overline{x}' \leq b_j, & m_1 + 1 \leq j \leq m_2 \\
(\overline{a}_l')^T\overline{x}' \leq b_l, & m_2 + 1 \leq l \leq m_3
\end{array} \tag{6.2}$$

$m_1$ is the number of inequalities in which the coefficient of $x_1$ is 1, $m_2$ the number of inequalities in which the coefficient is $-1$, and $m_3$ the number of inequalities in which the coefficient is 0. The first two lines of (6.2) are equivalent to

$$\max_{m_1+1 \leq j \leq m_2} ((\overline{a}_j')^T\overline{x}' - b_j) \leq x_1 \leq \min_{1 \leq i \leq m_1} (b_i - (\overline{a}_i')^T\overline{x}'). \tag{6.3}$$

Therefore the variable $x_1$ can be eliminated. We get that (6.2) is equivalent to the following system of inequalities.

$$\begin{array}{rlll}
(\overline{a}_j')^T\overline{x}' - b_j & \leq b_i - (\overline{a}_i')^T\overline{x}', & & 1 \leq i \leq m_1, m_1 + 1 \leq j \leq m_2 \\
(\overline{a}_l')^T\overline{x}' & \leq b_l, & & m_2 + 1 \leq l \leq m_3
\end{array} \tag{6.4}$$

The system can be rewritten as follows.

$$\begin{array}{rll} (\overline{a}_i + \overline{a}_j)^T \overline{x}' & \leq b_i + b_j, & 1 \leq i \leq m_1, m_1 + 1 \leq j \leq m_2 \\ (\overline{a}_l)^T \overline{x}' & \leq b_l, & m_2 + 1 \leq l \leq m_3 \end{array} \tag{6.5}$$

In the last system of inequalities the variable $x_1$ is eliminated. A solution for $x_1$ can be derived from a solution for (6.5) by choosing a value for $x_1$ satisfying (6.3). If no such value exists then obviously (6.2) has no integer solution.

We use this to define an algorithm which outputs on a given reduced type representation a tuple satisfying it. Given a reduced type representation $rtr(\rho)$, we find a tuple realizing the type $\rho$ by successively eliminating the variables as described above until only one variable is left. We then choose a solution for this last variable and use it to compute solutions for the other variables.

Having defined the algorithm, we now consider its data complexity. In our setup the number $k$ of variables is fixed and the number of inequalities in the initial system (6.1) is bounded by $3k^2 + 3k$. We have to iterate the method only $k$ times and if $n_i$ is the number of inequalities after the $i$-th step then the number of inequalities after the $(i+1)$-th step is bounded by $n_i^2 + n_i$. Therefore the number of inequalities after the $k$ iterations is also bounded by a constant and independent of the input. To measure the complexity of the algorithm we must take into account the size of the coefficients which have to be stored on the working tape. We prove that the coefficients generated by the algorithm can be stored in logarithmic space. The proof is by induction on the number of iterations using the following lemma.

**Lemma 6.9.** *Any integer $a$ which can be calculated by a fixed number of additions of constants or parameters occurring in the input can be stored in space logarithmic to the size of the input.*

*Proof.* Suppose $a := p_1 + \ldots + p_r + c_1 + \ldots c_s$, where the $p_i$ are parameters from the input and the $c_i$ are constants. The parameters can be stored by a pointer to their occurrences on the input tape. Thus the parameters and constants used to calculate $a$ can be stored on the working tape in logarithmic space. Now $a$ can be stored as the tuple $(+, p_1, \ldots, p_r, c_1, \ldots, c_s)$.
Each time $a$ is needed by the Turing machine, the value of $a$ is calculated by adding the parameters and constants. Clearly, the addition of a fixed number of integers is in LOGSPACE. As the composition of LOGSPACE-algorithms is also in LOGSPACE, we get that $a$ can be stored and accessed in logarithmic space. □

We now show by induction on the number of elimination steps in the Fourier-Motzkin method that the coefficients generated by the algorithm are either constants, occur in the reduced type representation, or can be obtained from those by a fixed number of additions.
As explained above, all coefficients of a variable in the initial system (6.2) are either 0, 1, or $-1$. The $b_i$'s on the right side of the inequalities either occur in the input or they can be obtained by decreasing a parameter from the input by one. By Lemma 6.9 we get that these decreased parameters can be stored on the working tape using only space

logarithmic in the size of the input. In the induction step suppose that all coefficients in the previous steps can be stored in logarithmic space. Note that the coefficients in (6.5) can be obtained by addition of two coefficients from the previous elimination step. Thus, using Lemma 6.9 again, we get that these coefficients can also be stored in logarithmic space. As the total number of elimination steps is fixed we get the following corollary.

**Corollary 6.10.** *For every complete atomic type, a tuple realizing it can be computed in space logarithmic to the size of its reduced type representation.*

We are now ready to define the algorithm *negate*.

**Algorithm 6.11. negate**$(\psi)$

**Input:** *A quantifier-free formula* $\psi(x_1, \dots, x_k)$ *in dnf.*

**Output:** *A formula in dnf equivalent to* $\neg\psi$.

*The algorithm proceeds as follows.*

*Extract from the input the set* $P$ *of parameters occurring in* $\psi$.
*If* $P$ *is empty, add* $0$ *to it.*
*Extract the set* $C := \{c \in \mathbb{N} \ : \ c = 0 \text{ or } <_c \text{ occurs in } \psi\}$.
**for each** $\xi \in k\text{-}rtr(P, C)$ **do**
  *choose a tuple* $\overline{z}_\xi$ *satisfying* $\xi$.
  *test if* $\overline{z}_\xi$ *satisfies* $\psi$.
**od**
*output the disjunction of all* $\xi$ *such that* $\overline{z}_\xi$ *does not satisfy* $\psi$.

The next lemma proves the correctness of the previous algorithm.

**Lemma 6.12.** *Suppose* $\psi(x_1, \dots, x_k)$ *is a quantifier-free formula in disjunctive normal form and* $\varphi := negate(\psi)$. *Then* $\neg\psi \equiv \varphi$.

*Proof.* To prove the forth direction let $\overline{a}$ be a tuple satisfying $\neg\psi$. Let $\rho$ be the complete atomic type realized by $\overline{a}$. In the *for*-loop of the algorithm all possible reduced type representations in $k$ variables are enumerated. Thus also $rtr(\rho)$ is generated in an iteration step of the *for*-loop. The algorithm chooses a tuple $\overline{z}$ and tests whether it satisfies $\psi$. As $\overline{a}$ does not satisfy $\psi$, $\psi$ is quantifier-free, and by Lemma 6.8 $\overline{a}$ and $\overline{z}$ realize the same complete atomic type, also $\overline{z}$ does not satisfy $\psi$. Thus $rtr(\rho)$ is a disjunct in $\varphi$ and therefore $\varphi$ is satisfied by $\overline{a}$.
For the back direction let $\overline{a}$ be a tuple satisfying $\varphi$. Because $\varphi$ consists of a disjunction of reduced type representations, there is a complete atomic type $\rho$ such that $\xi := rtr(\rho)$ occurs as a disjunct in $\varphi$ and is satisfied by $\overline{a}$. As $rtr(\rho)$ occurs in the output, the point $\overline{z}_\xi$ chosen by the algorithm does not satisfy $\psi$. Thus, as $\overline{a}$ and $\overline{z}_\xi$ realize the same complete atomic type $\rho$ and $\psi$ is quantifier-free, $\overline{a}$ does also not satisfy $\psi$. This proves the lemma. □

Now we are ready to define the evaluation algorithm.

**Algorithm 6.13. gap-evaluate$_\psi(\mathfrak{B})$**

**Input:** *A gap-order constraint database* $\mathfrak{B} = (\mathfrak{A}, R_1^{\mathfrak{B}}, \ldots, R_k^{\mathfrak{B}})$.

**Output:** *A finite representation of* $\psi^{\mathfrak{B}}$.

*The algorithm is defined by induction on the structure of the query.*

- *If* $\psi$ *is atomic and does not contain a database relation symbol, then output* $\psi$.
- *If* $\psi$ *is of the form* $Ru_1, \ldots, u_{ar(R)}$, *where* $R$ *is a database relation symbol and the* $u_i$ *are either variables or parameters, then the output consists of the formula* $\varphi_R^{\mathfrak{B}}[x_1/u_1, \ldots, x_{ar(R)}/u_{ar(R)}]$, *where* $\varphi_R^{\mathfrak{B}}(\overline{x})$ *is the formula defining* $R^{\mathfrak{B}}$ *in* $\mathfrak{B}$.
- *Suppose* $\psi$ *is a conjunction of two formulae* $\psi_1$ *and* $\psi_2$. *First, the algorithm calculates* $\varphi_1 := \text{gap-evaluate}_{\psi_1}(\mathfrak{B})$ *and* $\varphi_2 := \text{gap-evaluate}_{\psi_2}(\mathfrak{B})$. *Then it returns as output the disjunction of each* $\varphi_1' \wedge \varphi_2'$ *such that* $\varphi_1'$ *is a disjunct in* $\varphi_1$ *and* $\varphi_2'$ *is a disjunct in* $\varphi_2$.
- *Suppose* $\psi$ *is a disjunction of two formulae* $\psi_1$ *and* $\psi_2$. *Then the output consists of* $\varphi_1 \vee \varphi_2$, *where* $\varphi_1 := \text{gap-evaluate}_{\psi_1}(\mathfrak{B})$ *and* $\varphi_2 :=\text{gap-evaluate}_{\psi_2}(\mathfrak{B})$.
- *Suppose* $\psi(x_1, \ldots, x_k) := \neg\psi'(\overline{x})$. *First* gap-evaluate$_{\psi'}(\mathfrak{B})$ *is called. The output is then obtained by applying the algorithm* negate *to the resulting formula.*
- *Suppose* $\psi(x_1, \ldots, x_k) := \exists z \psi'(\overline{x}, z)$. *First* $\Phi :=\text{gap-evaluate}_{\psi'}(\mathfrak{B})$ *is calculated. Then the disjunction of* $\varphi' := \text{eliminate}(\varphi, z)$ *for every disjunct* $\varphi$ *in* $\Phi$ *is returned as output.*

Having defined the evaluation algorithm we now prove its correctness.

**Theorem 6.14.** *Suppose* $\mathfrak{B}$ *is a* $\sigma$*-database over* $\mathfrak{A}$ *and* $\psi$ *a first-order query. Then* gap-evaluate$_\psi(\mathfrak{B})$ *is a finite representation of* $\psi^{\mathfrak{B}}$.

*Proof.* The theorem can be proven by induction on the structure of the query. The proof for the negation and existential quantification cases follow from the Lemmas 6.12 and 6.2 respectively. The atomic and boolean cases are straightforward. □

The previous theorem proves the correctness of the evaluation algorithm. We investigate its complexity in the next section.

## 6.2 Complexity of first-order queries

The aim of this section is to prove a LOGSPACE upper bound for the evaluation algorithm of the previous section. We do this by proving that the two sub-algorithms *negate* and *eliminate* are LOGSPACE-algorithms. The LOGSPACE-bound for *gap-evaluate* then follows easily.

**Lemma 6.15.** *The algorithm* eliminate *has a* LOGSPACE *upper complexity bound.*

*Proof.* To prove the lemma we have to consider the four sub-cases of the algorithm. The proof of the LOGSPACE-bound of the first three sub-cases is straightforward. We now prove the LOGSPACE-bound of the fourth sub-case. The input of the algorithm consists of a conjunction of atomic or negated atomic formulae. For each pair $(\varphi_1, \varphi_2)$ of these formulae the algorithm checks whether an elimination step can be applied to them. If so, then the result of the elimination step is calculated. Clearly, each elimination step can be done in LOGSPACE, because only the addition of a fixed number of integers is involved. As the elimination steps are only defined for sub-formulae of the input, the output of one step does not have to be considered for further elimination steps. Therefore it does not have to be stored on the working tape but can directly be written on the output tape. We get that all elimination steps together can be done in LOGSPACE. After all steps have been done, the algorithm copies all input formulae which do not contain $z$ to the output. This is clearly in LOGSPACE. Thus *eliminate* is a LOGSPACE-algorithm. □

The next lemma proves the LOGSPACE upper bound for the algorithm *negate*.

**Lemma 6.16.** *The algorithm* negate$_\psi$ *has a* LOGSPACE *upper complexity bound.*

*Proof.* The algorithm starts by extracting the sets $P$ and $C$. These sets do not have to be stored on a Turing tape but can be looked up in the input each time an element of a set gets used. Clearly, this is in LOGSPACE. In the following *for*-loop, the algorithm generates one after another all reduced type representations for complete atomic types $\rho$ in the variables $x_1, \ldots, x_k$. Each reduced type representation consists of at most $3k^2 + 3k$ atoms or negated atoms. All gap-orders occurring in a reduced type representation occur also on the input tape and can therefore be stored by pointers to the input. Thus a reduced type representation can be stored in space logarithmic to the input size.
Once a reduced type information is generated, a tuple $\overline{z}$ satisfying it is computed. By Corollary 6.10 this is in LOGSPACE. Finally it is checked whether $\overline{z}$ satisfies $\psi$. Clearly, this can also be done in LOGSPACE. As the space used in one iteration step of the *for*-loop can be reused in the next iteration step, the space needed by *negate* is logarithmic in the size of the input. This proves the lemma. □

We can now prove the main result of this chapter.

**Theorem 6.17.** *First-order queries on gap-order constraint databases can be evaluated in* LOGSPACE.

*Proof.* Suppose $\psi$ is a first-order query. We show that the algorithm *gap-evaluate*$_\psi$ can be evaluated in logarithmic space by induction on the structure of the query. The atomic cases are simple. If $\psi$ is atomic and does not contain a database relation symbol, then the output consists of $\psi$ itself. Otherwise if $\psi$ contains a database relation symbol, then, essentially, the algorithm copies the representation of the relation in the database as output. Both can be done in LOGSPACE.

Now suppose $\psi$ consists of a conjunction of two formulae $\psi_1$ and $\psi_2$. First, *gap-evaluate* is called recursively on $\psi_1$ and $\psi_2$ resulting in two formulae $\varphi_1$ and $\varphi_2$. By induction, the recursive calls to *gap-evaluate* are in LOGSPACE. Then the conjunction of each pair of disjuncts from $\varphi_1$ and $\varphi_2$ is generated and the disjunction of these is returned as output. Clearly, this can also be done in LOGSPACE. The case where $\psi := \psi_1 \vee \psi_2$ can be treated analogously.

Lemma 6.16 and 6.15 prove the negation and quantification cases. As the composition of LOGSPACE algorithms is also in LOGSPACE, the complete algorithm works in LOGSPACE. This proves the theorem. □

# Chapter 7

# Conclusion

In this work we investigated the complexity of query languages for constraint databases. The main interest was in the data complexity of query languages based on fixed-point extensions of first-order logic. In most cases arithmetical structures served as context structures, either dense structures with the real numbers as universe, or discrete structures over the naturals or integers.

## 7.1 Summary of results

We began our investigations in Chapter 3 with methods to find complexity bounds for query languages. These bounds have been derived from known complexity bounds for problems related to the decision problem for the theory of the context structure. As an application of these methods we proved that for each level of the polynomial time hierarchy there are first-order queries on databases defined over Presburger arithmetic whose data complexity is complete for this level.

In the remaining three chapters we considered the data complexity of query languages for particular context structures. Chapter 4 gave an overview of complexity results for various context structures whereas in Chapter 5 and Chapter 6 we considered in detail two special context structures, namely dense and discrete linear orders.

In the main result of this work we presented a general method to prove complexity bounds for query languages over dense order databases. The idea was to code the finitely represented database as a finite database and then use the evaluation algorithms available for the query language on finite databases. It turned out that this encoding can be defined by first-order formulae using only the order predicate and some very limited kind of arithmetic. It can therefore be done with very low data complexity. This method enabled us to evaluate queries for various query languages within the same complexity classes as for finite databases.

This method also works for databases defined by inequality constraints over a countable infinite set. Therefore queries defined over inequality constraints can be computed within complexity bounds as for finite databases. We also proved that the various fixed-point logics considered here are too weak to express all LOGSPACE-computable queries. The proof was an application of Ehrenfeucht-Fraïssé games, one of the few methods from

| | inequality | $(\mathbb{R}, <)$ | $(\mathbb{R}, <, +)$ | $(\mathbb{R}, <, +, *)$ |
|---|---|---|---|---|
| FO | $\subset$ AC$_0$ | $\subseteq$ AC$_0$ | ? | NC |
| *FO+DTC* | $\subset$ LOGSPACE | = LOGSPACE | undecidable | undecidable |
| *FO+TC* | $\subset$ NLOGSPACE | = NLOGSPACE | undecidable | undecidable |
| *FO+LFP* | $\subset$ PTIME | = PTIME | undecidable | undecidable |
| *FO+PFP* | $\subset$ PSPACE | = PSPACE | undecidable | undecidable |

Table 7.1: Data complexity bounds for query languages on dense structures.

| | $(\mathbb{N}, <_c)$ | $(\mathbb{N}, <, +)$ | $(\mathbb{N}, <, +, \cdot)$ |
|---|---|---|---|
| FO | $\subseteq$ LOGSPACE | = PH | undecidable |
| *FO+DTC* | ex. pos. *FO+LFP* closed | undecidable | undecidable |
| *FO+TC* | | undecidable | undecidable |
| *FO+LFP* | undecidable | undecidable | undecidable |
| *FO+PFP* | undecidable | undecidable | undecidable |

Table 7.2: Data complexity bounds for query languages on discrete structures.

finite or classical model theory that still work for finitely representable databases.

We saw in Chapter 4 that adding arithmetical functions to a dense order as context structure still yields efficient query languages. In particular it has been shown that first-order queries have NC data complexity over real closed fields. The situation changes drastically if discrete instead of dense orders are under consideration. Although the data complexity of first-order queries on gap-order constraints was in Chapter 6 shown to be LOGSPACE, the complexity of more expressive logics increases enormously. Least fixed-point logic is Turing-complete and the only thing known about positive DATALOG, which is equivalent to positive existential *FO+LFP*, is that it can be evaluated in closed form. But nothing more is known about its complexity.

Also adding arithmetical functions to discrete orders yields very complex query languages. It was shown that first-order logic captures the polynomial time hierarchy on the class of databases defined over the Presburger arithmetic. First-order queries on the naturals with order, addition, and multiplication are even undecidable.

Thus it seems that using dense structures as context structures is more promising than the usage of discrete structures. Unfortunately this is only true for first-order queries. If not first-order but, for example, *FO+TC*-queries are considered, we end up in undecidable query languages if the context structure includes addition.

Table 7.1 and 7.2 give an overview of the complexity results presented in this work.

## 7.2 Open problems

When studying Table 7.1 one encounters undecidability results for many interesting query languages and context structures. The reason for this is that the combination of recursion and addition make the natural numbers with their addition and multiplication definable in the reals. One question is to find query languages that allow a useful form of recursion and addition but are still decidable and computable in closed form. Another open entry in the table is the complexity of first-order queries on $(\mathbb{R}, <, +)$.

Generally, there is a lack of methods to prove complexity bounds or (un)definability results for finitely representable structures. One of the most powerful ways to find upper complexity bounds seems to be to code a finitely representable database as a finite set of objects, evaluate the query there and transform the result back. Examples of this can be found in the proofs of the LOGSPACE-complexity for inequality constraints over countable infinite sets and in the proof for dense order databases given in Chapter 5. Recently Benedict and Libkin [BL98] as well as Vandeurzen, Gyssens, and Van Gucht [VGG98] considered this idea in a broader sense.

# Bibliography


[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[BL98] M. Benedict and L.Libkin. Safe constraint queries. In *Proceedings of the 17th ACM Symposium on Priciples of Database Systems*, pages 99–108, 1998.

[BoKR86] M. Ben-or, D. Kozen, and J. Reif. The complexity of elementary algebra and geometry. *Journal of Computer System Science*, 32:251–264, 1986.

[BST98] O. V. Belegradek, A. P. Stolboushkin, and M. A. Taitslin. Extended order-generic queries. *Annals of Pure and Applied Logic*, 1998. To appear.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, 1970.

[EF95] H.D. Ebbinghaus and J. Flum. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer Verlag, 1995.

[EFT94] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Springer, 1994.

[Grä88] E. Grädel. Subclasses of Presburger arithmetic and the polynomial-time hierarchy. *Theoretical Computer Science*, 56:289–301, 1988.

[GS94] S. Grumbach and J. Su. Finitely representable databases. In *Proceedings of the 13th ACM Symposium on Priciples of Database Systems*, pages 289–300, 1994.

[GS97a] S. Grumbach and J. Su. Finitely representable databases. *Journal of Computer and System Sciences*, 55(2):273–298, 1997.

[GS97b] S. Grumbach and J. Su. Queries with arithmetical constraints. *Theoretical Computer Science*, 173(1):151–181, 1997.

[GST95] S. Grumbach, J. Su, and C. Tollu. Linear constraint query languages: Expressive power and complexity. *Lecture Notes in Computer Science*, 960:426–446, 1995.

[Hod97] W. Hodges. *A shorter model theory.* Cambridge University Press, 1997.

[HU79] J. Hopcraft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Imm98] N. Immerman. *Descriptive complexity.* Springer Graduate Texts in Computer Science, 1998.

[KG94] P. C. Kanellakis and D. Q. Goldin. Constraint programming and database query languages. *Lecture Notes in Computer Science*, 789:96–120, 1994.

[KKR90] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 299–313, April 1990.

[KKR95] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *Journal of Computer and Systems Sciences*, 51(1)(1):26–52, 1995. (an extended abstract appeared in Proc. PODS'90).

[Len83] H. Lenstra. Integer programming with a fixed number of variables. *Mathmatics of Operation Research*, 8:538–548, 1983.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Pre27] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101, Warsaw, 1927.

[Rev90] P.Z. Revesz. A closed form for datalog queries with integer order. In *Proc. 3rd International Conference on Database Theory*, pages 187–201, 1990.

[Rev93] P.Z. Revesz. A closed form evaluation for datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116:117–149, 1993.

[Sca84] B. Scarpellini. Complexity of subcases of Presburger arithmetic. *Transactions of the AMS*, 284:203–218, 1984.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[Sch97] U. Schöning. Complexity of Presburger arithmetic with fixed quantifier dimension. *Theory of Computing Systems*, 30:423–428, 1997.

[Var82] M. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.

[VGG98] L. Vandeurzen, M. Gyssens, and D. Van Gucht. An expressive language for linear spatial database queries. In *Proceedings of the 17th ACM Symposium on Priciples of Database Systems*, pages 109–118, 1998.