FOUNDATIONS OF TEMPORAL QUERY LANGUAGES

by

DAVID TOMAN

Mgr., Masaryk University, Brno, Czech Republic, 1992

A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1995

Approved by:

Dr. Jan Chomicki
Major Professor

Abstract

Temporal Databases are repositories of information dependent on time. The major difference from standard, e.g., relational database systems, is the need of storing possibly infinite objects, e.g., time spans.

In recent years, there have been numerous proposals that introduce time into standard relational systems. Unfortunately, most of the attempts have been based on ad-hoc extensions of existing database systems and query languages, e.g., TQUEL and TSQL. Such extensions often create many problems, when precise semantics needs to be developed, if one exists at all. In a recent survey by J. Chomicki, a clean way of defining temporal databases based on logic was proposed. This methodology views temporal databases as multi-sorted, finitely representable first-order structures. Query languages then became formulas in suitable logics over the vocabulary of such structures. This method has been quite successful, as most of the existing proposals are subsumed by this approach with only minor adjustments for a few special cases.

In this dissertation we try to address several issues connected with this approach and the development of Temporal Databases:

- In most of the current implementations of Temporal Databases a seemingly natural restriction on the structure of relations is imposed—all the relations stored in the database have exactly one or a bounded number of attributes that represent time, e.g., TSQL or TQUEL. Such a restriction naturally corresponds to the single dimensional modeling of time. However, it also raises an obvious question about the generality of such a solution. We show that even in the first-order case, imposing such discipline on a query language affects the overall expressive power and rules out many interesting queries over the database.

- How to store and handle infinite objects, e.g., time spans. Here we use the idea of generalized databases and encode some of the infinite objects using formulas of appropriate logic. We generalize the common requirements on such encodings that make them suitable for query evaluation. We also propose a new class of infinite objects (periodic sets) together with efficient procedures that allow deductive queries to be evaluated over such objects.

- How to efficiently evaluate queries over such encodings. We propose a new query evaluation procedure based on a combination of the resolution method proposed for logic programs with the fixed point computation used for deductive query evaluation. Both theoretical complexity bounds and experimental measurements show the performance gains achieved by using this method.

Contents

List of Figures

Acknowledgement

First of all I would like to thank Jan Chomicki, my major professor, for his support and encouragement during the whole three and half years of my studies. Without his support I would not been able to come to Kansas and work on my PhD. During my studies, besides his invaluable technical insights, he also taught me the other very important part in doing research: how to write good research papers.

I would like to thank prof. Damian Niwinski for teaching me the basics of the finite model theory and his continuous interest in my work on temporal logic, which culminated in a joint paper and a chapter of my dissertation, and prof. Peter Revesz for valuable discussions, help, and encouragement in my work in the area of constraint databases.

My thanks also go to all the other people at Kansas State, who made my stay enjoyable both professionally and personally: Olivier Danvy, Rod Howell, Masaaki Mizuno, Dave Schmidt, Allen Stoughton, George Strecker, Brian Howard, Sergey Kotov, Anindya Banerjee, Christian Haack, Adrian Fiech, Angie Taylor (who helped me with all the necessary paperwork, I would otherwise certainly be lost in), and all the other nice people I met during my studies.

I would also like to thank Jason Dale, Kristi Hankley, Cindy Liu, and Hollie Niblett for proofreading parts of my dissertation and helping me to improve my (approximation to) English.

Last but not least, I would like to thank all the people back in Czechoslovakia (nowadays Czech Republic and Slovakia), especially to my parents for their patience and to my former professors for teaching me lots of math and computer science theory, both of which helped me tremendously in my studies and research here.

Chapter 1

Introduction

Representing, storing, and manipulating time-dependent data have been the focus of active research for the last several years. The approaches used evolved from ad-hoc extensions of the existing database systems to a more sophisticated and systematic studies of the fundamental properties of Temporal Databases as repositories of time-dependent data. The main difference from the standard database theory is that the temporal databases are often required to store encodings of infinite objects—time spans (possibly infinite sets of time instants). Together with the development of appropriate storage techniques, new languages for querying and manipulating such data and their encodings have been developed.

1.1 Temporal Databases

A systematic approach for studying temporal databases was proposed in [20]. This approach is based on viewing the temporal databases as precise mathematical objects—first-order structures, whose properties can be rigorously studied using the tools of model theory. However, in contrast to the classical model theory, in the area of databases the main focus lies in the finite or finitely representable structures.

This approach also raises important questions that are often not answered in many implementations of temporal databases. However, a precise specification of what the temporal databases actually are needs these questions to be answered first (we answer only those questions that are relevant to further technical development in this dissertation; a more complete discussion can be found in [20]):

Domains for the individual sorts of data to be stored in the database. A precise description of a database as a mathematical object requires a description of the *data* we plan to store in the database, e.g., strings, numbers, etc. This can be conveniently achieved using a *first-order structure* as a domain for the particular data type of storable objects. Such structure also provides the *built-in* relations defined on the data objects, e.g., equality, (linear) order, arithmetic, etc. For example, the classical theory of relational databases in its simplest incarnation uses a *set of uninterpreted constants* equipped only with *equality* and names for the elements, as the underlying domain of the objects, storable in the database. Thus, the only legal queries are combinations of questions about equality between components of the individual tuples, stored in the database, and constants.

In the case of temporal databases, we need at least one more domain of data—the domain that represents *time instants*. For representing time the equality relation and constants are no longer sufficient. Besides modeling the time instants themselves, we also need to model the progression of time—the temporal ontology. This goal can be conveniently achieved using an ordering on the time instants. However, there are many choices of the particular ordering, depending on the structure of time, we try to model, e.g.:

1. Are we considering bounded or unbounded time, i.e., is there a beginning of time, an end?
2. Do we use a discrete or a dense model of time, i.e., does every time instant have a successor, or is there an instant between every two comparable instants?
3. Is the time ordered linearly (a single time line)? Is it branching (alternative time lines)? Branching to the future (alternative futures)?

In this dissertation we consider only linearly ordered unbounded time domains. Thus, we can choose one of the familiar number structures like

$(Z, <)$ linearly ordered integers for the discrete time[1] and

$(Q, <)$ linearly ordered rationals for the dense time domains

The advantage of choosing a familiar number domain for modeling time over other more general domains lies in the well-studied properties of the number structures. Note that the time domains are inherently infinite and not necessarily countably infinite. As a side-effect we do not have a constant symbol for every element of the domain, although we can have constants for a few *important* time points like 0. In later chapters we introduce richer structures for modeling time, e.g., structures, equipped with gap-order relations or congruence relations.

The signature of the user-defined Relations. The choice of the underlying first-order structures fixes the objects that can be handled by the database. To capture the *relationships* between the objects we need a set of *user-defined* relations, often called tables in the world of relational databases. These relations, or their encodings in the computer storage, are the actual *data* stored in the database itself as they do *not* have a predetermined meaning like order or equality.

The temporal databases are designed to store relationships between time instants and data objects—the elements of the data domain. Thus, the user-defined relations need to be at least two-sorted[2]. There are two approaches to enforce the sort (type) discipline:

[1] We also introduce few other *built-in* relations on Z in later chapters.

[2] We consider temporal databases as two-sorted, first-order structures, where one sort represents the data objects and the other one models the time instants. An extension of this framework to more sorts is straightforward (cf. Section 4).

1. The most common approach, and the one chosen by virtually all implementations of temporal database management system (DBMS), is to have the user-defined relations of varying arity in the data sort but with exactly one distinguished attribute of the temporal sort. There is a clear underlying semantic interpretation of such an arrangement: to express the fact that $(a_1, \ldots, a_k)$ are related by R at time t we simply say $(t, a_1, \ldots, a_k) \in R$, where the first element of the tuple is the distinguished temporal attribute. This choice can be viewed as a *uniform* extension of the standard database schema by adding one distinguished attribute of the time sort to each relation.
2. The other choice is to have varying arity of relations in both sorts—an unrestricted database schema. This choice does not have a natural interpretation like (1). However, we gain greater flexibility using this approach. In Chapter 3 we also see that this choice is actually preferable to the first one, as it facilitates desirable properties of the query languages.

The finite encodings of Temporal Databases. So far we defined the temporal databases as two-sorted, first-order structures, equipped with several *built-in* relations on the sorts, e.g., equality or order, and *user-defined* relations conforming to a particular sort discipline. However, we did not (yet) impose any restrictions on the size of the user-defined relations in the structures[3]. This is obviously a problem from the implementation point of view—current computers can store only finite amounts of information. Thus, we cannot store all possible instances of the structures defined in the previous paragraph, provided we have chosen non-trivial domains. There are two major approaches to this problem:

1. we allow the storage of only those structures, where the user defined relations are finite—this approach is taken by virtually all relational systems, or
2. we store *finite encodings* of a subclass of the structures[4].

Appropriate query languages. We have described a way of precise description of a temporal database itself—we view it as a two sorted structure with the appropriate sorts extended by a finite set of user-defined relations that can be finitely encoded. To make the whole effort worthwhile, we also need to define tools that allow asking questions about the contents of the temporal database, i.e., about the data stored in the user-defined relations.

Here we can capitalize on our abstract definition of the temporal database: the queries are simply functions on the class of all temporal databases thought of as first-order structures. We can use a language of logic over the same vocabulary as the vocabulary of the structures representing the temporal database as a query language. The queries are defined by formulas and the *answer to a query* is simply the relation consisting of the tuples (valuations) that make the particular formula true in the temporal database

[3]The built-in relations are usually infinite, but we do not need to worry about them, as their interpretation is fixed and thus they do not need to be stored in the database itself, provided we have an algorithm for membership testing.

[4]Clearly, we cannot finitely encode all the structures for cardinality reasons.

as a first-order structure. Thus, the answers to queries can be viewed as temporal databases themselves .

The particular choice of appropriate logic depends on the required expressive power and/or required efficiency. The logics we explore in this dissertation are:

1. the single-sorted first-order temporal logic (cf. chapter 3),
2. the two-sorted first-order logic (cf. Chapters 2, 3, and 4), and
3. Datalog (universal Horn clauses with fixpoint semantics; Chapters 4, 5, 6, and 7).

We have informally described a fragment of the framework of the temporal database theory we use in this dissertation. In addition, the chosen domains for representing data objects and time instants, the shapes of the user-defined relations, the finite encodings of them, and the query languages have to fit together; more formally, at least the following properties have to be satisfied by our choices:

Faithfulness of Encoding. The advantage of choosing logic as a query language for temporal databases is its clear semantics and easy way of declarative formulation queries[5]. The drawback of this choice is that the semantics of the queries is defined on the abstract structures, rather than on their finite encodings. We have to make sure that the evaluation of a query as a computer program derived from a given formula preserves the meaning of the query evaluated as formula of logic with respect to the temporal database.

Closure. We view queries as maps on the class of temporal databases. Thus, the second natural requirement is that the answer to any query is *representable* as a database consisting of the relation holding the answer, or an appropriate finite encoding of the answer. In particular, this implies that queries (as algorithms) have to produce the answer in finite time, and thus only finite encodings of relations can be valid answers.

Note also that when dealing with large amounts of data, the implementation of the query evaluation algorithm may need to store not only the final result of the query, but also several intermediate results produced during the query evaluation, that represent answers for subformulas of the original query. This is possible only when the query language is *subquery closed*, i.e., all subqueries are queries themselves.

Genericity. In [18] generic queries were defined as queries invariant with respect to a permutation of constants. This definition is intended to prevent those queries, where the answer depends on a particular way of storing the constants, e.g., representation of numbers or strings in the computer storage. In our framework, we generalize this notion as follows: We require that if two different encodings encode the same temporal database, then the answers to a generic query, evaluated with respect to both the encodings, encode the same answer. Note also that in the case, when our encoding is simply embedding, as in relational systems, the generalized notion of genericity is equivalent to the original notion of [18].

[5] Provided, the user is sufficiently mathematically sophisticated.

1.2 Overview of the Contributions

The goal of this dissertation is to address several issues connected with the design and implementation of temporal databases and databases that allow one to represent and store encodings of infinite objects.

1.2.1 Expressive Power of First-order Query Languages

The first part of the dissertation addresses the choice of the allowed types of the user-defined relations. We show that restricting the number of temporal attributes to one or any fixed number, common to virtually all the current implementations of temporal databases [15, 83, 84], has a major drawback: there is no closed language that can express all first-order queries. This result also resolves (negatively) the long-open problem of existence of an expressively complete set of connectives for first-order temporal logic over linear orders. Moreover, it also means that there is no first-order complete relational algebra (cf. chapter 3 or [97]) that properly implements all first-order queries over the restricted schema.

1.2.2 Querying Finite Encodings of Infinite Objects

In the light of the first result, we concentrate on schema-unrestricted finite representations of infinite relations for representing the temporal contents of the database. We use quantifier-free formulas of logic (constraints) as a basis for finite encodings of potentially infinite user-defined relations. The contributions in this part of the dissertation are as follows:

1. We define abstract properties that need to be satisfied by every finite representation of infinite relations to guarantee correct evaluation of queries over such representation. We also show that the composition of disjoint domains causes no additional problems.

2. We introduce a new class of finitely representable structures over integers that allows manipulation of infinite periodic sets, together with all the operations needed for deductive query evaluation. We study termination and complexity of query evaluation for this class of temporal databases [96].

3. We develop an efficient method for evaluating deductive queries over the finite representations. We support the suitability of our query evaluation algorithm by both theoretical complexity results and benchmarks against an experimental implementation of the proposed method [98].

1.3 Organization of the Dissertation

The dissertation is organized as follows: Chapter 2 introduces precise definitions of the concepts, some of which have been informally introduced in this chapter, needed for further technical development. Chapter 3 shows the first result—the separation of two-sorted first-order logic from first-order temporal logic with arbitrary finite set of connectives over dense linear orders. This result implies that the enforcement of the sort restriction on the database schema, i.e., on the shapes of the user-defined relations, and the closure of a first-order query language, with respect to such schema, rules out large class of first-order queries over such

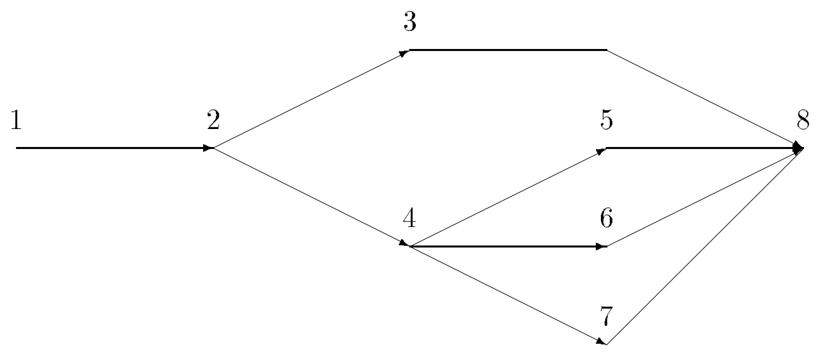


Figure 1.1: Dependencies of chapters in the dissertation.

a temporal database. We propose to use the unrestricted database schema to avoid this penalty. Chapter 4 presents a general methodology for finite representation of infinite, e.g., temporal relations. Chapter 5 introduces a new class of finitely representable structures over integers together with all operations on the finite representation needed for query evaluation. It also studies the complexity of query evaluation. Chapter 6 presents a list of other known finitely representable classes and briefly discusses their properties. Chapter 7 introduces an efficient query evaluation strategy and studies its properties. A thorough comparison with existing methods is carried out both theoretically and experimentally. Chapter 8 concludes the dissertation with possibilities of further directions of research.

Every chapter starts with a brief overview of the concepts and goals, and is concluded with discussion of relevant work and a brief bibliography. Figure 1.1 shows the logical dependencies between the individual chapters of this dissertation.

Chapter 2

Background

In this chapter we introduce the necessary theoretical foundations of temporal databases as an extension of the standard relational database theory. We introduce only the concepts needed for further technical development.

2.1 Databases as First-order Structures

To define a database, or rather a class of all possible databases we want to consider as a mathematical object, we need to answer following questions:

1. what are the values stored in the database?
2. how are the values stored and what does storing the values in a particular way mean?
3. what can we do with the stored values?

The first question can be answered by the following definitions that can be used for most of the standard relational systems; in the case of temporal databases we use an extension of this approach:

Definition 2.1 (Data Domain) *Let D be a first order structure*

$$\langle |D|; =, c_1, c_2, \ldots \rangle$$

with countable carrier $|D|$, equipped with equality and a constant symbol c_i for every element of the carrier $|D|$. We call D a Data Domain *or a* domain of uninterpreted constants.

Note that the constant symbols can be equally well represented by unary relations containing exactly one element.

Example 2.2 Typical examples of data domains are:

1. the natural numbers $\{1, 2, 3, 4, \ldots\}$. This is probably the most common choice, as all the objects stored in or manipulated by the database system are ultimately encoded as binary numbers in the computer storage.

2. the set of strings over a finite alphabet. This is another common choice. In this way we can conveniently represent the usual objects stored in the database, like names, addresses, etc. Note that such strings can be thought of as numbers with digits from the given finite alphabet of the strings.

However, the only relation defined on these sets is equality—the diagonal relation.

The choice of the data domain in this setting is not crucial, as all countably infinite structures with equality are equivalent up to isomorphism.

The answer to the second question is more complicated: it is not sufficient to store the elements of the data domain in the database by themselves, as we are trying to capture *relationships* between the values rather than the values themselves[1]. There are several different approaches developed to address this problem (often called database models). In this dissertation we use the *relational* approach: the relationships between the individual values are captured as tuples, and stored in *user-defined relations*—sets of such tuples. The contents, or the encoding of the contents of these relations, is then stored in the database, i.e., written into the permanent storage.

An important decision is what *shape* (relational type) the user-defined relations can have, i.e., what are the allowed arities and, in the multi-sorted case, what are the types of the components of the individual tuples. Consider the following definitions (again, for the relational case):

Definition 2.3 (Database Schema) *Let $\mathcal{R}$ be a countable set of predicate symbols of all finite arities. A* database schema *is a finite subset $\rho \subseteq \mathcal{R}$.*

$\mathcal{R}$ is the set of all *allowed* user-defined relation names and relational types—the domains of allowed values for the individual attributes of tuples stored in the relation, named by a particular predicate symbol. The choice of $\mathcal{R}$ determines what shapes the individual user-defined relations can have. In the case of relational databases, we can assume that $\mathcal{R}$ contains "all" possible predicate symbols of all arities—so we can always pick the ones we want in the database schema.

Definition 2.4 (Instance of Database Schema) *Let $\rho = (r_1, \ldots, r_k)$ be a database schema and D a data domain. An* instance of the database schema ρ in D *is a finite set of relations $R = (R_1, \ldots, R_k)$ such that $R_i \subseteq |D|^{\mathrm{arity}(r_i)}$.*

A finite *instance of the database schema ρ in D is an instance where $|R_i|$ is finite for all $0 < i \leq k$.*

Definition 2.5 (Relational Database) *Let D be a data domain, ρ a database schema, and R an instance of ρ over D. Then the structure D extended with R*

$$D \cup \{R_1, \ldots, R_k\}$$

is a Relational Database *over the domain D and the schema ρ.*

We define a class of all relational databases over D *as the class of all extensions of D by an instance of a database schema ρ for any finite $\rho \subseteq \mathcal{R}$.*

[1] That is why we often call the elements of the data domain *uninterpreted constants*.

A class of finite relational databases over D *is the class of all extensions of* D *by a finite instance of a database schema* ρ *for any finite* $\rho \subseteq \mathcal{R}$*.*

The difference from the standard relational theory is, that we *fix* the underlying domains in advance—this choice eliminates problems with the closed world vs. open world assumptions, the need for active domains, etc[2].

Now we can also see how the choice of $\mathcal{R}$ influences the structure of the databases, i.e., the shapes of the user-defined relations:

Example 2.6 Let $\mathcal{R}$ be a set of all unary and binary predicate symbols. Then all relations in our database have to be unary or binary.

Over the class of all relational databases we can define query languages based on first-order logic:

Definition 2.7 (Relational Calculus) *Let* $r_i \in \mathcal{R}$ *be predicate symbols.*

$$M ::= r_i(x_{i_1}, \ldots, x_{i_{\mathrm{arity}(r_i)}}) \mid x_i = x_j \mid M \wedge M \mid \neg M \mid \exists x_i.M$$

We give a natural Tarskian semantics to the formulas of the relational calculus:

Definition 2.8 (Semantics of the Calculus) *Let* DB *be a relational database over* D *and the schema* ρ*,* φ *a formula of the relational calculus, and* θ *a valuation, i.e., an assignment of values to variable names. We define* $\mathrm{DB}, \theta \models \varphi$ *by induction on the structure of* φ*:*

$\mathrm{DB}, \theta \models r_i(x_{i_1}, \ldots, x_{i_{\mathrm{arity}(r_i)}})$	if $r_i \in \rho$ and $(\theta(x_{i_1}), \ldots, \theta(x_{i_{\mathrm{arity}(r_i)}})) \in R_i^{\mathrm{DB}}$ for r_i atomic,
$\mathrm{DB}, \theta \models x_i = x_j$	if $\theta(x_i) = \theta(x_j)$
$\mathrm{DB}, \theta \models \varphi \wedge \psi$	if $\mathrm{DB}, \theta \models \varphi$ and $\mathrm{DB}, \theta \models \psi$
$\mathrm{DB}, \theta \models \neg\varphi$	if it is not the case that $\mathrm{DB}, \theta \models \varphi$
$\mathrm{DB}, \theta \models \exists x.\varphi$	if there is an $a \in \lvert D \rvert$ such that $\mathrm{DB}, \theta[x/a] \models \varphi$

where R_i^{DB} *is the interpretation of the predicate symbol* R_i *in the database* DB*.*

The answer to a query φ over the database DB *is the set of tuples (an* answer relation*):*

$$\{\theta_{|FV(\varphi)} : \mathrm{DB}, \theta \models \varphi\}$$

where $\theta_{|FV(\varphi)}$ *is the restriction of the valuation* θ *to free variables of the formula* φ*.*

Note that the queries can be thought of as functions mapping an input database to an output database that contains just the answer relation for the query.

The usual way of implementing the query evaluation algorithm in the relational database management systems is by translating the calculus queries into relational algebra, and then evaluating the resulting algebraic terms by having a relatively simple algorithm for every elementary algebra operation. To define an algebra, we need to define the universe and the operations on the universe: in this case the universe contains all possible user-defined relations, i.e., all extents of the predicate symbols in $\mathcal{R}$.

[2]These are all valid concerns. However, they are orthogonal to the topics of this dissertation.

Definition 2.9 (First-order Definable Algebra) *Let D be a data domain and $\mathcal{R}$ the set of allowed relational types. Then*

The universe of a first-order definable algebra *is the set*

$$U = \{R : R \subseteq |D|^{\text{arity}(r)} \text{ for some } r \in \mathcal{R}\}$$

Let τ be a relational type[3]*. We define a subset of all relations of the type τ in the universe U as:*

$$\tau(U) = \{R : R \subseteq |D|^{\text{arity}(r)} \text{ for some } r \in \mathcal{R} \text{ of the type } \tau\}$$

A function

$$f : \tau_1(U) \to \ldots \to \tau_n(U) \to \tau(U)$$

is a n-ary first-order definable relational operator of the type $\tau_1 \to \ldots \to \tau_n \to \tau$ *if there is a relational calculus formula φ and a database schema $\rho = (r_1, \ldots, r_k)$ such that:*

1. *the free variables of φ have the relational type τ, i.e., the relation $\{\theta_{FV(\varphi)} : D, \theta \models \varphi\}$ has type τ,*

2. *the the relational type of r_i is τ_i, and*

3. *$\{\theta_{|fv(\varphi)} : D \cup R, \theta \models \varphi\} = (f\ R_1\ R_2\ \ldots\ R_k)$ for every instance $(R_1, \ldots, R_k)$ of ρ.*

A safe relational operator *maps finite instances of ρ to finite relations and thus can be defined by a safe relational calculus formula.*

A first-order definable algebra *is a set of relational operators of bounded arity defined by first-order formulas of bounded quantifier depth.*

Let $\rho = (r_1, \ldots r_k)$ be a database schema and Γ a first-order definable algebra. An algebraic query *has the form defined by the following rules:*

1. *$r_i : \tau_i$ for $0 < i \leq k$, and*

2. *$(f\ t_1\ \ldots\ t_n) : \tau$ for $f : \tau_1 \to \ldots \to \tau_n \to \tau \in \Gamma$ and $t_i : \tau_i$ algebraic queries.*

An answer to an algebraic query *q over the schema ρ on the database D is the relation $[\![q]\!]$ defined inductively on the structure of q:*

$$\begin{array}{ll} [\![r_i]\!] & = R_i^D \\ [\![f\ t_1\ \ldots\ t_n]\!] & = f\ [\![t_1]\!]\ \ldots\ [\![t_n]\!] \end{array}$$

The main intuition behind the definition of the relational algebra is as follows: we try to define few generic operations on relations for which we have efficient algorithms. Then we can evaluate queries in terms of these operations as composites of functions defined by these operators. An example of such an algebra is the standard relational algebra defined as follows:

[3] In the single-sorted case, we could use arities of the predicate symbols or relations instead of the relational types. However, to allow multi-sorted relations, we define the algebra operations with respect to the types of the relations rather than with respect to the arities.

Example 2.10 (Relational Algebra) Let D be a data domain and U be the class of all relations of finite arity over D. We define following operators over U:

Product $\times : D^{k_1} \to D^{k_2} \to D^{k_1+k_2}$ defined over the schema (r_1, r_2) by the formula

$$R_1(x_1, \ldots, x_{k_1}) \wedge R_2(y_1, \ldots, y_{k_2})$$

for every $k_1, k_2 \geq 0$.

Projection $\exists x_i : D^k \to D^{k-1}$ defined over the schema (r) by the formula

$$\exists x_i . R(x_1, \ldots, x_k)$$

for every $k > 0$ and $0 < i \leq k$.

Selection $\sigma_F : D^k \to D^k$ defined over the schema (r) by the formula

$$R(x_1, \ldots, x_k) \wedge F$$

for all $k \geq 0$ and F quantifier-free first-order formula in the language of D.

Union $\cup : D^k \to D^k \to D^k$ defined over the schema (r_1, r_2) by the formula

$$R_1(x_1, \ldots, x_k) \vee R_2(x_1, \ldots, x_k)$$

for every $k \geq 0$.

Set difference $- : D^k \to D^k \to D^k$ defined over the schema (r_1, r_2) by the formula

$$R_1(x_1, \ldots, x_k) \wedge \neg R_2(x_1, \ldots, x_k)$$

for every $k \geq 0$.

The relational types are determined uniquely by the relation arities in this case; we used this fact in the definition of the relational algebra. Note that the arity of all the operators is bounded by 2 and the quantifier depth of the defining formulas by 1; thus the family of all such operators forms a relational algebra. Moreover, it is easy to verify that all the operators are safe.

We could also define the operator $(.)' : D^k \to D^k$ on the schema (r) defined by the formula

$$\neg R(x_1, \ldots, x_k)$$

for all $k \geq 0$. Clearly, this is also a first-order definable operator. However, it is easy to see that this operator is not safe if the data domain D is infinite.

Note that the restriction on the definition of the operators is quite natural: in general we need an infinite family of operators. However, as in Example 2.10, they can be thought as a finite set of operators *polymorphic* in the arities of the relations[4], and thus the quantifier depth of the defining formulas and the arities of the operators are bounded.

The correctness of evaluating calculus queries by translation to algebra is shown by the following proposition:

[4]This is true even for selection, as there are only finitely many quantifier-free formulas of a given arity up to logical equivalence.

Proposition 2.11 ([31]) *The following languages can express the same queries over relational databases:*

1. *relational algebra* $\times, \sigma_F, \pi_X, \cup, (.)'$ *and relational calculus,*
2. *Safe relational algebra* $\times, \sigma_F, \pi_X, \cup, -$ *and safe relational calculus.*

2.2 Temporal Databases

To generalize this methodology for the use with the temporal databases, we use essentially the same steps:

1. Choice of domains:

- the data domain: we use the same data domain as in the single-sorted case (cf. Definition 2.1).
- the temporal domain: similarly, we choose a first-order structure to be the model of time in our class of temporal databases; the elements of the domain serve as the representation of the time instants, and the relations defined on the structure determine the relationships between the individual time instants—the time ontology. We usually require an order relation to be defined on the temporal domain. The common choices of a temporal domain are the linearly ordered integers $(Z, <)$ for modeling discrete unbounded time or the rationals $(Q, <)$ to model dense time.

2. Choice of allowed relational types of the user-defined relations:

- uniform temporal relations: this is the most common choice. All relations are required to have exactly one temporal attribute (or a fixed number of such attributes). This corresponds to the relational types $T \times D^i, i \geq 0$, where T stands for the temporal sort and D for the data sort.
- temporal relations with varying temporal arity (unrestricted relational types): here we relax the uniformity requirement, and allow the relational types $T^i \times D^j$ for $i, j \geq 0$.

Note that in the case of the unrestricted schema we do not consider predicate symbols of arbitrary relational types, e.g., $T \times D \times T$, as we can always *permute* the attributes in the relations to obtain a relation of the sort $T^i \times D^j$.

2.2.1 Temporal Databases as First-order Structures

In the case of temporal databases we distinguish between the first-order structures and their finite encodings. This distinction was not necessary in the case of relational databases, where the finite encodings were just the finite structures themselves, and thus they formed a proper subclass of all possible relational databases. However, to avoid the restriction to finite structures, we distinguish the class of all temporal databases (the *Abstract Temporal Databases*), and the class of their finite encodings—the *Concrete Temporal Databases*.

The class of the abstract temporal databases is defined analogously to the class of the relational databases: we simply add the temporal domain that models time, and choose the appropriate types of the user-defined relations:

Definition 2.12 (Abstract Temporal Database) *Let D be a data domain, T a temporal domain, and $\rho = (r_1, \ldots, r_k)$ a database schema. Then*

$$DB = T \cup D \cup \{R_1, \ldots, R_k\}$$

is an Abstract Temporal Database *for any instance of ρ over T and D. Let* ATDB *be the class of all abstract temporal databases over T and D.*

Note that ρ has to contain only predicate symbols of the "allowed" relational types, i.e., $T \times D^i$ in the case of the uniform schema and $T^i \times D^j$ otherwise.

Definition 2.13 (Concrete Temporal Database) *Let S be a countable set and*

$$\|.\| : S \to \text{ATDB}$$

a meaning function. Then $(S, \|.\|)$ is the class of Concrete Temporal Databases *over the class* ATDB*.*

This definition captures the idea that S is the set of "codes" for the individual temporal databases we are interested in, and $\|.\|$ is the function which captures the meaning of the individual codes. We can store each of the individual "codes" as a number in sufficiently large computer storage.

The particular classes $(S, \|.\|)$ of concrete temporal databases can be defined in various ways:

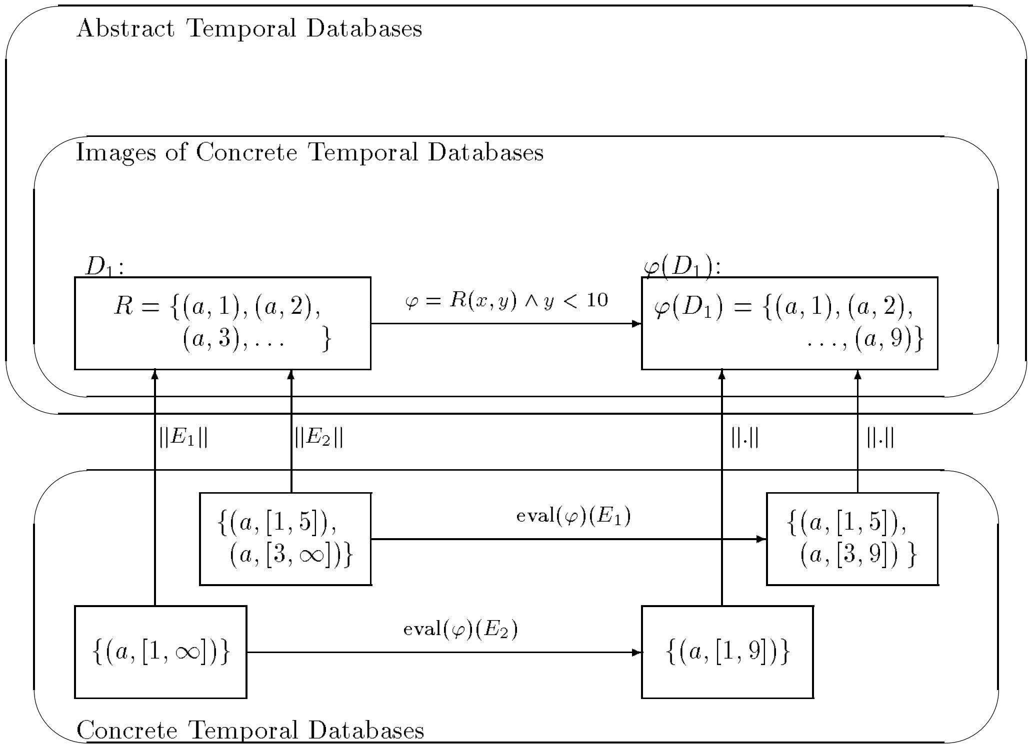
- we can store only finite structures—approach taken by the RDBMS. Then the encodings are the structures themselves, and the meaning function $\|.\|$ is a simple embedding.
- we can finitely encode *interesting* classes of infinite structures, e.g., constraint databases, where the codes can be formulas of first-order logic, and the meaning function is "all the satisfying valuations" function.

The disadvantage of the first solution is that the only sets of time instants storable in the database are finite: this means that in, e.g., the case of dense time, the only allowed sets are singletons or finite unions of those. This may not be sufficient for representing temporal databases. Consider the following examples:

Example 2.14 (Interval Representation) Currently this is the most common approach to encodings of infinite temporal databases over a linearly ordered temporal domain: we encode the possibly infinite intervals of the temporal domain by their endpoints—the elements of $|T| \cup \{-\infty, \infty\}$, assuming $|T|$ is countable. Then the meaning function $\|.\|$ is the uniform extension of the mapping of pairs

$$\|.\| : [a, b] \mapsto \{t : a \leq t \leq b\}$$

and the identity on the structure D to the whole temporal database. Note that the data domain is still mapped using the identity mapping as in the relational case; thus, the user-defined relations have to be finite at every time instant.



E_1 and E_2 are two concrete temporal databases that represent the abstract temporal database D_1.

Figure 2.1: Temporal Databases as First-order Structures

Figure 2.1 shows an example of a query, evaluated over the interval representation of temporal databases. Clearly, the intervals can also be represented by quantifier free formulas in the language of the temporal domain, e.g., $[a,b]$ can be represented by the formula $\varphi_{a,b}(t) = a \leq t \leq b$. We can generalize this idea in a more general way by representing sets of time instants using quantifier-free characteristic formulas:

Example 2.15 (Representation using Constraints) Let L be the language of the temporal domain. We denote $T(L)$ a subset of the quantifier-free formulas of L and $R_{D,T(L)}$ the class of all two-sorted finite relational databases over the sorts D and $T(L)$. We consider these databases as the finite encodings of the abstract temporal databases over D and T. The meaning function is the identity on D and the

$$||.|| : \varphi \mapsto \{\theta_{|FV(\varphi)} : \theta \models \varphi\}$$

mapping of formulas in $T(L)$ to sets of tuples over T. This mapping can be uniformly extended to the meaning mapping $||.||$ from concrete databases to abstract databases.

2.2.2 Temporal Query Languages

The abstract definition of the class of abstract temporal databases allows the immediate use of the query languages, based on logic: in this dissertation we consider the following formalisms:

1. temporal logic (a modal extension to the single-sorted relational calculus, cf. Chapter 3),
2. relational calculus and algebra that can be easily extended from the single-sorted relational databases to the two-sorted (or multi-sorted in general) case of the abstract temporal databases, (cf. Chapter 3), and
3. deductive queries over the temporal databases (cf. Chapters 4, 5, and 6).

However, while the meaning of the query languages is defined with respect to the *abstract temporal databases*, we need to provide conditions that allow us to answer such queries by using a *query evaluation algorithm*, that works on the class of the *concrete temporal databases.*

Definition 2.16 (Query Evaluation Algorithm) *Let $(S, \|.\|)$ be a class of the concrete temporal databases and $\mathcal{L}$ a query language. Then a* query evaluation algorithm *is a recursive function*

$$\mathrm{eval} : \mathcal{L} \to S \to S$$

The relationship between the encoding ($\|.\|$) and the query evaluation algorithm eval is captured by the following definitions:

Definition 2.17 (Closure) *Let $\|.\|$ be an encoding of temporal databases and $\mathcal{L}$ query language over the abstract temporal databases. Then $\mathcal{L}$* is closed with respect to $\|.\|$ *if for every $\varphi \in \mathcal{L}$ there is a concrete temporal database D', such that*

$$\varphi(\|D\|) = \|D'\|$$

for every concrete database D.

This definition is used to identify a subclass of the query language for which the results of queries can be finitely represented. This is analogous to the definition of the *safe relational calculus* queries in the case of finite relational databases. In our case, we use a more general definition that depends on the chosen class of finite encodings—the concrete temporal databases. However, the closure itself is not sufficient; we need an effective way of evaluating queries over the finite encodings that preserves the meaning of the queries:

Definition 2.18 (Faithful Encoding) *Let $\|.\|$ be an encoding of temporal databases and* eval *a query evaluation algorithm on the encodings. Then $\|.\|$* is faithful encoding of temporal databases with respect to eval *if*

$$\varphi(\|D\|) = \|\mathrm{eval}(\varphi)(D)\|$$

for all concrete temporal databases D and all queries φ on the abstract temporal databases from a query language closed with respect to $\|.\|$.

The faithfulness property guarantees that the semantics of queries is preserved by the query evaluation algorithm eval.

The last property guarantees that while an abstract temporal database can be encoded in different ways, i.e., $\|.\|$ does not have to be injective; the queries evaluated on different encodings of the same abstract database yield the same results, again, modulo the encoding:

Definition 2.19 (Generic queries) *Let $\|.\|$ be an encoding of temporal databases and φ a query on the class of concrete databases. We say that φ* is generic with respect to $\|.\|$ *if*

$$\|D_1\| = \|D_2\| \supset \|\varphi(D_1)\| = \|\varphi(D_2)\|$$

for all D_1 and D_2.

Clearly, queries that do not satisfy this condition cannot be expressed as queries over the abstract temporal databases, because only generic queries correspond to queries over the abstract temporal databases.

We would like to have a proposition similar to Proposition 2.11. This would allow us to implement the first-order temporal queries by translating them to temporal relational algebra (see Chapter 3). It is easy to see that the proposition can be extended to the two-sorted case for the temporal databases *with relations of varying temporal arity*. However, in Chapter 3 we show that this correspondence breaks down for the uniform temporal databases.

2.2.3 Complexity of Query Evaluation

The other important property of query evaluation algorithms is their complexity, i.e., how difficult is it to answer the queries: we need to study the complexity of the query evaluation algorithm eval. Note that the complexity is closely tied to the chosen query language and the class of concrete databases—the finite encodings. In [18] two different approaches to measuring the complexity of queries were introduced:

Definition 2.20 (Data Complexity) *Let $\mathcal{L}$ be a closed query language and D an arbitrary concrete database. The query evaluation algorithm* eval *has* data complexity $f(n)$ *if for every query $\varphi \in \mathcal{L}$ the execution of* $\text{eval}(\varphi)(D)$ *takes at most $f(n)$ steps on a concrete database D of size n.*

The rationale behind this choice is that the size of the database is usually much bigger than the size of the (representation of the) query. Thus, the complexity of query evaluation is defined as a function of the size of the database, while ignoring the size of the query itself[5]. A stronger result can be obtained using

Definition 2.21 (Combined Complexity) *Let $\mathcal{L}$ be a closed query language and D an arbitrary concrete database. The query evaluation algorithm* eval *has* combined complexity $f(m, n)$ *if the execution of* $\text{eval}(\varphi)(D)$ *takes at most $f(m, n)$ steps on a concrete database D of size n and query $\varphi \in \mathcal{L}$ with number of symbols m.*

This is the "standard" complexity of the eval algorithm with respect to the size if its inputs.

[5]More precisely, we study the complexity for every possible fixed query as a function of the size of the concrete database.

Bibliographical Notes

The relational model was introduced to computer science in [30], including Proposition 2.11. However, many of the concepts can be found much earlier in the work of Tarski [95]. A standard introduction to relational database theory can be found in various textbooks, e.g, [100] or a brand new book [2] with very crisp and mathematically precise definitions. Many of the concepts are based on the classical first-order logic and its model theory. An introduction to these concepts can be found in virtually any book on mathematical logic, e.g., [67] or [35]. Algebras for capturing first-order logic can be found in [49]. A thorough discussion of the application of these methods to temporal databases can be found in a survey of temporal databases and query languages [20].

Chapter 3

Expressive Power

Queries over temporal databases involve references to time. We study differences between two approaches of including such references into a first-order query language (e.g., relational calculus): explicit (using typed variables and quantifiers) vs. implicit (using a finite set of modal connectives). We also show that though the latter approach—a first-order query language with implicit references to time—is appealing by its simplicity and ease of use, it cannot express all queries expressible using the first one in general. This result also settles a longstanding open problem about the expressive power of first-order temporal logic. A consequence of this result is that there is no first-order complete query language subquery-closed with respect to a uniform database schema, and thus we cannot use temporal relational algebra over uniform relations to evaluate all first-order definable queries.

3.1 Introduction

In the last several years, various languages for querying temporal databases have been proposed in the literature. The first-order query languages can be divided into two main categories:

1. query languages based on a two-sorted version of relational calculus, e.g., TSQL2 [84] or TQUEL [83], and
2. query languages based on an extension of the relational calculus or algebra by temporal operators, e.g, HRDM's historical relational algebra [27], temporal relational algebra [99], etc.

These two approaches have been often considered equivalent in expressive power, e.g., in [99], where the temporal relational algebra based on an extension of the relational calculus by temporal operators has been proposed as a basis for first-order completeness of temporal query languages. This assumption was based on a well-known property of propositional temporal logic:

Proposition 3.1 (Kamp's Theorem) *Propositional temporal logic has an expressively complete set of temporal connectives over linear orders.*

This means that the propositional temporal logic has the same expressive power as the monadic first-order logic over linear orders. This result has been established by Kamp [56] for complete linear orders, later extended by Stavi [88] for all linear orders, and reproven several times using various proof techniques, e.g, [39, 50].

In this section we show that this correspondence does *not* generalize to the relationship between the First-order Temporal Logic (FOTL) and the Two-sorted First-order Logic (2-FOL). The query languages based on 2-FOL are strictly more expressive than the languages based on FOTL *for any finite set of temporal connectives* defined over a (dense) linear order. We show that FOTL cannot express a large class of queries that involve references to several time instants, like equality of database relations at two different time instants. The result can be interpreted as

Temporal query languages cannot be simultaneously
subquery-closed[1] *and first-order complete.*

Such a result has strong implications for the design of temporal query languages: essentially all subquery-closed relational algebra-like languages are incomplete, while all complete relational calculus-like languages do not preserve closure. This is a problem from the implementation point of view: while the subquery-closed query languages are easier to implement, essentially because all intermediate results in a bottom-up query evaluation have the same structure as database relations, there are first-order queries that cannot be formulated in such languages. Despite this drawback, FOTL was used as a basis of temporal query languages [99] or for specification of temporal integrity constraints [21, 22, 25, 26, 64]. The main reason for this choice is a simpler and more efficient implementation that does not generalize to full FOTL. The design of an expressively complete Temporal Relational Algebra requires an unbounded number of temporal attributes in the relations to guarantee completeness and preserve closure of products [94]. However, such solution does not allow to store the intermediate results needed to evaluate the query in a uniform temporal database as auxiliary relations.

In addition to the separation of FOTL from 2-FOL we show several other results, especially:

1. Expressive equivalence of the *future* temporal logic and the *full* temporal logic, i.e., with both future and past temporal connectives [41] does not generalize to the first-order case.

2. Expressive completeness cannot be achieved using more general temporal logic, e.g., the many-dimensional first-order temporal logic [42].

Note also that we restrict our attention solely to the first-order languages. Thus any higher-order properties (e.g., the property relations *even cardinality* given in [102]) are not expressible in either 2-FOL or FOTL.

[1]Assuming closure with respect to the database schema, where all the temporal relations, including derived ones, have exactly one temporal attribute or, in general, a bounded number of such attributes. This is a natural requirement imposed on virtually all practical temporal databases [94], e.g., on the timestamp relations in TSQL2 or TQUEL.

The rest of the chapter is organized as follows: Section 3.2 introduces the formal definition of the temporal database and first-order temporal query languages over such databases. Sections 3.3 and 3.4 introduce the model-theoretic games that allow us to study the expressive power of FOTL. In Section 3.5 we use these games to show our main result: FOTL$\subset$2-FOL[2]. Section 3.6 shows several other non-expressibility results that can be established using games. In the last section of this chapter we summarize the obtained results and conclude with the directions of further research.

3.2 First-order Query Languages over Temporal Databases

To show the separation of the two first-order languages, we need to fix the class of structures, over which these languages are defined. We use the approach from the previous chapter in slightly more general form:

The temporal structures considered in this section are built from the following three basic building blocks:

1. a first-order signature $\langle \rho_1, \ldots, \rho_n \rangle$ and a class of first-order structures $\langle T; \rho_1^T, \ldots, \rho_n^T \rangle$. We use a single structure from this class to serve as the *temporal domain.*
2. a first-order signature $\langle p_1, \ldots, p_l \rangle$ and a class of first-order structures $\langle D; p_1^D, \ldots, p_l^D \rangle$. Again, a single structure serves as the *data domain* of the database.
3. a single-sorted set of predicate symbols $\langle r_1, \ldots, r_m \rangle$; the arity of the symbol r_i is v_i. This choice defines the *database schema* for our temporal database.

This arrangement abstracts from the particular choice of the data and temporal domains and thus allows a wide range of applications of the proposed method.

In the rest of this chapter we use γ to denote the signature $\langle \rho_1, \ldots, \rho_n \rangle$ of the temporal domain, δ for the signature $\langle p_1, \ldots, p_l \rangle$ of the data domain, and σ for the signature $\langle r_1, \ldots, r_m \rangle$ of the database schema.

The temporal databases, considered in this chapter, are the uniform extensions of the single-sorted database schema ρ by a single distinguished temporal attribute:

Definition 3.2 (Uniform Temporal Database) *Let $\langle T, \rho_1^T, \ldots, \rho_n^T \rangle$ be a temporal domain (signature γ), $\langle D; p_1^D, \ldots, p_l^D \rangle$ a data domain (signature δ), and $\sigma = \langle r_1, \ldots, r_m \rangle$ a database schema. We define R_i to be two-sorted relation symbols of the sort $T \times D^{v_i}$ for each r_i in the database schema σ. We call R_i the* uniform temporal extension *of r_i. A* temporal signature *$\tau = \langle \rho_1, \ldots, \rho_n, p_1, \ldots, p_l, R_1, \ldots, R_m \rangle$ is a two-sorted signature composed of γ, δ, and the temporal extensions of all the symbols in σ. We define* a temporal database *$\mathcal{A}$ to be a two-sorted τ-structure*

$$\mathcal{A} = \langle T, \rho_1^T, \ldots, \rho_n^T; D, p_1^D, \ldots, p_l^D; R_1^{\mathcal{A}}, \ldots, R_m^{\mathcal{A}} \rangle$$

such that the sets $\{(x_1, \ldots, x_{v_i}) : \mathcal{A} \models R_i(t, x_1, \ldots, x_{v_i})\}$ are finite for every $t \in T$ and $0 < i \leq m$. The extensions $R_i^{\mathcal{A}}$ of R_i in $\mathcal{A}$ define the interpretation of the symbols r_i in the database schema for every element of the time domain, formally:

$$r_i(c_1, \ldots, c_{v_i}) \text{ holds at time } t \text{ iff } \mathcal{A} \models R_i(t, c_1, \ldots, c_{v_i})$$

[2] $\subset$ denotes a proper inculsion.

for $r_i \in \sigma$, $t \in T$, and $c_i \in D$.

Note that the interpretations of the predicate symbols, connected solely with the temporal domain (e.g., $<$) or the data domain (e.g., $=$), is fixed, while the interpretation of the symbols R_i depends on the actual contents of the database.

Example 3.3 Let $\langle Q; < \rangle$ be the temporal domain (dense linearly ordered rationals), $\langle D; = \rangle$ the data domain, and ρ the database schema consisting of a single binary relation $\rho = \langle \text{salary} \rangle$ holding employee IDs and salaries. Then the temporal structure representing such a temporal database is defined as

$$\text{DB} = \langle Q, <^Q; D, =^D; \text{Salary} \rangle$$

where the fact *x has salary y at time t* is encoded by

$$\text{salary}(x, y) \text{ holds at time } t \text{ in DB} \iff \text{DB} \models \text{Salary}(t, x, y)$$

Note that the relations $<^Q$ and $=^D$ have fixed meanings for the given domains, e.g., $<^Q$ is the linear order on Q and $=^D$ is the diagonal on D. On the other hand the interpretation of the relation Salary depends on the actual *contents* of the database. The relation Salary is infinite in general, but the sets $\{(x, y) : Salary(t, x, y)\}$ are finite for every fixed $t \in T$.

The choice of a first-order query language over such extension is not as straightforward: obviously we can use the two-sorted relational calculus or equivalently the two-sorted first-order logic (2-FOL) to express queries over such structures. The disadvantage of such a solution is that it refers to the *augmented* database schema rather than to the original schema. Moreover, all the references to *time* are explicit—we use variables and quantification over the time domain.

A more elegant solution is to use the first-order temporal logic (FOTL)—the single-sorted first-order logic augmented with a finite set of *temporal connectives* like 'sometime in the past' or 'always in the future'. This solution is preferable for two reasons: first, it refers to the *original* database schema rather than to the extended schema. However, the main advantage of this solution is that the references to time and to all the properties of the underlying temporal domain are *encapsulated* inside the temporal connectives added to the language—all the references to time are *implicit*.

We define two query languages over the uniform temporal databases represented by the τ-structures from Definition 3.2:

1. the two-sorted first-order logic language $\mathcal{M}$ and

2. the first-order temporal logic language $\mathcal{T}$.

The definition of the first language is a straightforward extension of definition 2.7. Let t_i be variables of sort T and x_i be variables of sort D.

Definition 3.4 *Let γ be the signature of the temporal domain and δ the signature of the data domain. A two-sorted FOL language $\mathcal{M}(\gamma, \delta)$ over the database schema σ is defined by the BNF rule:*

$$M ::= R(t_j, x_1, \ldots, x_{v_i}) \mid \rho(t_1, \ldots, t_{w_k}) \mid p(x_1, \ldots, x_{u_i}) \mid M \wedge M \mid \neg M \mid \exists x_i.M \mid \exists t_i.M$$

where R is the temporal extension of r for $r \in \sigma$, $\rho \in \gamma$, and $p \in \delta$.

The semantics of the formulas in this language is the standard first-order Tarskian semantics (cf. Definition 2.8).

Note that the database schema is *monadic* with respect to the sort T, i.e., the predicate symbols in the database schema have always exactly one distinguished argument of sort T. To preserve the *closure* of the query language we restrict the set of the *valid* queries to $\mathcal{M}$-formulas that have exactly one free variable over sort T , so the result of the query can be stored in the temporal database itself (cf. Definition 2.17). It is easy to see that the closure has been achieved artificially, and that not all $\mathcal{M}$ formulas are valid queries. Moreover, subformulas of a valid query in $\mathcal{M}$ may not be valid queries themselves.

Example 3.5 Consider the formula

$$\exists t_1, t_2 . t_1 < t < t_2 \wedge \forall x . P(t_1, x) \iff P(t_2, x).$$

This formula defines the query *"find all the intervals such that P contains exactly the same values on both the ends of each of the intervals"*. Clearly, this is a valid $\mathcal{M}$ query. However, the subformula

$$\forall x . P(t_1, x) \iff P(t_2, x)$$

is not a valid query as it contains two free variables of sort T: t_1 and t_2. Moreover, in Section 3.5 we show that there is no $\mathcal{M}$ formula with the property that all its subformulas are valid queries and that expresses the original query.

In the definition of the second query language we need to be more careful. The language is defined in two steps: first the temporal connectives are defined. Note that the definition of the temporal connectives is independent of both the data domain and the database schema. The definition of the temporal connectives is purely syntactical and depends only on the chosen temporal domain. The semantics of the connectives is defined by a translation to 2-FOL (cf. Definition 3.11). This approach differs from the usual definition of such connectives given in [39] or [40], that is based on the intended semantics and uses truth tables.

Definition 3.6 (Temporal Connective) *Let γ be the signature of the temporal domain and*

$$O ::= \rho(t_1, \ldots, t_{w_k}) \mid O \wedge O \mid \neg O \mid \exists t_i . O \mid X_i$$

where X_i are predicate variables and $\rho \in \gamma$. A (k-ary) temporal connective *(over γ) is an O-formula with exactly one free variable t_0 and k free predicate variables $X_1, \ldots, X_k$. We denote such connective by*

$$\omega(X_1, \ldots, X_k).$$

Let $\Omega(\gamma)$ denote a finite set of (names for) temporal connectives over γ.

The superscripts t_i denote explicitly the temporal contexts of the appropriate subformulas that the temporal connective glues together.

Definition 3.7 *If every quantification in a temporal connective $\omega \in \Omega(\gamma)$ is of the form $\exists t_i . (\rho(t_0, t_i) \wedge O)$ then ω is a ρ-*restricted temporal connective.

Example 3.8 We can express the usual temporal connectives given in [42] in the temporal signature $\langle < \rangle$ of linear orders as follows:

$$\begin{aligned}
X_1 \,\textbf{until}\, X_2 &\triangleq \exists t_2 . t_0 < t_2 \wedge X_2 \wedge \forall t_1 (t_0 < t_1 < t_2 \rightarrow X_1) \\
\Diamond X_1 &\triangleq \exists t_1 . t_0 < t_1 \wedge X_1 \\
\Box X_1 &\triangleq \forall t_1 . t_0 < t_1 \rightarrow X_1
\end{aligned}$$

Similarly we can express the past temporal connectives **since**, ◆, and ■:

$$\begin{aligned}
X_1 \,\textbf{since}\, X_2 &\triangleq \exists t_2 . t_0 > t_2 \wedge X_2 \wedge \forall t_1 (t_0 > t_1 > t_2 \rightarrow X_1) \\
\blacklozenge X_1 &\triangleq \exists t_1 . t_0 > t_1 \wedge X_1 \\
\blacksquare X_1 &\triangleq \forall t_1 . t_0 > t_1 \rightarrow X_1
\end{aligned}$$

Note that the connectives **until**, $\Diamond$, and $\Box$ are $>$-restricted while the connectives **since**, ◆, and ■ are $<$-restricted. We call the temporal logics that use only the $>$-restricted (the $<$-restricted) temporal connectives the *Future* FOTL (the *Past* FOTL), respectively.

The second step in the definition of the temporal extension to the language of first-order logic is straightforward: we augment the syntax of the FOL over the signatures of the data domain and the database schema by a finite set of temporal connectives:

Definition 3.9 *Let $\Omega(\gamma)$ be a finite set of temporal connectives over γ and δ be the signature of the data domain. A* First-order Temporal Logic Language $\mathcal{T}(\Omega(\gamma), \delta)$ *over the database schema σ is defined by the BNF rule:*

$$F ::= r(x_1, \ldots, x_{v_i}) \mid p(x_1, \ldots, x_{u_i}) \mid \omega(F_1, \ldots, F_k) \mid F \wedge F \mid \neg F \mid \exists x . F$$

for $r \in \sigma$, $p \in \delta$ and $\omega \in \Omega(\gamma)$.

In the rest of the chapter we use $\mathcal{M}$ and $\mathcal{T}$ in place of $\mathcal{M}(\gamma, \delta)$ and $\mathcal{T}(\Omega(\gamma), \delta)$, respectively.

Definition 3.10 *We say that $\mathcal{T}$ is a* Propositional Temporal Logic (PTL) *if all the relations in the database schema σ are 0-ary.*

It is easy to see that all the $\mathcal{T}$-formulas can be naturally embedded into the language of $\mathcal{M}$-formulas. This embedding also defines the semantics of the $\mathcal{T}$-formulas relatively to the semantics of $\mathcal{M}$:

Definition 3.11 (Embedding of $\mathcal{T}$ into $\mathcal{M}$) *Let* Embed *be a mapping of formulas in the language $\mathcal{T}(\Omega(\gamma), \delta)$ to the language $\mathcal{M}(\gamma, \delta)$ defined as follows:*

$$\begin{aligned}
\text{Embed}(r_i(x_1, \ldots, x_{v_i})) &= R_i(t_0, x_1, \ldots, x_{v_i}) \\
\text{Embed}(p_i(x_1, \ldots, x_{v_i})) &= p_i(x_1, \ldots, x_{v_i}) \\
\text{Embed}(F_1 \wedge F_2) &= \text{Embed}(F_1) \wedge \text{Embed}(F_2) \\
\text{Embed}(\neg F) &= \neg\, \text{Embed}(F) \\
\text{Embed}(\exists x . F) &= \exists x .\, \text{Embed}(F) \\
\text{Embed}(\omega(F_1, \ldots, F_k)) &= \omega^*(\text{Embed}(F_1)[t_0/t_1], \ldots, \text{Embed}(F_k)[t_0/t_k])
\end{aligned}$$

where ω^* is the (O-)formula denoted by ω in $\Omega(\gamma)$ and $F[t_0/t_i]$ is a substitution of t_i for t_0 in F.

In the following development we assume that all the $\mathcal{T}$-formulas are merely restricted $\mathcal{M}$-formulas, i.e., $\mathcal{T} \subseteq \mathcal{M}$. The translation of all $\mathcal{T}$-formulas yields a valid 2-FOL query as Embed(φ) has only one free variable of sort T, namely t_0. Thus, the translation of $\mathcal{T}$-formulas preserves the closure of the query language. Note that the translation of every subformula of the original formula has this property as well. This is obviously not true for subformulas of an arbitrary $\mathcal{M}$-formula even if the original formula is a valid query (cf. Example 3.5).

Example 3.12 Using the temporal database from Example 3.3 we can formulate the following queries in both 2-FOL and FOTL:

personal ID of everyone whose salary was negative

FOTL: $\blacklozenge \exists y.\text{salary}(x, y) \wedge y < 0$

2-FOL: $\exists t_1.t_1 < t_0 \wedge \exists y.\text{Salary}(t_1, x, y) \wedge y < 0$

personal ID of everyone whose salary has decreased

FOTL: $\exists y_1, y_2 \blacklozenge(\text{salary}(x, y_1) \wedge \blacklozenge \text{salary}(x, y_2)) \wedge y_1 < y_2$

2-FOL: $\exists y_1, y_2 \exists t_1(t_1 < t_0 \wedge \text{Salary}(t_1, x, y_1) \wedge \exists t_2.t_2 < t_1 \wedge \text{Salary}(t_2, x, y_2)) \wedge y_1 < y_2$

This example also illustrates why the use of FOTL is easier than the use of 2-FOL: in FOTL we do not have to introduce any of the variables t_i—the references to time are encapsulated in the temporal connectives, e.g.,$\blacklozenge$.

It is easy to see the essential difference between the full 2-FOL language $\mathcal{M}$ and the temporal extension to the first-order logic $\mathcal{T}$: each of the subformulas of a $\mathcal{T}$-formula is associated with exactly one temporal context represented by the variable t_0 in the translation to $\mathcal{M}$. Moreover the context can be changed only using a temporal connective. This distinction comes into the play when the temporal connectives, which are merely quantifiers over sort T, and the quantifiers over sort D are interleaved in a single formula.

It is quite clear that any two τ-structures distinguishable by a $\mathcal{T}$-formula φ are also distinguishable by a $\mathcal{M}$-formula, namely Embed(φ). However, two τ-structures distinguishable by a $\mathcal{M}$-formula may not be distinguishable by any $\mathcal{T}$-formula. In Section 3.5 we find concrete examples of structures that are distinguishable by a $\mathcal{M}$-formula but not distinguishable by any $\mathcal{T}$-formula.

3.3 Ehrenfeucht-Fraïssé Games

To show the separation of $\mathcal{L}$ from $\mathcal{M}$ we use a technique originally developed to capture the expressive power of the first-order logic—the Ehrenfeucht-Fraïssé Games [34, 38]. These games provide a complete combinatorial characterization of first-order definable properties; the main result can be summarized as:

two first-order structures can be distinguished by a first-order formula
if and only if
the same two structures can be distinguished by Ehrenfeucht-Fraïssé Games

Thus, if we can prove that two structures are *equivalent* (indistinguishable) with respect to the games, we know that there is no first-order formula that is true in one of the structures and false in the other one.

To make this idea more formal consider the following definitions:

Definition 3.13 (Ehrenfeucht-Fraïssé Game for First-order Logic) *Let A and B be first-order structures over the same signature. A* round of the Ehrenfeucht-Fraïssé Game of length *n is a sequence of n moves of the following form:*

1. Player I *picks an element from the carrier of A or B and then*

2. Player II *picks an element from the carrier of the remaining structure.*

Let the pairs of the chosen elements be $(a_1, b_1), \ldots, (a_n, b_n)$, where $a_i \in A$ and $b_i \in B$.

We say that Player II wins the round of the game *if the mapping defined by $a_i \mapsto b_i$ is a partial isomorphism on the structures $A \to B$.*

We say that Player II has a winning strategy *if* Player II *can win every round of the game of length n.*

Notation 3.14 *The winning strategy for* Player II *defines an equivalence relation on the first-order structures for every $n \geq 0$. We write $A \sim_n B$ if* Player II *has a winning strategy for all rounds of the Ehrenfeucht-Fraïssé game of length n.*

Proposition 3.15 (Ehrenfeucht [34]) *Let A and B be first-order structures of the same signature. Then*

$$A \sim_n B \text{ iff } \forall \varphi \in \mathcal{L}_n (A \models \varphi \iff B \models \varphi)$$

where $\mathcal{L}_n$ is the set of all first order sentences with at most n nested quantifiers (cf. Definition 3.25).

This proposition gives a method for showing first-order non-definability of properties (subclasses) of first-order structures:

Corollary 3.16 *Let P be a property of first order structures of a given signature and (A_n, B_n) pairs of such structures for every $n \geq 0$. To show that P is not first-order definable property it is sufficient to show*

1. *$A_n \in P$ and $B_n \notin P$ for all $n \geq 0$, and*

2. *$A_n \sim_n B_n$ for all $n \geq 0$.*

Example 3.17 (Characterization of Linear Orders) This technique can be used to show the well-known characterization of finite linear orders [79]:

Let L_m be a finite linear order of length m. Then

$$L_m \sim_k L_n \text{ iff } (m = n \text{ or } m, n \geq 2^k - 1)$$

P r o o f: We show the "if" direction only: if $m = n$ them $L_m \sim_k L_n$ trivially. Let $m, n \geq 2^k - 1$. Then the following is a winning strategy for Player II:

1. if Player I plays an element a in the first $(2^{k-1} - 1)$ elements of L_m then Player II answers with the corresponding element b in L_n (i.e., with the same distance from the left endpoint).

2. Similarly, if Player I plays an element in the last $(2^{k-1} - 1)$ elements of L_m then Player II answers with the corresponding element at the end of L_n.

3. otherwise Player II plays an arbitrary element whose distance from both endpoints is at least 2^{k-1}.

Similarly, if Player I picks an element from L_n, Player II uses a symmetric strategy to choose an element from L_m.

Every move splits the original linear orders into two parts: elements bigger than $(L_m^{>a}, L_n^{>b})$ and smaller than $(L_m^{<a}, L_n^{<b})$ the elements just played. Clearly, $|L_m^{>a}|, |L_n^{>b}| \geq (2^{k-1} - 1)$ or $|L_m^{>a}| = |L_n^{>b}|$ (similarly for $L_m^{<a}, L_n^{<b}$). Thus, $L_m^{>a} \sim_{k-1} L_n^{>b}$ and $L_m^{<a} \sim_{k-1} L_n^{<b}$. The result now follows by induction on k. The complete proof can be found in [79].

In particular, we can show that there is no formula

$$\varphi(R) = \begin{cases} \text{true} & \text{if } R \text{ contains an even number of tuples,} \\ \text{false} & \text{otherwise} \end{cases}$$

Clearly, to express this query we would need a first-order formula that distinguishes L_{2^k} and L_{2^k-1} for every $k > 0$. This is not possible by the characterization of the finite linear orders (cf. Example 3.17).

3.4 Ehrenfeucht-Fraïssé Game for Temporal Logic

Our situation is slightly more complicated, as we try to separate FOTL *from* first-order logic, i.e., we try to show that there are 2-FOL formulas that define properties not definable in FOTL.

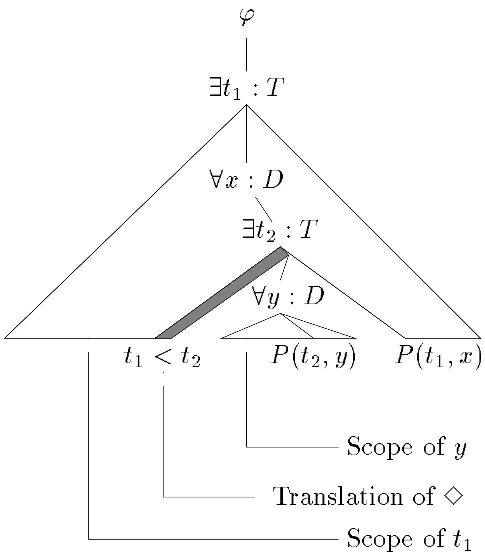
The rules of the game are modified to match exactly the syntactic restrictions placed on the temporal formulas by Definitions 3.6 and 3.9. The modification captures precisely the expressive power of FOTL.

Definition 3.18 *Let φ be a closed $\mathcal{T}$-formula and t_0 be the only free variable of the formula* Embed(φ)*. We define a (partial) order $\prec$ on the variables in* Embed(φ) *as follows:*

$$\begin{array}{l} t_0 \prec v_i \ \text{ for every } v_i \text{ in } \mathrm{Embed}(\varphi) \\ v_i \prec v_j \iff \exists v_j.\psi' \text{ is a subformula of } \exists v_i.\psi \text{ which is a subformula of } \mathrm{Embed}(\varphi) \end{array}$$

assuming that all quantified variables have unique names different from t_0.

Clearly, in the formulas of the form Embed(φ) for $\varphi \in \mathcal{L}$ variables that appear as arguments of a single atomic subformula (a leaf of the syntactic tree of Embed(φ)) follow a certain pattern. This pattern can be defined in terms of *compatibility* of individual variables.



The diagram is a graphical representation of the scopes of the individual variables of the sort T. We can easily see that the scope of the variable t_1 is not the whole subformula rooted by $\exists t_1 : T$ (as it would be in a 2-FOL formula). The scope of t_1 extends only to the point where other temporal connective (e.g., $\diamond$, represented by the shaded area) is encountered. While t_1 can still be used in the *body* of this connective (i.e., in the expansion of its definition—$t_1 < t_2$), it cannot be used in the subformula rooted by this connective, e.g., in $P(t_2, y)$.

Figure 3.1: Restriction on the scopes of the individual temporal variables.

An easy way of thinking about the compatibility of variables is by considering the scopes[3] of variables defined by the expansion of the temporal connectives: The restrictions imposed on the scopes of the individual variable names in the formula $\text{Embed}(\varphi)$ for $\varphi \in \mathcal{T}$ can be depicted as follows:

Example 3.19 Figure 3.1 shows the restrictions on the scopes of the individual variables in the translation $\text{Embed}(\varphi)$ for an FOTL formula φ.

In general the variables x and y can appear together as arguments of an atomic formula only if their scopes *intersect*. Note that this restriction does not have any impact on pairs of variables of sort D. This is very natural as the restrictions are connected solely with the sort T.

Following lemma shows which variables are *compatible* with respect to a temporal formula (i.e., which variables can appear as arguments of an atomic formula).

Lemma 3.20 *Let φ be a closed $\mathcal{T}$-formula and p an atomic subformula of* $\text{Embed}(\varphi)$*, such that both the variables v_i and v_j are arguments of p. Then*

[3]The parts of the syntactic tree of the formula, where the variable in question can legally appear.

if v_i is of sort T and v_j is of sort D then either
$v_j \prec v_i$ or
v_i is maximal in the set $\{v$ of sort $T : v \prec v_j\}$, or

if both v_i and v_j are of sort T and $v_i \prec v_j$ then either
v_k is of sort T for all v_k such that $v_i \prec v_k \prec v_j$ or
v_i is maximal in the set $\{v$ of sort $T : \exists v_k$ of sort $D . v \prec v_k \prec v_j\}$.

P r o o f: By induction on the structure of Embed(φ).

This lemma is used to define the winning condition of the game for temporal logic. The game itself is defined as follows:

Definition 3.21 (Game for Temporal Logic) *Let A and B be two τ-structures, $a = (a_1, \ldots, a_k)$ and $b = (b_1, \ldots, b_k)$ sequences of A-elements and B-elements (respectively), and $n \in N$. A* round of the game $T_n((A, a), (B, b))$ *consists of an ordered sequence of n moves $(a_{k+1}, b_{k+1}), \ldots, (a_{k+n}, b_{k+n})$ of the form "*PLAYER I *chooses an element in the carrier of either A or B and* PLAYER II *chooses an element in the carrier of the other structure". Both the players must obey the following rules:*

1. *the corresponding elements a_i and b_i must be of the same sort, and*
2. *the pair (a_1, b_1) must be of sort T.*

We say that i (i.e., i-th move) is a data move *if both a_i and b_i are of sort D, and that i is a* temporal move *if both elements are of sort T.*

We call the pair (a_1, b_1) the initial temporal context*, and the vectors a and b* the initial assignment.

The vectors $a = (a_1, \ldots, a_k)$ and $b = (b_1, \ldots, b_k)$ are used to handle the open formulas of $\mathcal{M}$ (cf. Theorem 3.30). Except for the additional rules, the game for temporal logic is the same as the Ehrenfeucht-Fraïssé game for first-order logic [79]. The first rule enforces the sort compatibility of the individual moves in the game—this is a natural requirement for the many-sorted structures (we can assume that the relations are empty if the arguments do not match the required sorts). The second rule is needed to define the initial temporal context, the subsequent moves are then relative to this context. Note that a closed temporal logic formula still has one free variable of sort T that represents the initial temporal context (cf. the semantic definition of the temporal logic in section 3.2 or [42]).

Besides the rules of the game itself we need to define the winning condition for the game for temporal logic. Here we use the observation from Example 3.19:

Definition 3.22 (Compatible Moves) *Let $T_n((A, a), (B, b))$ be a round of the game for temporal logic and i and j two moves in the game, such that $i < j$ (i precedes j in T_n). If at least one of:*

1. *i and j are data moves,*
2. *i is a data move and j is a temporal move,*

$$
\begin{array}{l|l}
\text{Round of the game:} & A \quad t_0\ a_1\ a_2\ t_3\ t_4\ a_5\ a_6\ t_7\ a_8\ \dots\ t_i\ \dots\ a_n \\
 & B \quad s_0\ b_1\ b_2\ s_3\ s_4\ b_5\ b_6\ s_7\ b_8\ \dots\ s_i\ \dots\ b_n \\
\hline
\text{Compatible moves:} & 0 \\
 & 3 \\
 & 4 \\
 & 7 \\
 & \vdots \qquad\qquad\qquad \vdots \\
 & i \\
 & \vdots \qquad\qquad\qquad \vdots
\end{array}
$$

Figure 3.2: A round of the game and the compatibility of the individual temporal moves.

3. *i is the last temporal move preceding the data move j,*

4. *i and j are both temporal moves, and all the moves between i and j are temporal moves,*

5. *i and j are both temporal moves, and i is the last temporal move preceding j such that there is a data move between i and j*

holds, then we say that the moves i and j are compatible. *A set of moves S is* self-compatible *if every two elements in S are compatible.*

This definition is related to the definition of the temporal connectives and Example 3.19. The blocks of consecutive temporal moves in T_n correspond to a temporal connective or several subsequent temporal connectives with the same number of nested quantifiers over T. The compatibility criterion exactly matches the compatibility of the scopes of the corresponding variables in a formula of $\mathcal{T}$:

Example 3.23 Figure 3.2 shows a round of the game $T_n((A, t_0), (B, s_0))$ on the structures A and B. The lower part of the figure shows moves compatible with the individual temporal moves in this particular round of the game. It is easy to see that compatibility of moves in the game exactly corresponds to scopes of variables in $\mathcal{T}$-formulas. Note that in the case of 2-FOL, the scopes of the individual variables would extend to the rightmost edge of the figure—the n-th column.

Definition 3.22 implies the following restrictions on the winning condition of the game for temporal logic:

Definition 3.24 *Let $a = (a_1, \dots, a_k)$ for $a_i \in A$, $b = (b_1, \dots, b_k)$ for $b_i \in B$, and $T_n((A, a), (B, b))$ be a round of the game for FOTL of length n. We say that* Player II wins the round of the game for FOTL *if the following conditions hold:*

1. *for any $p_i \in \delta$ and any $k_1, \dots, k_{u_i}$ such that $0 < k_j \le n + k$ where all k_j are data moves such that $\{k_1, \dots, k_{u_i}\}$ is self-compatible*

$$A \models p_i^A(a_{k_1}, \dots, a_{k_{u_i}}) \iff B \models p_i^B(b_{k_1}, \dots, b_{k_{u_i}})$$

2. *for any* $\rho_i \in \gamma$ *and any* $k_1, \ldots, k_{w_i}$ *s.t.* $0 < k_j \leq n+k$ *where all* k_j *are temporal moves such that* $\{k_1, \ldots, k_{w_i}\}$ *is self-compatible*

$$A \models \rho_i^A(a_{k_1}, \ldots, a_{k_{w_i}}) \iff B \models \rho_i^B(b_{k_1}, \ldots, b_{k_{w_i}})$$

3. *for any temporal extension* R_i *of* $r_i \in \sigma$ *and any* $k_0, \ldots, k_{v_i}$ *s.t.* $0 < k_j \leq n + k$ *such that* k_0 *is a temporal move where all* k_j *such that* $0 < j \leq v_i$ *are data moves and* $\{k_0, \ldots, k_{v_i}\}$ *is self-compatible*

$$A \models R_i^A(a_{k_0}, \ldots, a_{k_{v_i}}) \iff B \models R_i^B(b_{k_0}, \ldots, b_{k_{v_i}})$$

Otherwise we say that Player I wins the round of the game for temporal logic.

If Player II *can always win the game* $T_n((A, a), (B, b))$ *we say that* Player II *has a* winning strategy for $T_n((A, a), (B, b))$.

It is easy to verify that a winning strategy for Player II defines an equivalence relation $\sim_{k,n}$ on the class of τ-structures, where k is the length of the initial assignment and n is the number of moves in the game T_n.

Definition 3.25 (Quantifier Depth) *We define function* $\mathrm{qd} : \mathcal{M} \to N$

1. *If* φ *is atomic then* $\mathrm{qd}(\varphi) = 0$.
2. *If* φ *is* $\neg\phi$ *then* $\mathrm{qd}(\varphi) = \mathrm{qd}(\phi)$.
3. *If* φ *is* $\phi_1 \wedge \phi_2$ *then* $\mathrm{qd}(\varphi) = \max\{\mathrm{qd}(\phi_1), \mathrm{qd}(\phi_2)\}$.
4. *If* φ *is* $\exists x.\phi$ *or* $\exists t.\phi$ *then* $\mathrm{qd}(\varphi) = \mathrm{qd}(\phi) + 1$.

Definition 3.26 *Let* $\mathcal{L}_{k,n}$ *be set of all formulas* φ *of* $\mathcal{M}$ *such that* φ *is a subformula of* $\mathrm{Embed}(\psi)$ *for some* $\psi \in \mathcal{T}$, $|FV(\varphi| = n$, *and* $\mathrm{qd}(\varphi) = n$.

In general the sets $\mathcal{L}_{k,n}$ are countably infinite, but we can always find a finite subset of this set that "represents" all the formulas in $\mathcal{L}_{k,n}$:

Lemma 3.27 *For each* $k, n \in N$ *there is a finite subset* $\Phi_{k,n}$ *of* $\mathcal{L}_{k,n}$ *such that every formula* $\phi \in \mathcal{L}_{k,n}$ *is equivalent to a formula* $\varphi \in \Phi_{k,n}$. *Moreover, all free variables in* φ *are* $\{v_1, \ldots, v_k\}$.
P r o o f: By induction on n.

Notation 3.28 *Let* $\varphi \in \Phi_{k,n}$ *be a formula and* $a = (a_1, \ldots, a_k)$ *a vector of elements from the carrier of a first-order structure* A. *Then we write*

$$A; a \models \varphi \text{ for } A \models \varphi[v_1/a_1, \ldots, v_k/a_k].$$

Note that the variables v_i *are the only free variables of* φ.

Now we can show the relationship between the initial assignments and the sets $\Phi_{k,n}$:

Example 3.29 Consider a $\mathcal{T}$-formula ψ, e.g.:

$\Diamond\exists x.\Diamond\exists y.p(x,y)$	a FOTL formula ψ
$\exists t_1 > t_0.\exists x.\exists t_2 > t_1.\exists y.P(t_2,x,y)$	the translation Embed(ψ)
$\exists v_2 > v_1.\exists v_3\exists v_4 > v_2.\exists v_5.P(v_4,v_3,v_5)$	the translation after renaming of variables

The renaming of variables allows us to figure out, what set $\Phi_{k,n}$ a particular subformula of Embed(ψ) belongs to:

$$\begin{aligned} \exists v_2 > v_1.\exists v_3.\exists v_4 > v_2.\exists v_5.P(v_4,v_3,v_5) &\in \Phi_{1,4} \\ \exists v_3.\exists v_4 > v_2.\exists v_5.P(v_4,v_3,v_5) &\in \Phi_{2,3} \\ \exists v_4 > v_2.\exists v_5.P(v_4,v_3,v_5) &\in \Phi_{3,2} \\ \exists v_5.P(v_4,v_3,v_5) &\in \Phi_{4,1} \\ P(v_4,v_3,v_5) &\in \Phi_{5,0} \end{aligned}$$

The sorts of the individual variables v_i correspond to the sorts of the elements in the initial assignment that is used to show the non-definability by this particular formula. Note that for, e.g., $\exists v_5.P(v_4,v_3,v_5) \in \Phi_{4,1}$ the elements of the initial assignment must have the sorts T, D, T, D, respectively. However only the last two of them are used to verify the winning condition with respect to $P(v_4,v_3,v_5) \in \Phi_{5,0}$ (which is atomic).

This idea is used in the next theorem that shows the equivalence between the first-order temporal formulas and the games for temporal logic.

Theorem 3.30 *Let A and B be τ-structures, $a = (a_1,\ldots,a_k)$ and $b = (b_1,\ldots,b_k)$ be sequences of A-elements and B-elements (respectively) such that $a_1 \in T^A$ and $b_1 \in T^B$, and $n \in N$. Then*

$$(A,a) \sim_{k,n} (B,b) \iff \forall\varphi \in \Phi_{k,n} : A; a \models \varphi \iff B; b \models \varphi$$

P r o o f: By induction on n for all k:

Base: If $n = 0$, then $(A,a) \sim_{k,0} (B,b)$ iff $\forall\varphi \in \Phi_{k,0} : A; a \models \varphi \iff B; b \models \varphi$ by Definition 3.26 because the only $\varphi \in \Phi_{k,0}$ are atomic formulas or negations and conjunctions of those. Note that all the atomic formulas preserve the compatibility criterion, as otherwise they would not be subformulas of Embed(θ) for $\theta \in \mathcal{L}$.

Induction step: Assume that the claim has been proven for n and suppose that

$$(A,a) \sim_{k,n+1} (B,b).$$

Let $\varphi \in \Phi_{k,n+1}$ be an arbitrary formula. We can assume that φ is of the form $\exists t.\psi$ or $\exists x.\psi$ where $\psi \in \Phi_{k+1,n}$ (the correspondence for boolean combinations of such formulas is determined by the result for the formulas themselves).

In both cases, since $(A,a) \sim_{k,n+1} (B,b)$ we know that for each a_{k+1} there is a b_{k+1} (of the same sort) such that $(A,a,a_{k+1}) \sim_{k,n} (B,b,b_{k+1})$. By induction hypothesis, for each $\psi \in \Phi_{k,n+1}$

$$A; a, a_{k+1} \models \psi \iff B; b, b_{k+1} \models \psi$$

Thus we can pick $\varphi = \exists v_{k+1}.\psi$ (with the quantifier of the appropriate sort) and we have $A; a \models \varphi \iff B; b \models \varphi$. This covers all formulas in the set $\Phi_{k,n+1}$ and thus proves the claim. Moreover, all the leaves of ψ have to obey the variable compatibility restrictions. This guarantees that ψ is a subformula of Embed(θ) for some $\theta \in \mathcal{L}$.

Conversely, suppose that $(A; a) \not\sim_{k,n+1} (B; b)$. Then there exists an element a_{k+1} such that for all b_{k+1}

$$(A; a, a_{k+1}) \not\sim_{k+1,n} (B; b, b_{k+1})$$

By induction hypothesis for each b_{k+1} (that satisfies the rules of the game) we have formula $\psi_{b_{k+1}} \in \Phi_{k+1,n}$ such that

$$A; a, a_{k+1} \models \psi_{b_{k+1}} \text{ and } B; b, b_{k+1} \not\models \psi_{b_{k+1}}$$

Since $\Phi_{k+1,n}$ is finite by Lemma 3.27 we can form $\phi = \bigwedge_{b_{k+1}} \psi_{b_{k+1}}$. Then

$$A; a, a_{k+1} \models \phi \text{ and } B; b, b_{k+1} \not\models \phi$$

for all b_{k+1} allowed by rules of the game.

Similarly to the previous case we pick $\varphi = \exists v_{k+1}.\phi$. Thus, we have constructed $\varphi \in \Phi_{k,n+1}$ such that $A; a \models \varphi$ and $B; b \not\models \varphi$.

Note the crucial use of the restriction on the scopes of the individual variable names in both the base case of the induction and the inductive step: the restriction on the compatibility of the variables from Example 3.19 and Definition 3.22 holds for all subformulas of a given temporal formula, especially for the atomic formulas.

Note also that if $(A, a) \sim_{1,n} (B, b)$ then the τ-structures A and B are indistinguishable by any closed $\mathcal{T}$-formulas of quantifier depth at most n over an arbitrary set of temporal connectives built using Definition 3.6 for a given initial temporal context a, b.

Now we can use a procedure similar to Corollary 3.16 to show the non-definability in FOTL. However, we need to be careful here: to show the separation of the languages L and $\mathcal{M}$, we need to show non-definability of a first-order definable property.

3.5 Structures indistinguishable by FOTL formulas

We prepared all the necessary techniques to show our main result: the language $\mathcal{T}$ is strictly less expressive than the language $\mathcal{M}$. We show that for a standard choice of the data domain being the set of uninterpreted constants with equality and a standard choice of a linearly ordered temporal domain:

- the propositional temporal logic has an expressively complete set of temporal connectives [39, 50, 56, 88], but
- there is no finite set of temporal connectives expressively complete in the first-order case.

This result implies that the choice of FOTL with arbitrary finite set of connectives as a basis of a *first-order complete* query language for temporal databases is not adequate, as there are first-order queries not expressible in this language. We prove this result in two steps:

1. First we show that the FOTL language $\mathcal{L}$ is strictly less expressive than the 2-FOL language $\mathcal{M}$ for a degenerate choice of the temporal domain: a set of elements *without* any structure on the elements, even equality (i.e., the signature of the temporal domain is empty) and the standard data domain of uninterpreted constants with equality.

2. In the second step we extend this result to all dense linear orders.

3.5.1 Time Domain: a Set

The first claim is proven as follows: Let ϵ be an empty signature, $\Diamond_\epsilon X_1 \equiv \exists t_1.X_1$ be a temporal connective, and $\mathcal{T}(\Omega(\epsilon), =)$ be an FOTL language.

Lemma 3.31 *The propositional TL $\mathcal{T}(\{\Diamond_\epsilon\})$ is expressively complete with respect to the monadic 2-FOL $\mathcal{M}(\epsilon)$.*

P r o o f: By induction on the structure of $\varphi \in \mathcal{M}(\epsilon)$.

Note that in the propositional case the signature of the data domain is irrelevant.

Notation 3.32 *Let $\mathcal{S}_n^m = \{S_1, \ldots, S_k\}$ be the set of all n-element subsets of the m-element set $\{1, \ldots, m\}$.*

We prove that Lemma 3.31 does not generalize to the first-order case: the logic $\mathcal{T}(\Omega(\epsilon); =)$ is strictly less expressive than the logic $\mathcal{M}(\epsilon; =)$:

Theorem 3.33 *Let $m, n \in N$ such that $m > n$. We define*

$$P_n^m = \{(t, x) : x \in S_t \in \mathcal{S}_n^m\} \text{ and } R_n^m = \{(-t, x) : x \in S_t \in \mathcal{S}_n^m\}.$$

Let

$$\mathcal{A}_n = \langle Z; \{1, \ldots, 2n\}, =; P_{n-1}^{2n}, R_n^{2n}\rangle \qquad \mathcal{B}_n = \langle Z; \{1, \ldots, 2n\}, =; P_n^{2n}, R_n^{2n}\rangle$$

be two temporal structures (with the signature $\langle \epsilon; =; p/1, r/1\rangle$). Then

1. *$\mathcal{A}_n$ and $\mathcal{B}_n$ can be distinguished by a $\mathcal{M}$ formula (of quantifier depth 3), but*

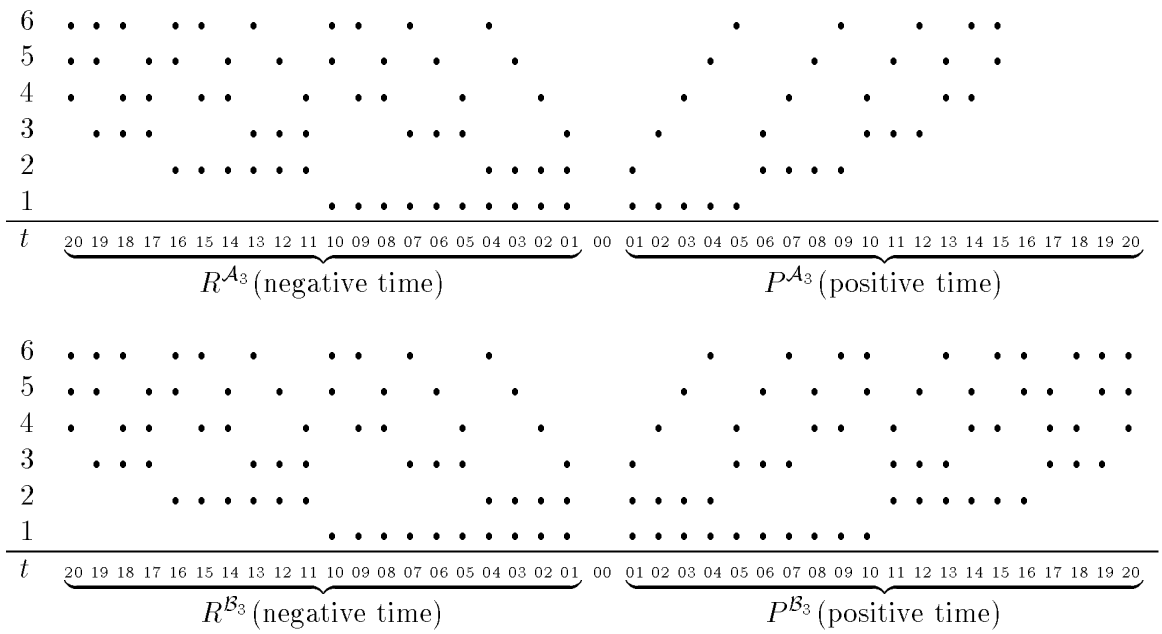
2. *$\mathcal{A}_n$ and $\mathcal{B}_n$ cannot be distinguished by any $\mathcal{T}$ formula of quantifier depth less or equal to $n-1$.*

P r o o f: (1) Let

$$\varphi \equiv \exists t_1.\exists t_2.((\exists x.P(t_1, x)) \wedge (\exists x.R(t_2, x)) \wedge (\forall x.P(t_1, x) \iff R(t_2, x)))$$

Clearly $\varphi \in \mathcal{M}(\epsilon, =)$, $\varphi \not\models \mathcal{A}_n$, and $\varphi \models \mathcal{B}_n$. To prove (2) we define a winning strategy for Player II: Let $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$ be all the data moves and (t, s) the last temporal move in a prefix of a round of the game T_n that satisfies the winning condition. We show that every sequence of moves, shorter than n can be extended by one move. By case analysis (assuming Player I picks an element from the carrier of the structure $\mathcal{A}_n$):

1. Player I plays a data move $a \in \{1, \ldots, 2n\}$ and $a = a_i$ for some $0 < i \leq k$. Then Player II responds by $b = b_i$.



The interpretation of the relation symbols P and R in the structures $\mathcal{A}_3$ and $\mathcal{B}_3$, respectively. A dot means that the data element on the same line is in the appropriate relation with the time instant in the dot's column).

Figure 3.3: Interpretation of P and R in the structures $\mathcal{A}_3$ and $\mathcal{B}_3$.

2. Player I plays a data move $a \in \{1, \ldots, 2n\}$ and $a \neq a_i$ for all $0 < i \leq k$. Then Player II plays $b \in \{1, \ldots, 2n\}$ such that $b \neq b_i$ for $0 < i \leq k$ and if $(t, a) \in P^{2n}_{n-1}$ then $(s, b) \in P^{2n}_n$, otherwise $(s, b) \notin P^{2n}_n$. Similarly, if $(t, a) \in R^{2n}_n$ then $(s, b) \in R^{2n}_n$, otherwise $(s, b) \notin P^{2n}_n$. In all cases Player II can find such a b as there are only $k < n$ elements chosen so far.

3. Player I plays a temporal move $t' \in \{1, \ldots, |\mathcal{S}^{2n}_{n-1}|\}$. Then Player II plays $s' \in \{1, \ldots, |\mathcal{S}^{2n}_n|\}$ such that $(t', a_i) \in P^{2n}_{n-1}$ if and only if $(s', b_i) \in P^{2n}_n$ for all $0 < i \leq k$. Again, such a s' exists because $\mathcal{S}^{2n}_n$ contains all n-element subsets of $\{1, \ldots, 2n\}$ and the number of data moves played so far is less than n. The choice may not be unique—in that case Player II can choose arbitrarily as all the elements of T are indistinguishable in the signature ϵ.

4. Player I plays $t' \in \{-1, \ldots, -|\mathcal{S}^{2n}_n|\}$. Then Player II plays $s' \in \{-1, \ldots, -|\mathcal{S}^{2n}_n|\}$ such that $(t', a_i) \in R^{2n}_n$ if and only if $(s', b_i) \in R^{2n}_n$ for all $0 < i \leq k$. Similarly to the previous case, such a s' always exists.

Similarly, if Player I chooses an element from the carrier of the structure $\mathcal{B}_n$, Player II replies with an element from $\mathcal{A}_n$ using symmetric strategy. In all cases the new move does not violate the winning condition.

The conclusion of this theorem follows from Theorem 3.30.

Example of a 3-move game T_3: For the game on the structures $\mathcal{A}_3$ and $\mathcal{B}_3$ we need to distinguish 3 cases (depending on how many data moves have been played so far):

1. No data moves: this is very easy case, as the PLAYER II can reply almost arbitrarily to any temporal move of PLAYER I. The only restriction is that he has to play a "nonempty" state if and only if PLAYER I plays an "nonempty" state (this guarantees that he is able to find an answer to any data move as well). If PLAYER I plays a data move, PLAYER II plays also a data move: assume that in this move PLAYER I picked data element 1 in $\mathcal{A}_3$ and the game so far looks like follows:

$$\begin{array}{lll} \dots & 5_T & 1_D \\ \dots & 8_T & x \end{array}$$

   Then PLAYER II looks in the 5-th state of $\mathcal{A}_3$ and finds that $(5,1) \in P^{\mathcal{A}_3}$. Thus he picks an element x such that $(8,x) \in P^{\mathcal{B}_3}$ arbitrarily, e.g., $x = 4$.

2. One data move: in this case PLAYER II has to maintain the similarity of the temporal states: his reply to any temporal move of the PLAYER I has to respect the partial mapping $1 \mapsto 5$: assume PLAYER I plays a temporal move -11 in $\mathcal{A}_3$. Then PLAYER II has to answer with, e.g., -10 in $\mathcal{B}_3$ as $(-11,1) \notin R^{\mathcal{A}_3}$ and $(-10,4) \notin R^{\mathcal{B}_3}$. The reply for a data move is similar to the previous case. Assume that PLAYER I plays $2 \in \mathcal{B}_3$:

$$\begin{array}{llllll} \dots & 5_T & 1_D & \dots & -11_T & x \\ \dots & 5_T & 4_D & \dots & -10_T & 2_D \end{array}$$

   PLAYER II picks $x = 5$ as both 2 and 5 are not in the appropriate relations R at the times -11 and -10 (cf. figure 3.3).

3. after two data moves: the strategy for finding answers to the temporal moves of PLAYER I is the same—we just have to look how both the two played data elements interact with the temporal moves of PLAYER I (in the previous case we had to watch just for one element). This is not a problem, as the relations contain all possible subsets of size at least 2. However, there will be no more data moves in the T_3 game.

Note that the number of temporal moves does not matter: as long as there are at most 2 data moves, the PLAYER II can maintain the winning strategy for any number of temporal moves.

Figure 3.4: A round of the game for Temporal Logic for $\mathcal{A}_3$ and $\mathcal{B}_3$.

Corollary 3.34 *FOTL $\mathcal{T}(\Omega(\epsilon), =)$ is strictly less expressive than 2-FOL $\mathcal{M}(\epsilon, =)$ for an arbitrary finite set of connectives $\Omega(\epsilon)$.*

Theorem 3.33 and Corollary 3.34 also provide an example of when expressive completeness in the propositional case does not imply the existence of a complete set of connectives in the first-order case.

3.5.2 Time Domain: Dense Linear Order

To show a similar result for temporal logic over linear orders is more complicated. We need to account for the structure of the temporal domain. Consider the following definition:

Definition 3.35 *Let $(Q, <)$ be a linear order, S a finite set, $I \subseteq Q$ a non-trivial interval, and $f : I \to S$ a function. We say that f is* dense over S *if for every $s, t \in I$ such that $s < t$ and for every $x \in S$ there is $r \in I$ such that $s < r < t$ and $x = f(r)$.*

Clearly, for any dense linear order we can define functions dense over any finite codomain.

Lemma 3.36 *Let $(Q, <)$ be a dense linear order and $I \subseteq Q$ an open interval such that $|I| > 1$. Then for every finite set S there is a $f : I \longrightarrow S$ that is dense over S.*
P r o o f: Immediate from density of $(Q, <)$.

Figure 3.5 shows an example of a function that is dense over a five element set.

Theorem 3.37 *Let $(Q, <)$ be a dense linear order, $f^I_{m,n} : I \longrightarrow \mathcal{S}^m_n \cup \{\emptyset\}$ dense over $\mathcal{S}^m_n \cup \{\emptyset\}$ for every non-trivial interval $I \subset Q$, and let J_1 and J_2 be two nontrivial, open, and disjoint intervals in Q. We define*

$$P^m_n = \{(t, x) : x \in f^{J_1}_{m,n}(t)\} \quad \text{and} \quad R^m_n = \{(t, x) : x \in f^{J_2}_{m,n}(t)\}.$$

Let

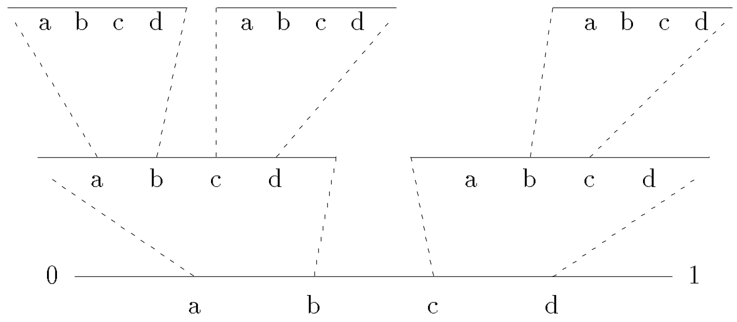
$$\mathcal{A}_n = \langle Q, <; \{1, \ldots, 2n\}, =; P^{2n}_{n-1}, R^{2n}_n\rangle \quad \text{and} \quad \mathcal{B}_n = \langle Q, <; \{1, \ldots, 2n\}, =; P^{2n}_n, R^{2n}_n\rangle$$

be two temporal databases. Then

1. *$\mathcal{A}_n$ and $\mathcal{B}_n$ can be distinguished by a $\mathcal{M}$ formula (of quantifier depth 3), but*
2. *$\mathcal{A}_n$ and $\mathcal{B}_n$ cannot be distinguished by any $\mathcal{T}$ formula of quantifier depth less than or equal to $n - 1$.*

P r o o f: (1) immediate from the proof of Theorem 3.33 (1), (2) follows from the proof of Theorem 3.33 (2) and observation that $f^I_{m,n}$ is dense over $\mathcal{S}^m_n$ and thus PLAYER II can find an appropriate answer to every temporal move of the PLAYER I that respects the compatible temporal moves, played so far (which follows immediately from density of $f^I_{m,n}$).

Corollary 3.38 *FOTL $\mathcal{T}(\Omega(<), =)$ over a dense linear order $<$ is strictly less expressive than 2-FOL $\mathcal{M}(<, =)$ for an arbitrary finite set of connectives $\Omega(<)$.*



The open unit interval $]0,1[$ is split to 5 sub-intervals of equal length. Every inner boundary is mapped to one of the letters in $\{a,b,c,d\}$ (in alphabetical order). Then the same process is applied on all the subintervals. Thus

$$\begin{array}{lcl}
\frac{1}{5},\frac{1}{25},\frac{6}{25},\ldots,\frac{21}{25},\frac{1}{125},\ldots,\frac{121}{125},\frac{1}{625},\ldots,\frac{621}{625},\frac{1}{3125},\ldots,\frac{3121}{3125},\ldots & \mapsto & a\\
\frac{2}{5},\frac{2}{25},\frac{7}{25},\ldots,\frac{22}{25},\frac{2}{125},\ldots,\frac{122}{125},\frac{2}{625},\ldots,\frac{622}{625},\frac{2}{3125},\ldots,\frac{3122}{3125},\ldots & \mapsto & b\\
\frac{3}{5},\frac{3}{25},\frac{8}{25},\ldots,\frac{23}{25},\frac{3}{125},\ldots,\frac{123}{125},\frac{3}{625},\ldots,\frac{623}{625},\frac{3}{3125},\ldots,\frac{3123}{3125},\ldots & \mapsto & c\\
\frac{4}{5},\frac{4}{25},\frac{9}{25},\ldots,\frac{24}{25},\frac{4}{125},\ldots,\frac{124}{125},\frac{4}{625},\ldots,\frac{624}{625},\frac{4}{3125},\ldots,\frac{3124}{3125},\ldots & \mapsto & d
\end{array}$$

The remaining elements of $]0,1[$ are mapped to the remaining element e in the range of f.

Figure 3.5: Dense function $f:]0,1[\longrightarrow\{a,b,c,d,e\}$.

3.6 Additional Results

Using our technique we can prove several additional results about FOTL. Among other results, we show that the gap between FOTL and 2-FOL cannot be bridged using a more powerful version of FOTL—the weakness is inherent to the restriction on the maximal temporal arity (dimension) of the logic FOTL.

3.6.1 Separation of Future FOTL from FOTL

Using a similar technique as in Theorem 3.37 we can separate the *Future* FOTL with arbitrary future temporal connectives from the *full* FOTL $\mathcal{T}(\mathbf{since}, \mathbf{until})$. Note that these two logics are expressively equivalent in the propositional case [41].

Lemma 3.39 *Let* $(Q^+; <, 0)$ *be a dense linear order with a left endpoint* (0). *Let* $J \subset Q^+$ *be a non-trivial open interval and* $f^I_{m,n} : I \longrightarrow \mathcal{S}^m_n \cup \{\emptyset\}$ *be dense over* $\mathcal{S}^m_n$. *We define* $P^m_n = \{(t, x) : x \in f^J_{m,n}(t)\}$ *and* $R_n = \{(0, x) : 0 < x \leq n\}$. *Let*

$$\mathcal{A}_n = \langle Q, <; \{1, \ldots, 2n\}, =; P^{2n}_{n-1}, R_n\rangle \qquad \mathcal{B}_n = \langle Q, <; \{1, \ldots, 2n\}, =; P^{2n}_{n}, R_n\rangle$$

be two temporal structures. Then

1. *The structures* $\mathcal{A}_n$ *and* $\mathcal{B}_n$ *can be distinguished by a* $\mathcal{T}(\mathbf{since}, \mathbf{until})$ *formula of quantifier depth 3, but*

2. $\mathcal{A}_n$ *and* $\mathcal{B}_n$ *cannot be distinguished by any future* $\mathcal{T}$ *formula of quantifier depth less or equal to* $n - 1$.

P r o o f: (1) Let

$$\varphi = (\Diamond \forall x.(p(x) \iff \blacklozenge r(x))) \in \mathcal{T}(\mathbf{since}, \mathbf{until})$$

Clearly, $\varphi, 0 \not\models \mathcal{A}_n$ and $\varphi, 0 \models \mathcal{B}_n$.

(2) follows from observation that $f^I_{m,n}$ is dense over $\mathcal{S}^m_n$ (and thus Player II can find an appropriate answer to every temporal move of the Player I), and the fact, that the quantifiers in future FOTL formulas are restricted (this means that Player I cannot return to *time* 0 once he played any other temporal move, as the sequence of temporal moves in a game for Future FOTL has to be non-decreasing sequence of elements of T).

3.6.2 Expressive Incompleteness for Many-dimensional FOTL

When using FOTL as a query language, we cannot express queries that reference two distinct time instants (in an essential way, cf. Section 3.5). This is due to the limitation on the number of temporal contexts passed to the individual subformulas of a FOTL formula. In [42] many-dimensional propositional temporal logics are studied. These logics can be easily modified to many-dimensional first-order temporal logics (k-FOTL, where k is the dimension of the logic). The main idea behind this approach is to allow k *temporal contexts* in all the temporal (sub-)formulas. Clearly, 2-FOTL can express the query from Theorem 3.33 (2) as exactly two temporal contexts are needed in this query. However, using our technique we can show that k-FOTL is strictly less expressive than $(k+1)$-FOTL and thus is also less expressive than 2-FOL for any $k > 1$.

Definition 3.40 (Many-dimensional Temporal Connective) *Let γ be a signature of the temporal domain, $k > 0$, and t_i be variables ranging over k-tuples of elements of T (time instants). Then*

$$O ::= \rho(t_1[j_1], \ldots, t_{w_k}[j_k]) \mid O \wedge O \mid \neg O \mid \exists t_i.O \mid X_i$$

where X_i are predicate variables, $\rho \in \gamma$, and $t_i[j]$ is the j-th component of the tuple variable t_i. A (n-ary) k-dimensional temporal connective (over γ) is an O-formula with exactly one free variable t_0 and n free predicate variables $X_1, \ldots, X_n$. We denote it by

$$\omega(X_1, \ldots, X_n)$$

Let $\Omega(\gamma)$ denote a finite set of (names for) temporal connectives over γ.

Now we can define the language of k-FOTL exactly the same way as in the case of FOTL (cf. Definition 3.9). The semantics is also defined in the same way as in Definition 3.11 with a small "fix" to the base case:

$$\text{Embed}(r_i(x_1, \ldots, x_{v_i})) = R_i(t_0[1], x_1, \ldots, x_{v_i}).$$

The game for k-dimensional temporal logic is similar to the game for FOTL. The only difference is that for every temporal move we always pick k-tuples of elements of sort T instead of a single element. We have to guarantee that the winning condition is met for all *components* of every tuple move.

Note that we are still using the uniform temporal databases with relations monadic in the sort T. Clearly, a higher dimension of the base relations in the temporal sort leads to the separation result immediately. However, even with the restriction on the database relations we can prove the separation of the k-FOTL from $(k+1)$-FOTL.

To show the equivalence of two structures with respect to the game for k-dimensional temporal logic, consider the following definition:

Definition 3.41 *Let $S_1, \ldots, S_k$ be subsets of $\{1, \ldots, n\}$. A* cover of $\{1, \ldots, n\}$ with respect to *$S_1, \ldots, S_k$ is a collection of equivalence classes A_i generated by the relation $\asymp$ on $\{1, \ldots, n\}$ defined as*

$$x \asymp y \iff \forall i \in \{1, \ldots, k\}.(x, y \in S_i \vee x, y \notin S_i).$$

Let A_i and B_i be elements in the covers of $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, respectively. We say that A_i corresponds to *B_i if*

$$A_i \subseteq S_j \iff B_i \subseteq T_j \text{ for all } 0 < j \leq k.$$

The relation $\asymp$ is obviously an equivalence relation, as it is reflexive, symmetric, and transitive. We use sets $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$ as the snapshots of the temporal structures with respect to the last temporal move in the game. The following definition shows, how the covers generated from these sets influence the winning condition of the game for many-dimensional FOTL:

Definition 3.42 *Let $\{A_i\}$ and $\{B_i\}$ be covers of $\{1, \ldots, n\}$ with respect to $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, respectively, and $(a_1, b_1), \ldots, (a_l, b_l)$ pairs of elements of $\{1, \ldots, n\}$. We say that $S_1, \ldots, S_k$* is similar *to $T_1, \ldots, T_k$* with respect to *$(a_1, b_1), \ldots, (a_l, b_l)$ if*

$$a_i \in A_j \iff b_i \in B_j$$

for all $0 < i \leq l$ and all pairs A_j, B_j.

Essentially, the moves in the game have to match with the covers, generated by the sets $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$ (and vice versa). Moreover, we have to ensure that the sequence of data moves can be extended up to the maximal number of moves in the particular round of the game. Thus we have to make sure that if an element A_i in the cover of $S_1, \ldots, S_k$ has cardinality less or equal to the length of the game, then the corresponding element B_i in the cover of $T_1, \ldots, T_k$ has exactly the same cardinality and thus they cannot be distinguished by the data moves.

We build the pairs of structures $(\mathcal{A}_n, \mathcal{B}_n)$ similarly to the way they were built in Theorems 3.33 and 3.37:

Definition 3.43 *Let $k, n > 0$ be natural numbers and $N = 2^k(n + k)$. We define*

$S_P^{n,k}$ to be the set of all subsets of cardinality N,

$S_{R(A)}^{n,k}$ the set of all subsets of cardinality at least kN, and

$S_{R(B)}^{n,k}$ the set of all subsets of cardinality bigger than kN

of the set $\{1, \ldots, 2(k + 1)N\}$.

We use these sets similarly to the way they were used in Theorems 3.33 and 3.37 to represent the contents of the relations P and R at distinct time instants. The following lemma shows that whatever k-tuple of time instants Player I chooses as his temporal move, Player II can always find a *similar* k-tuple in the other structure:

Lemma 3.44 *Let $(a_1, b_1), \ldots, (a_l, b_l)$ be pairs of elements of $\{1, \ldots, 2(k + 1)N\}$ pairwise different in both projections for every $l \leq n$. Then for every $S_1, \ldots, S_k$ from $S_P^{n,k} \cup S_{R(A)}^{n,k}$ there are $T_1, \ldots, T_k$ from $S_P^{n,k} \cup S_{R(B)}^{n,k}$ (and vice versa) such that:*

1. *$S_1, \ldots, S_k$ is similar to $T_1, \ldots, T_k$ with respect to $(a_1, b_1), \ldots, (a_l, b_l)$,*
2. *$|A_i| = |B_i|$ or $|A_i|, |B_i| \geq n$ for all pairs of corresponding elements in the covers of $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$, and*
3. *$S_i \in S_P^{n,k} \iff T_i \in S_P^{n,k}$.*

P r o o f: Let $B_i^0 = \{b_j : a_j \in A_i\}$. Clearly $|B_i^0| \leq |A_i|$. To satisfy condition (1) it is sufficient that every element B_i in the cover generated from $T_1, \ldots, T_k$, contains the corresponding set B_i^0. This can be easily achieved as follows: let f be a bijection from $\{1, \ldots, 2(k+1)N\}$ to $\{1, \ldots, 2(k + 1)N\}$, such that $f(a_i) = b_i$ for all $0 < i \leq l$. We need to modify those T_j''s with cardinality kN as they are not members of $S_P^{n,k} \cup S_{R(B)}^{n,k}$:

1. $S_i \in S_P^{n,k}$ for all $0 < i \leq k$. Then there is nothing to do as we let $T_i = T_i' \in S_P^{n,k}$ and clearly no T_j' has cardinality kN.

2. $S_i \in S_{R(A)}^{n,k}$ for all $0 < i \leq k$. Then for every $0 < i \leq k$ there is A_j in the cover of $S_1, \ldots, S_k$ such that $|A_j| \geq n + k$ (as $|S_i| > (n+k)2^k$). This is especially true for the sets S_j with cardinality equal to kN. Moreover, there is an element A in the cover such that $|A| > n+k$ and $A \cap S_i = \emptyset$ for all S_i with cardinality less than $kN + k$. We *move* one element of A to the set A_j in S_j. The new collection of $f(A_i)$ is a cover for $T_1, \ldots, T_k$ that satisfies all the conditions of the theorem.

3. There are S_i both from $S_P^{n,k}$ and $S_{R(A)}^{n,k}$. We proceed similarly to (2) with the distinction that we consider only those A_i's that have empty intersection with all the $S_i \in S_P^{n,k}$. Note that in every $S_j \in S_{R(A)}^{n,k}$ there still is A_i such that $|A_i| > n + k$ and $A_i \cap S_j = \emptyset$ for all $S_j \in S_P^{n,k}$. Then we use similar procedure as in (2) to produce the cover of $T_1, \ldots, T_k$.

The opposite direction is trivial as $S_{R(B)}^{n,k} \subset S_{R(A)}^{n,k}$ and thus the identity mapping of the covers satisfies the requirements of the lemma immediately.

The consequence of this lemma is that not only Player II can find an answer to every temporal move of Player I that is compatible with all the data moves played so far but also that the sequence of data moves can be always extended to a sequence of length n that satisfies the winning condition. The ordering of temporal moves is handled similarly to the ordering in Theorems 3.33 or 3.37, depending on the signature of the temporal domain. All together:

Theorem 3.45 *Let*

$$\varphi := \exists t.\exists t_1, \ldots, t_k.\forall x \left(R(t,x) \iff \bigvee_{0<i\leq k} P(t_i, x) \right)$$

Then φ is expressible in $(k+1)$-FOTL but not in k-FOTL.

Proof: That φ is not expressible by k-FOTL follows immediately from Lemma 3.44 and Theorems 3.33 or 3.37 by constructing structures $\mathcal{A}_n, \mathcal{B}_n$ using the sets $S_P^{n,k}$ for the relation P (in both structures) and the sets $S_{R(A)}^{n,k}$ and $S_{R(B)}^{n,k}$ to construct the relation R in the structures $\mathcal{A}_n$ and $\mathcal{B}_n$, respectively.

To show that φ is expressible in $(k+1)$-FOTL, consider following $(k+1)$-dimensional connectives:

$$\begin{aligned} \diamond \quad &= \exists t_1.X_1 \\ \uparrow\downarrow_i &= \exists t_1.(t_1[1] = t_0[i] \wedge X_1) \end{aligned}$$

Note that the variables t_i range over $(k+1)$-tuples of temporal elements. Then

$$\varphi \text{ is equivalent to } \diamond\forall x \left(r(x) \iff \bigvee_{0<i\leq k} \uparrow\downarrow_i p(x) \right)$$

which is a $(k+1)$-FOTL formula.

Thus the many-dimensional FOTL logics form a proper hierarchy with respect to their relative expressive power:

$$\text{FOTL} \subset \text{2-FOTL} \subset \ldots \subset (k-1)\text{-FOTL} \subset k\text{-FOTL} \subset \ldots \subset \text{2-FOL}$$

Note that for an arbitrary finite set of connectives we can find a k-FOTL that can express all the connectives for any $k \in N$. Also, from the previous results and [57], we know that

$$\text{Future-FOTL} \subset \text{FOTL} \subset \text{FOTL(now)} \subseteq \text{2-FOTL}$$

On the other hand, the hierarchy of k-FOTL *approximates* 2-FOL:

Theorem 3.46 *Let ψ be a 2-FOL formula. Then there is a natural number k and a k-FOTL formula φ such that $\psi \equiv \varphi$.*

P r o o f: Let k be the maximal number of free variables in the subformulas of ψ of the form $\exists x.\psi'$. Then the whole formula ψ can be encoded by choosing the maximal subformulas of ψ that are (syntactically) temporal operators. These connectives form a finite set of connectives of dimension k. This gives immediately the formula φ in k-FOTL.

3.6.3 Temporal Relational Algebras

Previous results also show that every temporal relational algebra is strictly weaker the temporal relational calculus:

Theorem 3.47 *Let Γ be a first-order definable relational algebra over uniform relational types. Then the relational algebra queries over Γ cannot express all first-order queries.*

P r o o f: Let the maximal arity of the operators in Γ be k and the maximal quantifier depth of the defining formulas be n. Clearly, every operator can be expressed in $(k+n)$-FOTL. The result then follows from Theorem 3.45.

3.6.4 Ordered Data Domain

Theorems 3.33 and 3.37 hold even in the presence of the linear order on the data domain (which is a usual assumption in relational databases). The only difference is in the choice of the cardinality of the subsets of elements that are in the relations P and R. The subsets are chosen similarly to proof of Theorem 3.37. We choose the data domain to be $\{1, \ldots, 2^{n+1}\}$ and define the τ-structures as

$$\mathcal{A}_n = \langle Q, <; \{1, \ldots, 2^{n+1}\}, =; P^{2^{n+1}}_{2^n-1}, R^{2^{n+1}}_{2^n} \rangle \text{and } \mathcal{B}_n = \langle Q, <; \{1, \ldots, 2^{n+1}\}, =; P^{2^{n+1}}_{2^n}, R^{2^{n+1}}_{2^n} \rangle$$

The strategy for Player II is essentially the same as two linear orders with cardinalities 2^n and $2^n - 1$ cannot be distinguished by n data moves (cf. Example 3.17), and Player I cannot pick more than $n-1$ data moves in a round of the game for temporal logic of length n.

3.6.5 Fixed Data Domain

Corollary 3.38 may suggest that the logic FOTL is always strictly less expressive than 2-FOL in the first-order case. This is not true either:

Example 3.48 Consider the situation where D is a finite set of fixed size. In this case the proof of Lemma 3.37 will not work as we cannot build the structures $\mathcal{A}_n$ and $\mathcal{B}_n$ of arbitrary size. If, moreover, every element of D is denotable by a constant we can replace quantifiers over D as follows:

$$\forall x.\varphi = \bigwedge_{c\in D} \varphi[x/c] \qquad \exists x.\varphi = \bigvee_{c\in D} \varphi[x/c]$$

It is easy to see that this case is essentially the same as the case of propositional temporal logic. Note that the fixed size of the domain is a first-order property (i.e., it can be defined by a first-order theory of D). However, the size of the resulting formula depends on the size of D.

3.6.6 Maximal Expressive power of FOTL

A natural question is: "what is the strongest possible FOTL for a given dimension k, if there is a bound?" It turns out that over the complete linear orders FOTL(**since**,**until**) is such a logic (for $k = 1$). This result follows immediately from Proposition 3.1 and the observation that every temporal connective can be decomposed into embedded applications of the **since** and **until** connectives (as the temporal connectives can be thought of as formulas in propositional temporal logic). [42] introduces several propositional logics complete for the particular choice of $k > 1$.

3.7 Conclusion

In this chapter we have defined a model-theoretical game that captures the expressive power of k-FOTL in a very general setting—the definition is independent of the underlying signatures and theories.

We have used this game to resolve several open questions about temporal logics. Our main result is the separation of FOTL from 2-FOL: 2-FOL is strictly more expressive than *any* variant of temporal logic over dense orders. Thus k-FOTL cannot be used as a basis for the design of a first-order complete temporal query language.

We conjecture that the same results hold for the expressive power over all (sufficiently large) linear orders—the proof of Theorem 3.37 for discrete orders needs to be modified using a *restricted* density property that allows one to maintain the winning strategy for PLAYER II but makes the order isomorphic to the standard linear order on integers. We can prove Theorem 3.37 for FOTL with unary temporal connectives over discrete linear orders. However, substantiating this claim in its full generality is the goal of ongoing research.

Future research in this area will concentrate on the following topics:

1. In the case of complete first-order query languages, new and more general implementation techniques need to be developed for handling the temporal information (cf.

Section 4). The intermediate results of the query evaluation algorithms may be more complicated than allowed by the uniform database schema. Thus, such intermediate results cannot be stored in a uniform temporal database itself as auxiliary relations.

2. In the area of incomplete but subquery-closed temporal query languages the relationships between the different sets of temporal connectives need to be investigated. For many applications a limited number of simultaneous temporal contexts may be sufficient. This would allow the use of a sufficiently large set of temporal connectives tailored for the specific application.

3. The temporal databases are infinite structures in general. However, all the data stored in the database must be finitely representable (cf. Chapter 2). This restriction disallows the use of arbitrarily complex temporal databases (cf. Definition 3.2)[4]. Sufficient restriction on the class of allowed temporal databases may be sufficient to guarantee both the closure and completeness. We have seen that bounded number of elements in the data domain is such a restriction. Are there other (nontrivial) restrictions (especially on the temporal domain) that also guarantee both the closure and completeness?

3.8 Related Work

The separation is proven using a modification of the Ehrenfeucht-Fraïssé Games to capture the properties of Temporal Logic. In [50] pebble games have been used to show expressivity results for the *monadic logic* over linear orders. However, our results and techniques are different, as we are interested in the *first-order* temporal logics (and the corresponding 2-FOL). The method introduced in [50] is no longer sufficient as it cannot handle unrestricted quantification over the data sort.

In [57] a restricted version of 2-FOTL was presented introducing the *now* connective that allows one to reset the temporal context of a subformula to the original evaluation point. This logic was also shown to be strictly stronger than FOTL. However, the technique used in [57] does not apply to temporal databases—the proof of the fact is carried out over first-order structures that cannot be finitely encoded as the contents of the *database relation(s)* is an infinite and coinfinite[5] *at every time instant*. Such sets cannot be represented over the data domain (as the theory of equality can finitely encode only finite and cofinite sets.

A recent result [1] that separates FOTL with the **since** and **until** connectives from 2-FOL in the case of finite (sufficiently large) linear orders supports our conjecture. However [1] uses a counting argument in the proof and thus it does not generalize to arbitrary linear orders. Also, the relationship between this work and various sets of temporal connectives defined in [42] is not clear.

Bibliographical Notes

Temporal logic with the connectives **since** and **until** was introduced in [56]. Prior to that,

[4]Note that all the databases used in our proofs are finitely representable.

[5]I.e., its complement is infinite.

temporal logic was studied as a form of modal logics on unlabeled transition systems, whose transition relation is an order [69]. There has been a substantial interest in temporal and modal logics in the specification/verification community. A whole volume of the *Oxford Logic Guides* [42] is dedicated to the study of various aspects of temporal logic. However, with the exception of [57], almost all the effort is spent on *propositional logics* and the corresponding monadic first-order counterparts, which are not very useful in the area of temporal databases.

Chapter 4

Constraint Databases

One of the leading approaches to finite encodings of classes of infinite databases is the use of constraints—quantifier-free characteristic formulas that describe sets of elements in the particular domain. This section introduces an abstract notion of a constraint class as a basis for a finite encoding of (classes of) such infinite sets. We introduce the basic building blocks in terms of which the evaluation of Datalog$^{\mathcal{C}}$ *(deductive queries) over such representation is defined. The standard bottom-up query evaluation procedure is defined in terms of these building blocks. We prove the correctness of the evaluation over such encoding and show a sufficient condition for termination of query evaluation. We also discuss a general method for combining different classes of constraints in the framework of Datalog.*

4.1 Introduction

Generalized databases [12, 55, 60] are infinite databases that can be represented using finite sets of *generalized* (or *constraint*) tuples. A number of query languages over such databases have been studied. The proposed query languages differ with respect to:

- the underlying inference mechanism (first order vs. deductive)
- the constraint language used.

In this section we study generalized Datalog programs (function-free logic programs) that operate on constraint tuples instead of ground atoms. This idea comes from the area of constraint logic programming [53] and allows declarative specification of problems that are not solvable using the standard Datalog framework (e.g., reasoning about infinite periodic sets of integers).

Constraint tuples are a natural extension of relational tuples, e.g.:

$R(1,2)$	tuple of arity 2 representing the set $\{(x,y) \mid x = 1 \wedge y = 2\}$
$R(x,y) :-\ x+5<y$	generalized tuple $\{(x,y) \mid x+5<y\}$ (order constraint)
$R(x,y) :-\ x \equiv_3 (y+1)$	$\{(x,y) \mid 3 \text{ divides } x-(y+1)\}$ (periodicity constraint)

The later two tuples represent infinite sets of integers. The constraint tuples can be combined using conjunction, e.g.,

$$R(x) :-\ 0 < x, x \equiv_5 3, x < 1000000.$$

Note that this constraint represents the finite set $\{3, 8, \ldots, 999998\}$ which can be handled using the standard relational techniques. Conjunction is also used to produce tuples of higher arity, e.g.,

$$R(x, y) :- \ 0 < x, x + 3 < y.$$

Nevertheless the constraint representation is much more compact than the explicit representation of the same set and potentially amenable to a more efficient evaluation.

We give an abstract definition of the objects (constraints) that the deductive query evaluation algorithm is defined over.

4.2 Constraint Classes

The objects (constraints) used to finitely encode infinite sets are generated from a set of *atomic* constraints. These constraints serve as the basis for defining the underlying set of objects over which operations needed for query evaluation are defined:

Definition 4.1 *Let $\mathcal{C}_0$ be a set of satisfiable atomic constraints. We define $\mathcal{C}$ to be the least set of constraints that satisfies the following rules:*

1. true $\in \mathcal{C}$
2. $\mathcal{C}_0 \subseteq \mathcal{C}$
3. *if $C_1, C_2 \in \mathcal{C}$ and $C_1 \wedge C_2$ is satisfiable then $C_1 \wedge C_2 \in \mathcal{C}$*
4. *if $C \in \mathcal{C}$ and $x \in FV(C)$*[1] *then there exists a quantifier free formula $C_1 \vee \ldots \vee C_k$ (in DNF) equivalent to $\exists x.C$ such that $C_i \in \mathcal{C}$ for every satisfiable C_i where $0 < i \leq k$.*
5. *if $C \in \mathcal{C}$ and θ is a renaming of variables then $C\theta \in \mathcal{C}$.*

This definition is similar to the definition of Constraint Domain [51]. However, $\mathcal{C}$ contains only satisfiable constraints. The elements of $\mathcal{C}$ are used as a finite representation of (possibly infinite) relations in a constraint database: Similarly to Definition 2.13 we have

Notation 4.2 *Let $C \in \mathcal{C}$. Let $\|C\|$ denote the set of valuations*[2] *θ such that $\theta \models C$. For a set of constraints $S \subseteq \mathcal{C}$ we define $\|S\| = \bigcup_{C \in S} \|C\|$.*

The query evaluation over such a representation is based on the following operations:

Definition 4.3 (Constraint Class) *A* Constraint class *is a set of constraints $\mathcal{C}$ from Definition 4.1 equipped with the following (computable) operations:*

Constraint Conjunction $\wedge^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \to \mathcal{C} \cup \{\bot\}$ *that computes the* conjunction $C_1 \wedge C_2$ *for every pair of constraints $(C_1, C_2) \in \mathcal{C} \times \mathcal{C}$ if the conjunction is satisfiable—otherwise it returns $\bot$. Also it must make the following diagram to commute:*

$$\begin{tikzcd}
C_1, C_2 \arrow[rr, "\wedge^{\mathcal{C}}"] \arrow[d] & & C_1 \wedge C_2 \arrow[d] \\
\|C_1\|, \|C_2\| \arrow[r, "\bowtie"] & \|C_1\| \bowtie \|C_2\| \arrow[r, phantom, "="] & \|C_1 \wedge C_2\|
\end{tikzcd}$$

[1] The set of free variables in the constraint C.

[2] We identify tuples and valuations over finite sets of variable names.

where $\bowtie$ is the relational algebra join.

Constraint Projection $\exists^{\mathcal{C}} : \mathcal{C} \times 2^{\mathrm{Vars}} \to 2^{\mathcal{C}}$ *that for every constraint $C \in \mathcal{C}$ and every set of variable names $V \subseteq FV(C) \subset$ Vars[3] computes a set $\{C_i\}$ (a* projection *of C to V) that satisfies the condition*

$$\left(\bigvee_{C_i \in \exists^{\mathcal{C}}(C,V)} C_i \right) \equiv \exists x_1. \cdots \exists x_l.C$$

where $x_j \in FV(C) - V$ for all $0 < j \leq l$, and makes the following diagram commute

$$\begin{tikzcd} C \arrow[rr, "\lambda x.\exists^{\mathcal{C}}(x,V)"] \arrow[d] & & \{C_i\}_{i=1}^{k} \arrow[d] \\ \|C\| \arrow[r, "\pi_V"] & \pi_V \|C\| = & \bigcup_{i=1}^{k} \|C_i\| \end{tikzcd}$$

Note that by Definition 4.1 the function is well defined and always returns a finite *subset of $\mathcal{C}$.*

Constraint Subsumption $\leq^{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \to$ bool *that satisfies the following condition:*

$$C_1 \leq^{\mathcal{C}} C_2 \text{ implies } \|C_1\| \subset \|C_2\|$$

The first two operations are, in the world of constraints, the equivalents of relational algebra operations: join and projection. However, while in the case of ground tuples, represented using equality constraints, the constraint projection returns always only one constraint (tuple), in the case of more general constraints (e.g., Definitions 5.38 or [96]) the constraint projection may return a set containing more than one tuple. The last operation, constraint subsumption, is used to eliminate redundant answers[4] and reduces to checking equality in the case of ground tuples. Note that the $\not\leq^{\mathcal{C}}$ is not unique by definition and does not have to imply $\not\supset$. However, a *better approximation* of the $\supset$ relation by the $\leq^{\mathcal{C}}$ operation reduces the number of possible duplicate answers and improves the efficiency of the evaluation methods (cf. Chapter 7). In the following text we omit the superscripts ${}^{\mathcal{C}}$. We also use a *strict*[5] version of $\wedge^{\mathcal{C}}$. Figure 4.1 shows an example of a naive evaluation procedure for first-order queries over a constraint database.

The following definition states a fundamental property of the constraint classes on which the termination proofs of bottom-up query evaluation procedures are implicitly based [73, 85, 96, 100].

Definition 4.4 (Constraint-compact Class of Constraints)
Let $\mathcal{C}$ be a constraint class. If for every finite set of variables V and for every set $\mathcal{D} \subseteq \mathcal{C}$ such that

$$\forall C \in \mathcal{D}.FV(C) \subseteq V$$

[3] Vars is the set of all legal variable names.
[4] Similarly to duplicate elimination in the *ground* case.
[5] I.e., $\perp$-preserving.

there is a finite subset $\mathcal{D}_{fin} \subseteq \mathcal{D}$ such that

$$\forall C \in \mathcal{D}. \exists C' \in \mathcal{D}_{fin}. C \leq^{\mathcal{C}} C'$$

(i.e., $\mathcal{D}_{fin}$ covers $\mathcal{D}$ with respect to $\leq^{\mathcal{C}}$) then the constraint class $\mathcal{C}$ is called constraint-compact.

This property plays a central role in the termination proofs of all the evaluation methods for deductive queries over the particular constraint class based on the immediate consequence operator.

Definition 4.5 (Datalog$^{\mathcal{C}}$ **Program)** *Let $\mathcal{C}$ be a constraint class . An* atom *is a predicate symbol with distinct variables as its arguments. A* Datalog$^{\mathcal{C}}$ *program is a set of clauses of the form*

$$A \leftarrow D, B_1, \ldots, B_k$$

where A and B_i are atoms and $D \in \mathcal{C}$.

For simplicity we assume that the extensional database (EDB) is represented by a set of unit clauses of the form $A \leftarrow D$ as a part of the constraint database (a Datalog$^{\mathcal{C}}$ program). A query over such a database is a tuple containing of an atom and a constraint the returned tuples have to satisfy. The database itself is included as the last component of the query:

Definition 4.6 (Query) *Let P be a* Datalog$^{\mathcal{C}}$ *program, G an atom, and $C \in \mathcal{C}$. We call the tuple (G, C, P) a* query *over P. The* answer *to the query (G, C, P) is a set of valuations θ such that $P \models (G \wedge C)\theta$. A* query evaluation procedure *is an algorithm that computes an answer to the query. A query is* partially instantiated *if the constraint C is nontrivial (i.e., $C \neq$ true).*

Note that this definition of query clouds the distinction between the data complexity and the overall complexity of query evaluation. In the following chapters we assume that the data complexity of a query (G, C, P) is measured with respect to the number of clauses in P and the overall complexity with respect to size of P. This is very natural and corresponds to the intuition given in Definitions 2.20 and 2.21. The number of clauses in P includes the number of facts and thus is equal to the size of the generalized EDB relations (up to a constant multiple).

Let $R = \{C_1, C_2, \ldots, C_n\}$ be a generalized relation and φ a first-order query (over the relation R only). Then the query evaluation of such a first-order query can be thought of as

$$\varphi(R) \equiv \varphi(C_1 \vee \ldots \vee C_n) \longrightarrow C_1' \vee \ldots \vee C_k' \equiv \varphi'(R)$$

where $\longrightarrow$ is the quantifier elimination procedure (and conversion to DNF). The answer relation is then the generalized relation $R = \{C_1', \ldots, C_k'\}$.

Figure 4.1: Constraints and First-order Query Evaluation.

4.3 Closed-form Bottom-up Evaluation

The usual approach to query evaluation for Datalog$^{\mathcal{C}}$ is a variation on the bottom-up evaluation algorithm [100]. In its simplest form a bottom-up evaluation algorithm is defined as follows:

Definition 4.7 (Interpretation) *Let $R(x_1, \ldots, x_k)$ be an atom and $C \in \mathcal{C}$ a constraint such that $FV(C) \subseteq FV(R)$. A pair (R, C) is a* named constraint*. A $\mathcal{C}$-*interpretation *is a set of named constraints.*

Named constraints play the role of ground atoms (tuples) stored in the tables of a standard relational (or deductive) system. The definition of the immediate consequence (TP) operator on the constraint representation is similar to the definition of the operator on ground atoms. However, in the constraint case, all the operations in the definition of the TP operator are defined with respect to the chosen constraint class $\mathcal{C}$ (cf. Definition 4.3).

Definition 4.8 (Immediate Consequence Operator) *Let I be a $\mathcal{C}$-interpretation and P a* Datalog$^{\mathcal{C}}$ *program. We define*

$$\begin{aligned}\mathrm{TP}_{\mathcal{C}}(I) = \{(A, C') : \; & (A \leftarrow D, B_1, \ldots, B_k) \in P, \\ & (B_i, C_i) \in I \textit{ for all } 0 < i \leq k, D \in \mathcal{C}, \\ & C = D \wedge C_1 \wedge \ldots \wedge C_k \textit{ exists}, C' \in \exists_A.C, \\ & \textit{and if } C' \notin I \textit{ then } C' \not\leq C'' \textit{ for any } (A, C'') \in I\}\end{aligned}$$

where $\exists_A$ is a shorthand for $\lambda x.\exists^{\mathcal{C}}(x, FV(A))$. The variables in the constraints are renamed using the variable names in the associated atoms in the body of the clause $A \leftarrow D, B_1, \ldots, B_k$.

Lemma 4.9 $\mathrm{TP}_{\mathcal{C}}$ *is monotonic and continuous (preserves suprema of directed chains).*

P r o o f: Continuity: Let $I_0 \subseteq I_1 \subseteq \ldots$ be a chain of $\mathcal{C}$-interpretations such that if $(A, C) \in I_j$ then either $(A, C) \in I_{j-1}$ or $C \not\leq C'$ for any $(A, C') \in I_{j-1}$ (a directed chain).

$\subseteq$: Let $(A, C) \in \mathrm{TP}_{\mathcal{C}}(\bigcup_{i>0} I_i)$ be a named constraint. Then there is a clause $A \leftarrow D, B_1, \ldots, B_k$ such that $(B_i, C_i) \in \bigcup_{i>0} I_i$ and $C \in \exists_A.D \wedge C_1 \wedge \ldots \wedge C_k$. Thus, there must be $j > 0$ such that $(B_i, C_i) \in I_j$. Then $(A, C) \in \mathrm{TP}_{\mathcal{C}}(I_j) \subseteq \bigcup_{i>0} \mathrm{TP}_{\mathcal{C}}(I_i)$.

$\supseteq$: Let $(A, C) \in \bigcup_{i>0} \mathrm{TP}_{\mathcal{C}}(I_i)$. Then there is $j > 0$ such that $(A, C) \in \mathrm{TP}_{\mathcal{C}}(I_j)$. Similarly to the previous case, there is a clause $A \leftarrow D, B_1, \ldots, B_k$ such that $(B_i, C_i) \in I_j \subseteq \bigcup_{i>0} I_i$ and $C \in \exists_A.D \wedge C_1 \wedge \ldots \wedge C_k$. Thus $(A, C) \in \mathrm{TP}_{\mathcal{C}}(\bigcup_{i>0} I_i)$.

Monotonicity: follows immediately from continuity.

The standard bottom-up evaluation algorithm remains the same as for ground tuples (i.e., all the modifications needed for the evaluation of *constraint* queries are encapsulated in the definition of the $\mathrm{TP}_{\mathcal{C}}$ operator).

Algorithm 4.10 (Naive Bottom-up Evaluation) *Let (G, C, P) be a query and I and J two variables over $\mathcal{C}$- interpretations.*

$I := \emptyset;$
repeat
    $J := I;$
    $I := \mathrm{TP}_{\mathcal{C}}(I)$
while $J \neq I;$
return $\{C \wedge D : (G, D) \in I\}$

Algorithm 4.10 produces the same result as the standard bottom-up evaluation of the same program with respect to ground interpretations. We show this fact relating the iteration of the standard TP operator on the ground interpretations to the iteration of $\mathrm{TP}_{\mathcal{C}}$:

Definition 4.11 (Immediate consequence operator on ground interpretations) *Let I be a ground interpretation and P a* $\mathrm{Datalog}^{\mathcal{C}}$ *program. We define*

$$\mathrm{TP}(I) = \{A\theta : A \leftarrow D, B_1, \ldots, B_k \in P \wedge \theta \models D \wedge B_i\theta \in I\}$$

Now we can compare the results obtained by iteration of TP and $\mathrm{TP}_{\mathcal{C}}$ with respect to the same $\mathrm{Datalog}^{\mathcal{C}}$ program P:

Theorem 4.12 *For any* $\mathrm{Datalog}^{\mathcal{C}}$ *program P and any predicate symbol R in P*

$$R(c_1, \ldots, c_k) \in \mathrm{TP}^{\omega}(\emptyset) \iff (R, C) \in \mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset) \wedge [c_1/x_1, \ldots, c_k/x_k] \models C$$

where TP *is the standard consequence operator on Herbrand interpretations.*

P r o o f: By simultaneous induction on TP and $\mathrm{TP}_{\mathcal{C}}$. Base case holds trivially.

Let $A(c_1, \ldots, c_k) \in \mathrm{TP}^{i+1}(\emptyset)$. Then there exists a clause $A \leftarrow B_1, \ldots, B_k, D \in P$ and a (ground) substitution θ such that $A\theta = A(c_1, \ldots, c_k)$, $B_i\theta \in \mathrm{TP}^i(\emptyset)$, and $D\theta$ holds. Then by IH $(B_i, C_i) \in \mathrm{TP}^i_{\mathcal{C}}(\emptyset)$ where $\theta \models C_i$. Also, there exists a representation of the constraint D (and $\theta \models D$). Then by Definitions 4.3 and 4.8 $C_1 \wedge \ldots \wedge C_m \wedge D \in \mathcal{C}$ exists, $(A, \pi_{FV(A)}(G)) \subseteq \mathrm{TP}^{i+1}(\emptyset)$ and $\theta \models \pi_{FV(A)}(G)$.

Let $(A, C) \in \mathrm{TP}^{i+1}_{\mathcal{C}}(\emptyset)$ and $\theta \models C$ (θ must be "ground", i.e., it must assign values to all free variables of C). Then by Definition 4.8 there exists a clause $A \leftarrow B_1, \ldots, B_k, D \in P$ such that $(B_i, C_i) \in \mathrm{TP}^{i-1}_{\mathcal{C}}(\emptyset)$ and $\theta' \models C_1 \wedge \ldots \wedge C_m \wedge D$ where $\theta \subseteq \theta'$. By IH $B_i\theta' \in \mathrm{TP}^{i-1}(\emptyset)$ and thus by the definition of TP we have $A\theta \in \mathrm{TP}^i(\emptyset)$.

Both TP and $\mathrm{TP}_{\mathcal{C}}$ are continuous. Thus,

$$(A, C) \in \mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset) \iff (\forall\theta.\theta \models C \supset A\theta \in \mathrm{TP}^{\omega}(\emptyset)).$$

Note that the number of iterations of TP and $\mathrm{TP}_{\mathcal{C}}$ is the same. However, the TP operator has to operate on infinite ground interpretations.

This arrangement also shows how other evaluation procedures based on the TP operator can be utilized for constraint query evaluation, e.g., the semi-naive bottom-up evaluation (cf. Chapter 7 or [100]).

To show termination of Algorithm 4.10 we need:

Theorem 4.13 (Termination) *Let $\mathcal{C}$ be a constraint-compact class of constraints. Then Algorithm 4.10 terminates for every query.*

P r o o f: By contradiction: assume that Algorithm 4.10 does not terminate. Thus every iteration of the loop produces at least one named constraint that is not subsumed by any named constraint in I (follows immediately from the termination condition of Algorithm 4.10's *repeat ... while* loop). Because the Datalog$^{\mathcal{C}}$ program contains only finitely many different predicate symbols, there must be at least one predicate symbol for which the algorithm derives an infinite sequence of named constraints. Let the sequence be $\{C_i\}_{i=1}^{\infty}$. However, the class $\mathcal{C}$ is constraint compact. Thus, there is a finite subset $\{C_{i_1}, \ldots, C_{i_k}\}$ of $\{C_i\}_{i=1}^{\infty}$, that covers the whole infinite sequence. Let $j = \max\{i_1, \ldots, i_k\}$. Then, by Definition 4.4, C_i is subsumed by some constraint in $\{C_{i_1}, \ldots, C_{i_k}\}$ for every $i > j$. This is a contradiction as all $C_{i_1}, \ldots, C_{i_k} \in I$ and $\mathrm{TP}_{\mathcal{C}}$ will not derive any named constraints, subsumed by I (but not in I) by Definition 4.8.

Note that $\mathrm{TP}^{\omega}(\emptyset)$ may be an infinite set. However, the set $\mathrm{TP}_{\mathcal{C}}^{\omega}(\emptyset)$ is always finite for any constraint compact class of constraints $\mathcal{C}$.

Corollary 4.14 *Let $\mathcal{C}$ be a finite class of constraints. Then Algorithm 4.10 terminates for every query.*

An instance of the general definitions is the standard Datalog over ground interpretations (tuples):

Example 4.15 Let D be a flat domain of "uninterpreted constants" equipped with the equality relation only. Then we can define the language Datalog$^{=D}$ as follows:

1. a =-interpretation is a set of tuples (A, R_A) where A is a predicate symbol and R_A is a A-tuple over D.
2. a conjunction of two atoms (A, R_A) and (B, R_B) is a tuple where fields of R_A are matched against fields of R_B with the same name; the conjunction is consistent if and only if the matching is successful,
3. a projection operation is based on removing a field from a given tuple, and
4. a named constraint (A, R_A) subsumes (A, S_A) iff $S_A = R_A$.

Both the $\mathrm{TP}_{=D}$ operator, the bottom-up evaluation procedure, and the termination argument follow immediately from Definitions 4.8 and 4.13.

4.4 Classes of Constraints with Disjoint Domains

The previous section suggests that we can integrate various classes of constraints into Datalog. The main difficulty of doing so is the design of the quantifier elimination procedure (the constraint projection—cf. Chapter 5). However, combining classes of constraints over disjoint domains (like integers and uninterpreted constants) is not a problem. Due to the *constraint* approach and typing of variables, the evaluation over each of the classes is independent of the other classes.

Definition 4.16 (Datalog$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$) *Let $\mathcal{S}_i$ be classes of constraints for $0 < i \leq n$. A Datalog$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ program is a finite set of clauses of the form*

$$A \leftarrow B_1, \ldots, B_m, C_1, \ldots, C_n$$

where A and $B_1, \ldots, B_m$ are predicate symbols, and C_i is a satisfiable constraint in $\mathcal{S}_i$.

Note that each of the classes $\mathcal{S}_i$ contains the trivial constraint *true*.

Definition 4.17 *Let $\mathcal{S}$ be a constraint class, $\mathrm{TP}_{\mathcal{S}}$ the consequence operator for Datalog$^{\mathcal{S}}$, and P a Datalog$^{\mathcal{S}}$ program and C a clause in P. Then the* consequence operator relative to C *is defined as $\mathrm{TP}_{\mathcal{S}}$ with respect to the program $\{C\}$ and denoted $\mathrm{TP}_{\mathcal{S}}^{C}$.*

A $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$-interpretation is a set of $(n+1)$-tuples $(A, G_1, \ldots, G_n)$ where A is a predicate symbol and G_i a finite representation of a constraint $C_i \in \mathcal{C}_i$. All G_i's must have the same arity as he atom A.

Definition 4.18 *Let P be a Datalog$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ program, such that the domains of $\mathcal{S}_i$ and $\mathcal{S}_j$ are disjoint for all $i \neq j$, I a $(\mathcal{S}_1, \ldots, \mathcal{S}_n)$-interpretation, and*

$$\pi_{\mathcal{S}_j}(I) = \{(A, G_j) : (A, G_1, \ldots, G_n) \in I\}.$$

Then we define

$$\begin{aligned}\mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}(I) = \{(A, G_1, \ldots, G_n) : \quad & A \leftarrow B_1, \ldots, B_m, C_1, \ldots, C_n \in P \\ & (A, G_i) \in \mathrm{TP}_{\mathcal{S}_i}^{A \leftarrow B_1,\ldots,B_m,C_i}(\pi_{\mathcal{S}_i}(I))\}\end{aligned}$$

Note that G_i must exist for each $0 < i \leq n$ for $\mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ to produce an output.

The correctness and the termination for the bottom-up evaluation of Datalog$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ are proven as follows:

Theorem 4.19 *For a Datalog$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ program P and R a predicate symbol in P*

$$\begin{aligned}R(c_1, \ldots, c_k) \in \mathrm{TP}^{\omega}(\emptyset) \quad \Longleftrightarrow \quad & (R, G_1, \ldots, G_n) \in \mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}^{\omega}(\emptyset) \\ & \wedge (\forall i.[c_1/x_1, \ldots, c_k/x_k] \models G_i)\end{aligned}$$

P r o o f: By a simultaneous induction on the number of iterations of TP. The base case holds trivially.

Let $A(c_1, \ldots, c_k) \in \mathrm{TP}^{i+1}(\emptyset)$. Then there exists a clause $A \leftarrow B_1, \ldots, B_k, C_1, \ldots, C_n \in P$ and a (ground) substitution θ such that $A\theta = A(c_1, \ldots, c_k)$, $B_i\theta \in \mathrm{TP}^{i}(\emptyset)$, and $C_i\theta$ holds for $0 < i \leq n$. Then by IH there are constraints $G_i^1, \ldots, G_i^m$ such that $(B_i, G_i^1, \ldots, G_i^n) \in \mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}^{i}(\emptyset)$ where $\theta \models G_j^i$ for $0 < j \leq n$. Thus $(A, G^1, \ldots, G^m) \in \mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}^{i+1}(\emptyset)$ by the definition of the consequence operator.

The opposite direction is similar. The claim holds by the observation that TP and $\mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ are continuous (similar to Lemma 4.9).

Lemma 4.20 *Let P be a Datalog$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ program. Then $\pi_{\mathcal{S}_j}(\mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}^{i}(\emptyset)) \subseteq \mathrm{TP}_{\mathcal{S}_j}^{i}(\emptyset)$ for all $0 < j \leq n$ and $0 \leq i$.*

P r o o f: Immediate from Definition 4.18.

Note that the $\mathrm{TP}^i_{\mathcal{S}_j}$ operator is applied on clauses of P projected to $\mathrm{Datalog}^{\mathcal{S}_j}$, i.e., all constraint goals except C_j are replaced by *true* in the clauses of P.

Corollary 4.21 *Let* $\mathrm{TP}^\omega_{\mathcal{S}_j}$ *be finite for every* $0 < j \leq n$. *Then the* $\mathrm{TP}^\omega_{\mathcal{S}_1,\ldots,\mathcal{S}_n}(\emptyset)$ *is finite and the bottom-up evaluation of Datalog*$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ *programs terminates.*

P r o o f: Immediate from Lemma 4.20.

Corollary 4.22 *Let* $\mathcal{S}_j$ *be constraint compact for every* $0 < j \leq n$. *Then the evaluation of every Datalog*$^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ *program terminates.*

We can use the standard bottom-up algorithm to evaluate the $\mathrm{Datalog}^{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ queries and the termination is guaranteed. Moreover, the product of the cardinalities of the $\mathrm{TP}^\omega_{\mathcal{S}_j}$ sets limits the number of applications of the operator $\mathrm{TP}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}$ needed to reach the fixed point. Especially, if all the sets are polynomial in the size of the program then we have a PTIME evaluation procedure.

4.5 First-order queries, Negation

Clearly, using the operations of the constraint class, we can also evaluate all positive existential[6] first-order queries of the relational calculus. This can be easily seen, as every positive existential query can be written as a non-recursive $\mathrm{Datalog}^{\mathcal{C}}$ program [100]. However, to use this methodology to evaluate all first-order queries over the constraint databases we need to extend the definition of the constraint class by allowing some sort of *complementation*:

Definition 4.23 (Constraint Class closed under complementation) *Let* $\mathcal{C}$ *be a constraint class. We say that* $\mathcal{C}$ is closed under complementation *if there exist* $C_1, \ldots, C_k \in \mathcal{C}$ *such that*

$$\neg^{\mathcal{C}} : C \mapsto \{C_1, \ldots, C_k\}$$

and $\neg C \equiv C_1 \vee \ldots \vee C_k$ *for all* $C \in \mathcal{C}$. *We also require the existence of a (computable) function* $\neg^{\mathcal{C}} : \mathcal{C} \to 2^{\mathcal{C}}$ *that computes the complement of every constraint in* $\mathcal{C}$.

Note that an image of every $C \in \mathcal{C}$ under $\neg^{\mathcal{C}}$ is always a finite set of constraints. This operation can be used to compute the complements of generalized relations as follows:

Lemma 4.24 *Let* $\mathcal{C}$ *be a constraint class closed under complementation, and* R *be a generalized relation over* $\mathcal{C}$. *Then*

$$\neg^{\mathcal{C}} R := \bigwedge_{C \in R} \neg^{\mathcal{C}} C$$

defines the constraint complement of R *and*

$$D^n - \|R\| = \|\neg^{\mathcal{C}} R\|$$

for all R.

P r o o f: Immediate from Definition 4.23, Definition 4.23, and DeMorgan laws [35, 67].

[6] I.e., queries, specified by formulas, containing only $\wedge, \vee, \exists$.

Note that for every generalized relation the previous lemma gives an algorithm for computing the complement, as R contains only finitely many constraints C and thus we can iterate the constraint conjunction over them.

Similarly we can use the complementation in the case of Datalog$^\neg$. However, in this case we need to make sure that the constraint class is still constraint-compact to guarantee termination[7].

4.6 Conclusion

In this chapter we have defined an abstract notion of the constraint class in terms of which we can encode (classes of) infinite databases. Also, we found general conditions, under which the evaluation of the deductive queries is possible. The correctness of the evaluation and the termination of it is guaranteed for all classes of constraints that are constraint-compact.

4.7 Related Work

This chapter is based on a generalization of the concepts present in the specific results on various concrete constraint query languages, e.g., [55, 60, 96, 100]. We have extracted the common underlying ideas and shown general properties of the abstracted model. A similar approach can be found in [51] for the general constraint logic programming languages. Besides few technical differences (our constraint class is defined *semantically*, while in [51] the constraint domain is generated by a syntactic closure), our main aim is to show termination of deductive queries, which is not an issue for general constraint programming languages: they are extensions of the general-purpose logic programming languages that can be easily shown to be Turing-complete.

Bibliographical Notes

The work on constraint representation of infinite sets of elements was originated in the area of databases in [60]. However, in the Logic Programming community, the area of Constraint Logic Programming has been an active area of research for many years. A survey of the results, implementation techniques, and applications can be found in [51]. The main difference between the database approach and the Logic Programming approach is that the former uses *quantifier elimination* while the later uses resolution as the basis for query evaluation/computation. A comprehensive reference for quantifier elimination in various theories is [62]. A recent survey of constraint databases and query languages can be found in [78].

[7] Another approach is to use syntactic restriction on Datalog$^\neg$ clauses to guarantee termination for a restricted class of queries, e.g., restricting the constraints in the complemented relation to unary ones [77]. Note that the restriction on the syntactic shape of the Datalog$^\neg$ programs often implies the constraint-compactness under such a limited version of complementation.

Chapter 5

Constraint Classes over Integers

This chapter introduces a generalization of Datalog operating on periodicity and gap-order constraints over the set of integers. We instantiate the general definitions from the previous chapter to these particular classes of constraints and prove their properties. A free bonus is a closed-form evaluation procedure running in PTIME[1] *for this class of constraints.*

5.1 Introduction

In this chapter we define a finite representation for periodic sets of integers. Such sets are quite useful for encoding and storing information about periodic activities in the temporal databases, such as schedules or experimental data. In addition, we show how these kinds of constraints can be combined with other classes of constraints over integers—the gap-order constraints [73] and the equality constraints.

Example 5.1 A simple example of a Datalog program that uses integer constraints is a database of airline connections between cities (cf. figure 5.1). A natural query over such a database is "Can I get from Paris to Toronto on Monday starting at 8 a.m. or later, but to arrive no later than 5 p.m.?" This example takes the advantage of both the periodicity and gap-order constraints. Note that time units need to be converted to a smallest common unit (i.e., in our example the smallest common unit can be one minute; `1day` is then just a shorthand for 1440; similarly `Mon10:00` is 2040, etc. Thus `X = Mon10:00(mod 1day)` is a syntactic sugar for $X \equiv_{1440} 2040$). The representation of the inherently periodic information (like "flight every day") is very natural using the periodicity constraints. The conditions of the type "you need at least 15 minutes to check in" are captured easily using the gap-order constraints.

The main contribution of this chapter is the definition of a closed-form bottom-up evaluation procedure for Datalog programs with the integer periodicity and order constraints. Such language can be used as a simple but very expressive query language over temporal databases. The evaluation for the gap-order constraints is based on the results in [73, 75]. The proposed bottom-up evaluation procedure has *polynomial* data complexity. Also, a general method for incorporating various classes of constraints into Datalog is studied.

[1] Under some mild restrictions.

```
% Database of connections:
connection(paris, london, X, Y):-           % flight every day
        X = Mon10:00(mod 1day),
        Y = Mon11:30(mod 1day).
connection(paris, london, X, Y):-           % and weekdays
        X = Mon14:30(mod 1day),
        Y = X+1:30(mod 1day),
        Z = Mon00:00(mod 7days), W = Fri-23:59(mod 7days),
        Z < X, X < W.
connection(london, toronto, X, Y):-         % twice a week
        X = Tue14:30(mod 7days),            % on Tuesday
        Y = Tue16:50(mod 7days).
connection(london, toronto, X, Y):-         % and
        X = Fri14:30(mod 7days),            % on Friday
        Y = Fri16:50(mod 7days).

% Change times for airports
changetime(london, X, Y) :- X+30<Y.         % 30' to change planes

% Planning of trips:
trip(From, To, Start, End) :-
        connection(From, To, Dtime, Atime),
        Start+15<Dtime,                     % 15' check in
        Atime+20<End.                       % 20' baggage claim
trip(From, To, Start, End) :-
        trip(From, Change, Start, Atime),
        trip(Change, To, Dtime, End),
        changetime(Change,Atime,Dtime).     % specific for airport
```

Figure 5.1: Example of Datalog query that uses integer constraints in an essential way.

The rest of the chapter is organized as follows. Section 5.2 gives the definition of a finite representation of periodicity constraints together with all the operations needed for the bottom-up evaluation. Section 5.3 describes the bottom-up evaluation procedure itself and gives proofs of its correctness and termination. Also, the complexity of the evaluation procedure is analyzed. Section 5.4 deals with combining the periodicity constraints with order constraints. Again, correctness, termination, and complexity results are presented. The chapter is concluded with a few open problems and discussion of relevant work.

5.2 Periodicity Graphs

This section describes a graph-based representation of periodicity (congruence) constraints. The representation together with the operations defined on it serves as the basis for a closed-form bottom-up evaluation of Datalog programs with periodicity constraints.

Definition 5.2 (Periodicity Constraint) *Let K be a finite set of natural numbers and $\mathcal{C}_K$ a set of formulas defined inductively as follows:*

1. $x_i \equiv_k (x_j + c)$ *where x_i, x_j are variables, $k \in K$, and $c \in \{0, \ldots, k-1\}$*
2. $x_i \equiv_k c$ *where x_i is a variable, $k \in K$, and $c \in \{0, \ldots, k-1\}$*
3. *if $C_1, C_2 \in \mathcal{C}_K$ then $C_1 \wedge C_2 \in \mathcal{C}_K$*

Let C be an element of $\mathcal{C}_K$ and let X be the set of free variables in C. Then we call C a periodicity constraint over variables X. *A constraint built only using rules (2) and (3) is called a* simple (periodicity) constraint.

In the context of Datalog we can restrict our attention to the class of *simple* constraints without losing expressive power. This restriction allows a more efficient query evaluation procedure. However, the general theory can be applied to other logic-based query languages over generalized databases with periodicity constraints where such simplification might not be possible, e.g., the conjunctive queries.

Constraints of this form are used in place of (ground) atoms during the bottom-up evaluation. They are directly represented using graphs:

Definition 5.3 (Periodicity Graph) *Let $G = (N_G, E_G)$ be a complete graph over $n+1$ nodes such that $N_G = \{0, x_1, \ldots, x_n\}$ and all edges $e \in E_G$ are labeled with a pair of positive integers $\alpha(e)$ and $\beta(e)$ where $0 \leq \beta(e) < \alpha(e)$. We call G a* periodicity graph *over variables $x_1, \ldots, x_n$ (of arity n).*

We define $\alpha(x, y) := \alpha((x, y))$ and $\beta(x, y) := \beta((x, y))$ for $x, y \in N_G$.

More complex periodicity constraints can be represented as a set of such graphs, each representing a disjunct of the corresponding DNF.

Definition 5.4 (Assignment) *Let G be a periodicity graph over $x_1, \ldots, x_n$ and*

$$\vartheta : \{x_1, \ldots, x_n\} \to Z$$

be an assignment of integers to nodes $x_1, \ldots, x_n$ *such that* $\vartheta(0) = 0$. *We say that* ϑ satisfies G *(*$\vartheta \models G$*) if*

$$\forall x, y \in N_G : \vartheta(x) \equiv_{\alpha(x,y)} \vartheta(y) + \beta(x, y)$$

We call G a satisfiable periodicity graph *if there exists an assignment* ϑ *such that* $\vartheta \models G$.

To combine constraints over the same variables we use the following Proposition [48]:

Proposition 5.5 (Chinese Remainder Theorem) *If* $\gcd(k_1, k_2)$ *divides* $|c_1 - c_2|$ *then there exists a unique* c *such that*

$$x \equiv_{k_1} (y + c_1) \wedge x \equiv_{k_2} (y + c_2) \iff x \equiv_{\mathrm{lcm}(k_1,k_2)} (y + c)$$

where $0 \le c < \mathrm{lcm}(k_1, k_2)$. *Otherwise* $x \equiv_{k_1} (y + c_1) \wedge x \equiv_{k_2} (y + c_2)$ *is insonsistent.*

This proposition allows us to combine the periodicity constraints over two given variables into a single constraint or to show the inconsistency of the original constraints.

Definition 5.6 *Let* G *be a periodicity graph over* $x_1, \ldots, x_n$ *and* C *a periodicity constraint over the same set of variables. We say that* G *is equivalent to* C *if for all assignments* ϑ

$$\vartheta \models C \iff \vartheta \models G$$

Each constraint in $\mathcal{C}_K$ can be shown to be equivalent to a periodicity graph. The graph is defined inductively with respect to the structure of the constraint using the following lemmas:

Lemma 5.7 *Let* $C = (x_i \equiv_k x_j + c)$ *be a periodicity constraint and* G *a periodicity graph over* $\{x_i, x_j\}$ *where* $\alpha(0, x_i) = \alpha(0, x_j) = 1$, $\beta(0, x_i) = \beta(0, x_j) = 0$, $\alpha(x_i, x_j) = k$, *and* $\beta(x_i, x_j) = c$. *Then* G *is equivalent to* C.

P r o o f: Immediate from Definitions 5.3, 5.4, and 5.6.

Lemma 5.8 *Let* $C = (x_i \equiv_k c)$ *be a periodicity constraint and* G *a periodicity graph over* $\{x_i\}$ *where* $\alpha(0, x_i) = k$ *and* $\beta(0, x_i) = c$. *Then* G *is equivalent to* C.

P r o o f: Immediate from Definitions 5.3, 5.4, and 5.6.

To define a constraint class of periodicity constraints we need to define the operations on the representation of the individual constraints:

Algorithm 5.9 (Conjunction) *Let* G_1 *and* G_2 *be periodicity graphs over a set of variables* X *and* Y *respectively. We construct a periodicity graph* $G = G_1 \wedge G_2$ *as follows:*

1. *Let* $N_G = N_{G_1} \cup N_{G_2}$ *be nodes of graph* G.
2. *If for any edge* $(x_i, x_j) \in E_{G_1} \cap E_{G_2}$ *the condition*

   $$\gcd(\alpha_{G_1}(x_i, x_j), \alpha_{G_2}(x_i, x_j)) \text{ divides } |\beta_{G_1}(x_i, x_j) - \beta_{G_2}(x_i, x_j)|$$

   is violated, then the conjunction of G_1 *and* G_2 *is not satisfiable and no graph is produced.*

3. *Otherwise we label the edges* $(x_i, x_j) \in E_G$ *as follows*

$$\alpha_G(x_i, x_j) = \begin{cases} \mathrm{lcm}(\alpha_{G_1}(x_i, x_j), \alpha_{G_2}(x_i, x_j)) & (x_i, x_j) \in E_{G_1} \cap E_{G_2} \\ \alpha_{G_1}(x_i, x_j) & (x_i, x_j) \in E_{G_1} - E_{G_2} \\ \alpha_{G_2}(x_i, x_j) & (x_i, x_j) \in E_{G_2} - E_{G_1} \\ 1 & \textit{otherwise} \end{cases}$$

$$\beta_G(x_i, x_j) = \begin{cases} \gamma_{i,j} & (x_i, x_j) \in E_{G_1} \cap E_{G_2} \\ \beta_{G_1}(x_i, x_j) & (x_i, x_j) \in E_{G_1} - E_{G_2} \\ \beta_{G_2}(x_i, x_j) & (x_i, x_j) \in E_{G_2} - E_{G_1} \\ 0 & \textit{otherwise} \end{cases}$$

where $0 \leq \gamma_{i,j} < \alpha_G(x_i, x_j)$ *is the unique solution of the equation*

$$x_i \equiv_{\alpha_{G_1}(x_i,x_j)} x_j + \beta_{G_1}(x_i, x_j) \wedge x_i \equiv_{\alpha_{G_2}(x_i,x_j)} x_j + \beta_{G_2}(x_i, x_j)$$

as in Proposition 5.5.

Lemma 5.10 *Let* C_1 *and* C_2 *be periodicity constraints over the sets of variables* X *and* Y *respectively and* G_1 *and* G_2 *periodicity graphs equivalent to* C_1 *and* C_2*, respectively. If* $C_1 \wedge C_2$ *is satisfiable then there exists a periodicity graph* $G = G_1 \wedge G_2$ *over* $X \cup Y$ *equivalent to* $C_1 \wedge C_2$.

P r o o f: Let $\vartheta \models C_1 \wedge C_2$ be an arbitrary assignment. Then $\vartheta \models C_1$ and $\vartheta \models C_2$ by the assumption $\vartheta \models G_1$ and $\vartheta \models G_2$. Proposition 5.5 condition (2) of the algorithm 5.9 allows combining the constraints over the same pairs of nodes to a single and unique constraint. Because by the assumption the conjunction of the constraints is satisfiable, $\vartheta \models G_1 \wedge G_2$.

On the other hand, if $\vartheta \models G = G_1 \wedge G_2$, then obviously $\vartheta \models G_1$ and $\vartheta \models G_2$. By the assumption $\vartheta \models C_1$ and $\vartheta \models C_2$, and thus $\vartheta \models C_1 \wedge C_2$.

Theorem 5.11 *Let* $C \in \mathcal{C}_K$ *be a satisfiable periodicity constraint over* $x_1, \ldots, x_n$*. Then there exists a periodicity graph* G *over the same set of variables equivalent to this constraint.*

P r o o f: By induction on the structure of C. Base case follows from Lemma 5.7 and Lemma 5.8. The induction step follows from Lemma 5.10 and Lemma 5.13.

When checking wheather or not an assignment of values to any two nodes of a periodicity graph satisfies all the constraints represented by the graph, we need to check all the paths between these two nodes in general. To avoid this problem we introduce a *normal form* for the periodicity graphs, where only a single edge needs to be checked for satisfaction.

Definition 5.12 (Normal Form) *Let* G *be a periodicity graph. We say that* G is in normal form *if for all* $x_i, x_j, x_k \in N_G$

$$(x_i \equiv_{\alpha(x_i,x_j)} (x_j + \beta(x_i, x_j))) \Rightarrow (x_i \equiv_{\mathrm{gcd}(\alpha(x_i,x_k),\alpha(x_k,x_j))} (x_j + (\beta(x_i, x_k) + \beta(x_k, x_j))))$$

where $\mathrm{gcd}(x, y)$ *is the greatest common divisor of* x *and* y.

Lemma 5.13 *All periodicity graphs in normal form are satisfiable.*

P r o o f: By induction on the size of the graph.

Algorithm 5.14 (Periodicity Graph Normalization) *Let $G = (N_G, E_G)$ be an arbitrary periodicity graph.*

```
repeat
  for (x_i, x_j) ∈ E_G do
    for x_k ∈ N_G − {x_i, x_j} do
      M := lcm(α_G(x_i, x_j), gcd(α_G(x_i, x_k), α_G(x_k, x_j))
      if M divides |β(x_i, x_j) − (β(x_i, x_k) + β(x_k, x_j))|
      then
        α_G(x_i, x_j) := M
        β_G(x_i, x_j) := γ
      else
        no graph can be produced and the algorithm fails
while α_G(x_i, x_j) changes for some i, j
```

where γ is the unique solution of the the pair of congruences $x_i \equiv_{\alpha_G(x_i,x_j)} x_j + \beta(x_i, x_j)$ and $x_i \equiv_{\gcd(\alpha_G(x_i,x_k),\alpha_G(x_k,x_j))} x_j + \beta(x_i, x_k) + \beta(x_k, x_j)$ as in Proposition 5.5.

This normalization procedure is also used to detect inconsistent (i.e. not satisfiable) periodicity graphs.

Lemma 5.15 *A periodicity graph is satisfiable if and only if it has an equivalent normal form.*

P r o o f: Let G' be a periodicity graph produced by algorithm 5.14 applied on G. Obviously the resulting graph is in normal form. If an assignment $\vartheta \models G'$ then $\vartheta \models G$ because all labels in G' are implied by those in G. If $\vartheta \models G$ then all paths between any two nodes x_{i_1}, x_{i_k} in G are satisfied by ϑ. Thus (x_{i_1}, x_{i_k}) must be satisfied in G' because its label is a combination of the composite labels of all the paths in G using Proposition 5.5. Every step of algorithm 5.14 propagates the composite constraint implied by the path x_{i_1}, x_{i_k} at least one step closer to the edge (x_{i_1}, x_{i_k}).

Moreover, if any two paths between x_{i_1} to x_{i_k} are inconsistent, algorithm 5.14 fails. Because the longest simple path in a complete graph over $n+1$ nodes has length n we need at most n repetitions of the outermost loop in algorithm 5.14. Then G is satisfiable by Lemma 5.13.

For a closed-form bottom-up evaluation, two additional operations on the periodicity graphs are needed. The first one is the quantifier elimination procedure (constraint projection):

Algorithm 5.16 (Projection) *Let G be a periodicity graph over a set of variables X and $x \in X$. We construct a periodicity graph $G' = \pi_x(G)$ over the set of variables $X - \{x\}$ as follows:*

1. *Let G'' be normal form of G.*
2. *Let G' be $(N_{G''} - \{x\}, E_{G''} - \{(x, y) | y \in E_{G''}\})$ and the labeling of the edges in G' be a restriction of the labeling in G''.*

Lemma 5.17 *Let C be a satisfiable periodicity constraint over a set of variables X, $x \in X$, and G a periodicity graph equivalent to C. Then the periodicity graph $G' = \pi_x(G)$ over $X - \{x\}$ is equivalent to $\exists x.C$.*

P r o o f: Let ϑ be arbitrary such that $\vartheta \models C$. Clearly, $\vartheta \models G$ as well. Then obviously $\vartheta \models \exists x.C$ iff $\vartheta \models G'$.

Note that the 'iff' is preserved because the graph G'' is normalized, i.e., all the paths between any two nodes are 'implied' by the edge between these two nodes. In the case of the *simple* periodicity constraints in Step 1. in algorithm 5.16 can be omitted.

The last operation on the representation of periodicity constraints, required by the definition of constraint class (Definition 4.3), is the subsumption checking:

Definition 5.18 (Subsumption) *Let G_1 and G_2 be periodicity graphs over the same set of variables. We say that G_1* subsumes *G_2 if for all assignments $\vartheta : \vartheta \models G_2$ implies $\vartheta \models G_1$.*

Lemma 5.19 *Let G_1 and G_2 be normal periodicity graphs over the same set of variables. Then G_1* subsumes *G_2 if for every edge (x, y)*

1. *$\alpha_{G_1}(x, y)$ divides $\alpha_{G_2}(x, y)$ and*
2. *$\beta_{G_1}(x, y) = \beta_{G_2}(x, y) \bmod \alpha_{G_1}(x, y)$*

P r o o f: Immediate from Definitions 5.12 and 5.18.

5.3 Datalog with Periodicity Constraints

The previous section introduced a finite representation of the periodicity constraints. This representation is used in the bottom-up evaluation of Datalog$^{\equiv Z}$ programs—the periodicity graphs serve as elements of the computed interpretation. Datalog$^{\equiv Z}$ programs are defined naturally as follows:

Definition 5.20 (Datalog$^{\equiv Z}$ **program)** *Let P be a finite set of function-free clauses of the form*

$$A \leftarrow B_1, \ldots, B_k, C$$

where A and $B_1, \ldots, B_k$ are atoms and C is a satisfiable periodicity constraint in $\mathcal{C}_K$. Then P is a Datalog$^{\equiv Z}$ *program.*

Note that the usual Datalog constants are not allowed here. This is not a problem; the constants can be added using the technique from Section 4.4.

In a Datalog$^{\equiv Z}$ program (with constraints from $\mathcal{C}_K$) every integer constant can be identified with its remainder class in $Z_{\mathrm{lcm}\, K}$[2]. Thus, all the operations on numbers are performed in a "modulo" arithmetic.

Similarly to Definition 4.7 we define the interpretation of a Datalog$^{\equiv Z}$ program with respect to constraint interpretations:

[2] Z_n is the additive cyclic group with n elements.

Definition 5.21 (Named periodicity graph) *Let P be a* Datalog$^{\equiv Z}$ *program, R a predicate symbol that occurs in P of arity k, and G a consistent periodicity graph over the set of variables $\{x_1, \ldots, x_k\}$. We call the pair (R, G) a* named periodicity graph.

The named periodicity graphs take the role of ground atoms in the standard bottom-up evaluation of Datalog:

Definition 5.22 (TP$_\equiv$) *Let P be a* Datalog$^{\equiv Z}$ *program. Let I be a set of named periodicity graphs* (an $\equiv$-interpretation)*. We define*

$$\begin{aligned} \mathrm{TP}_\equiv(I) = \{(A, \pi_{FV(A)}(G)) : \; & A \leftarrow B_1, \ldots, B_m, C \in P \\ & (B_i, G_i) \in I \textit{ for all } 0 < i \leq m \\ & G \textit{ is the normal form of } G_1 \wedge \ldots \wedge G_m \wedge G_C \\ & (A, \pi_{FV(A)}(G)) \textit{ is not subsumed by any atom in } I\} \end{aligned}$$

where $FV(A)$ is the set of free variables in A.

Let $\mathrm{TP}^0_\equiv(I) = I$ *and* $\mathrm{TP}^i_\equiv(I) = \mathrm{TP}_\equiv(\mathrm{TP}^{i-1}_\equiv(I))$ *for* $i > 0$.

Note that the nodes of the periodicity graphs are "renamed" using the variable names in the associated subgoals of the clause; the conjunction operation is then performed with respect to those names (similarly to the natural join; cf. Definition 4.8).

Definition 5.23 (Upper bound) *Let P be a* Datalog$^{\equiv Z}$ *program, $\{C_1, \ldots, C_n\}$ a set of all constraints in P, and k_i the modulo factor in the constraint C_i. Then we say that $\alpha(P) = \mathrm{lcm}\{k_i | 0 < i \leq n\}$ is the maximal modulo factor for P.*

We show that for a given Datalog$^{\equiv Z}$ program (with respect to a fixed set $\mathcal{C}_K$) the labels of all the edges of the periodicity graphs are bounded by a constant. This is shown for the periodicity constraints first:

Lemma 5.24 *Let C be a satisfiable periodicity constraint that occurs in a* Datalog$^{\equiv Z}$ *program P and G a periodicity graph equivalent to C. Then for all $e \in E_G : \alpha_G(e) \leq \alpha(P)$.*

P r o o f: By structural induction on C. For an atomic C the claim holds trivially; for $C = C_1 \wedge C_2$, the claim follows from Lemma 5.10, and for $C = \exists x.C$, the claim follows from Lemma 5.17.

The application of TP$_\equiv$ also preserves the bound:

Lemma 5.25 *Let P be a* Datalog$^{\equiv Z}$ *program and $G \in \mathrm{TP}^i_\equiv(\emptyset)$. Then for all $e \in E_G$: $\alpha(e) \leq \alpha(P)$.*

P r o o f: By induction on i. The claim holds trivially for $\emptyset$. Let $(A, G) \in \mathrm{TP}^{i+1}_\equiv(\emptyset)$. By definition of the TP$_\equiv$ operator, (A, G) is the result of applying the TP$_\equiv$ operator on the set $\mathrm{TP}^i_\equiv(\emptyset)$. The claim holds by IH for all periodicity graphs in $\mathrm{TP}^i_\equiv(\emptyset)$; it also holds for all satisfiable periodicity constraints by Lemma 5.24. G is constructed using a clause in P, i.e., $G = \pi_{FV(A)}(G_1 \wedge \ldots \wedge G_m \wedge G_C)$. Thus, the claim holds for (A, G) by Lemma 5.10, Lemma 5.17, and the idempotence of the lcm operation.

Using the above two lemmas, we have:

Theorem 5.26 *The class of the periodicity constraints is constraint-compact.*

An immediate consequence of this theorem and Theorem 4.13 is:

Corollary 5.27 *The bottom-up evaluation of any* Datalog$^{\equiv Z}$ *terminates.*

5.3.1 Complexity of Bottom-up Evaluation for $\mathrm{Datalog}^{\equiv Z}$

The number of applications of the $\mathrm{TP}_{\equiv}$ operator in algorithm 4.10 can be bounded for each $\mathrm{Datalog}^{\equiv Z}$ program as follows:

Lemma 5.28 *Let P be a $\mathrm{Datalog}^{\equiv Z}$ program, R a predicate symbol in P, and a the arity of R. Then the number of different elements of $\mathrm{TP}^{\omega}_{\equiv}(\emptyset)$ of the form (R,G) is at most $\alpha(P)^{a(a+1)}$.*

P r o o f: For every edge e in G we have only $\alpha(P)$ different labels $\alpha(e)$ by Lemma 5.25 and only $\alpha(P)$ different labels $\beta(e)$ by the definition of periodicity graphs (Definition 5.3). All the graph components of the elements $(R,G) \in \mathrm{TP}^{\omega}_{\equiv}(\emptyset)$ have exactly $\frac{a(a+1)}{2}$ edges.

The actual number of the periodicity graphs in the resulting model is usually smaller due to the subsumption checking (Lemma 5.18). For *simple* constraints the bound is $\alpha(P)^{2a}$ as there are only a nontrivial edges in any of the periodicity graphs.

Corollary 5.29 *Let P be a $\mathrm{Datalog}^{\equiv Z}$ program where all the atomic constraints are from $\mathcal{C}_K$, a the maximal arity of an atom in P, v the number of distinct variables in a single clause of P, p the number of predicate symbols in P, n the number of clauses in P (including facts), and m the maximal number of goals in any clause in P. Then algorithm 4.10 terminates in $O(n)$ steps.*

P r o o f: An application of the $\mathrm{TP}_{\equiv}$ operator takes at most $mO(v^2)+O(v^4)$ steps for each clause in P and each of the $\alpha(P)^{a(a+1)}$ possible assignments of named periodicity graphs to goals in the body of the clause. The subsumption check takes another $a^2\alpha(P)^{a(a+1)}$ steps. Because there are at most $p\alpha(P)^{a(a+1)}$ different elements in the $\mathrm{TP}^{\omega}_{\equiv}(\emptyset)$ (by Lemma 5.28), the total time needed to compute the model is at most:

$$n \cdot \left[\left(a^2\alpha(P)^{a(a+1)}\right)^{m+1} \cdot \left(mO(v^2)+O(v^4)\right)\right] \cdot p\alpha(P)^{a(a+1)} = O(n)$$

as a, m, p, v, and $\alpha(P)$ are constants for a given P with respect to $\mathcal{C}_K$.

For *simple* constraints the application of $\mathrm{TP}_{\equiv}$ takes only $mO(v)$ time. Note that the complexity of the subsumption checking does not affect the overall complexity of the bottom-up evaluation—it depends on the maximal arity of predicates but not on the size of the database.

Also the testing if a standard relational tuple belongs to the computed model can be done in PTIME:

Theorem 5.30 (Tuple recognition) *Let P be a $\mathrm{Datalog}^{\equiv Z}$ program where all the atomic constraints are from $\mathcal{C}_K$ and $A(c_1,\ldots,c_k)$ be a ground atom. Then $P \models A(c_1,\ldots,c_k)$ is decidable in $O(n)$.*

P r o o f: The bottom-up evaluation procedure for P takes $O(n)$ time by Corollary 5.29, testing if A belongs to the $\mathrm{TP}^{\omega}_{\equiv}(\emptyset)$ takes constant time (with respect to the size of P) by Lemma 5.28.

However, it is easy to show that the restriction to a fixed $\mathcal{C}_K$ is crucial for obtaining the polynomial bound: consider the following example:

Let $n > 0$ be a natural number, $p_0, \ldots, p_{n-1}$ distinct prime numbers, and $a_0, \ldots, a_n$ distinct constants. Then

$$
\begin{array}{llll}
\text{begin}(a_0). & \text{cong}(a_0, X) \leftarrow X \equiv_{p_0} 0 & \ldots & \text{cong}(a_0, X) \leftarrow X \equiv_{p_0} (p_0 - 1) \\
\text{edge}(a_1,a_2). & \text{cong}(a_1, X) \leftarrow X \equiv_{p_1} 0 & \ldots & \text{cong}(a_1, X) \leftarrow X \equiv_{p_1} (p_1 - 1) \\
\vdots & \vdots & & \vdots \\
\text{edge}(a_{n-1},a_n). & \text{cong}(a_{n-1}, X) \leftarrow X \equiv_{p_{n-1}} 0 & \ldots & \text{cong}(a_{n-1}, X) \leftarrow X \equiv_{p_{n-1}} (p_{n-1} - 1) \\
\text{end}(a_n). & & &
\end{array}
$$

```
go(N,X):- end(N).
go(N,X):- cong(N,X), edge(N,M), go(M,X).
query(X):- start(N), go(N,X).
```

Figure 5.2: Datalog$^{\equiv Z}$ program with unrestricted number of modulo factors.

Example 5.31 Consider the Datalog$^{\equiv Z}$ program in Figure 5.2. Then the query $query(X)$ produces the set of constraint tuples

$$\{X \equiv_P i : 0 \leq i < P\}$$

where $P = p_0 \cdot p_1 \cdot \ldots \cdot p_{n-1}$. The size of this set is clearly exponential in the size of the input database as $n + \sum_{i=0}^{n-1} p_i < (n+1)p_{n-1}$.

The fact, that the program generates all the constraints, follows immediately from the following proposition:

Proposition 5.32 (The cyclic primary decomposition theorem [10])
Any finite Abelian group is isomorphic to a finite product of cyclic groups of prime power orders, and the list of the prime power orders is unique up to permutation.

and the fact that the growth rate of the sequence of primes is approximately $n \log n$ [82]; every element of Z_P is isomorphic to the product of the appropriate elements in the individual groups Z_{p_i}. These elements are represented by the constraints $X \equiv_{p_i} j$ and any particular element of the group Z_P is uniquely determined by the conjunction of such constraints over all p_i.

5.4 Combining Classes of Constraints over Integers

Periodicity constraints over integers are clearly not the only interesting class of constraints. In this section we consider the possibility of combining different classes of constraints over the same domain, in our case integers.

5.4.1 Gap-order constraints

First we combine the constraint language developed so far with Datalog$^{<Z}$—Datalog with gap-order constraints [73, 75].

Definition 5.33 (Gap-order constraint) *Let u, l be integers, c a nonnegative integer, and $x, y, \ldots$ be variables over integers. Then a finite conjunction of formulas of the form $l < x$, $x < u$, and $x + c < y$ is called a* gap-order constraint.

Conjunctions of gap-order constraints can be efficiently represented using *gap graphs* [73] (directed acyclic graphs where nodes represent variables and the lower and the upper bounds of constraints, and directed labeled edges represent gaps, i.e., the minimal integer distances between nodes). For this representation we also have all the necessary operations for closed-form bottom-up evaluation [73].

We show that the combination of these two approaches still has a closed-form evaluation procedure and the complexity bounds do not increase.

Definition 5.34 (Datalog$^{\equiv Z,<Z}$ **programs)** *Let P be a finite set of clauses of the form*

$$A \leftarrow B_1, \ldots, B_k, C_1, C_2$$

where $A, B_1, \ldots, B_k$ are atoms, C_1 is a satisfiable periodicity constraint in $\mathcal{C}_K$, and C_2 is a satisfiable gap-order constraint. Then P is a Datalog$^{\equiv Z,<Z}$ *program.*

Note that we cannot immediately apply the technique from Section 4.4 as both the periodicity and the gap-order constraints are over the same domain—integers.

5.4.2 Complexity of consistency checking

Consistency (satisfiability) checking of periodicity graphs and gap-order graphs (separately) can be done in polynomial time with respect to the size of the graph (cf. Section 5.2 and [73, 75]). However,

Theorem 5.35 *Consistency checking of a conjunction of a periodicity graph with a gap graph is NP-complete.*

P r o o f: By reduction of 3SAT to satisfiability of conjunctions of constraints of the form $x \equiv_k (y + c)$ and $x + c < y$. Let S be an instance of 3SAT over variables $x_1, \ldots, x_k$ and clauses $c_1, \ldots, c_n$. We encode S by a quantifier-free conjunction of periodicity and gap-order constraints as follows:

Encoding of variables x_i: For every variable x_i we create a formula

$$\delta(x_i) = (z_i \equiv_2 z_i^* + 1) \wedge (-1 < z_i < 2) \wedge (-1 < z_i^* < 2)$$

Encoding of clauses c_i: Let $c_i = l_i^1 \vee l_i^2 \vee l_i^3$. We define a formula

$$\gamma(c_i) = (y_i \equiv_2 t_i^1) \wedge (y_i \equiv_3 t_i^2) \wedge (y_i \equiv_5 t_i^3) \wedge (0 < y_i < 30)$$

where $t_i^j = z_k$ if $l_i^j = x_k$ and $t_i^j = z_k^*$ if $l_i^j = \neg x_k$ for $0 < j \leq 3$.

The whole instance S is then reduced to the formula

$$\varphi(S) = \bigwedge_{0<i\leq k} \delta(x_i) \wedge \bigwedge_{0<i\leq n} \gamma(c_i)$$

It is easy to see that a satisfying assignment of values to the free variables of $\varphi(S)$ yields a satisfying assignment to S, where x_i is true if and only if $z_i = 1$.

```
consistent(G,H) =
  if acyclic(G)
    then
      for every path l -e0-> x1 -e1-> ... -e(n-1)-> xn -en-> u do
        p := l + α(e0)
        for i := 1 to n do
          p := ⌈p⌉_xi
          p := p + α(ei)
        od
        if p > u return false
      od
      return true
    else
      return false
```

where $\alpha(e)$ is the value of the gap associated with the edge E in G, l and u are the lower and upped bounds of G, and $\lceil p \rceil_{x_i}$ is the *rounding* up to the smallest integer p' greater than or equal to p and satisfying the constraint $p' \equiv_{\alpha_H(x_i)} \beta_H(x_i)$. The searching through all the paths from l to u can be efficiently achieved using a dynamic programming technique.

Figure 5.3: Consistency checking for conjunctions of simple periodicity and gap constraints.

Fortunately:

Theorem 5.36 *Consistency checking of a conjunction of a periodicity graph constructed from simple periodicity constraints only and a gap graph can be computed in PTIME.*

P r o o f: By generalization of the consistency checking procedure given in [73]; the algorithm computes the length of every path from the lower bound to the upper bound of the gap-graph by adding the gap sizes on the edges of the path. Moreover, for every node along the path, the length of the partial path from the lower bound to this node is rounded up with respect to the periodicity constraint associated with this node. The graph is consistent iff the length of every path is shorter than the difference between the lower and upper bounds in the gap-graph. Figure 5.3 gives an algorithm for consistency checking of such conjunctions.

Limiting the periodicity constraints to the form $x \equiv_k c$ does not reduce the expressive power of the query language. All the constraints of the form $x \equiv_k (y+c)$ can be defined using the following Datalog$^{\equiv Z}$ clauses:

$$\begin{array}{lcll} x \equiv_k (y+c) & :- & x \equiv_k 0 & \wedge \;\; y \equiv_k c \\ x \equiv_k (y+c) & :- & x \equiv_k 1 & \wedge \;\; y \equiv_k (c+1) \\ & \vdots & & \\ x \equiv_k (y+c) & :- & x \equiv_k (k-1) & \wedge \;\; y \equiv_k (c-1) \end{array}$$

In a similar way, we can also represent constraints of the form $x \not\equiv_k y + c$. The periodicity graphs become much simpler and smaller in this limited case; especially the normalization is not needed anymore. The disadvantage of this solution is that we need to store more rules in the database.

Moreover, the overall complexity of the bottom-up evaluation is exponential in the *arity* of predicate symbols in the program. Thus even using a NP-complete consistency checking procedure does not have an impact on the overall data complexity. However, we consider only simple periodicity constraints in the rest of the chapter.

5.4.3 Bottom-up evaluation

Now we show that while we cannot compose the two classes over integers immediately, we can *reuse* a large portion of the algorithms developed so far:

1. we can reuse the constraint conjunctions and subsumptions for the individual classes, but

2. we have to develop a new constraint projection operation and the satisfiability check.

A $(\equiv, <)$-interpretation is a set of triples (A, G, H) where A is a predicate symbol, G is a periodicity graph, and H is a gap graph, respectively. A, G, and H must have the same arity.

In the definition of the consequence operator $\mathrm{TP}_{\equiv,<}$ we can reuse the conjunction and subsumption procedures defined for the component classes of constraints. We can also use the consistency check from Theorem 5.36. However, the most complex operation—the quantifier elimination—has to take care of the interactions of the periodicity constraints with the gap-order constraints [105].

Example 5.37 Assume that we want to eliminate quantifier $\exists y$ from the constraint:

$$\exists y.x + c_1 < y \wedge y + c_2 < z \wedge y \equiv_k d$$

Clearly we cannot replace it simply by $x + c_1 + c_2 + 1 < z$ as we did in the case of gap-order constraints only: we need to take into account the periodicity constraint $y \equiv_k d$, i.e., we need to make sure that there is at least one integer of the form $\{d + nk\}$ between $x + c_1$ and $z - c_2$. Thus, the equivalent quantifier-free formula is

$$\begin{array}{lcll}
(x + c_1 + 1) \equiv_k d & \wedge & x + c_1 + c_2 + 1 < z & \vee \\
(x + c_1 + 2) \equiv_k d & \wedge & x + c_1 + c_2 + 2 < z & \vee \\
 & \vdots & & \\
(x + c_1 + k) \equiv_k d & \wedge & x + c_1 + c_2 + k < z &
\end{array}$$

It is easy to see that the variable y was successfully eliminated and the resulting constraint is a disjunction of conjunctions of periodicity and gap-order constraints.

This idea is used for the projection operation needed in the $\mathrm{TP}_{\equiv,<}$ operator; a similar idea was used in [55]; our procedure is simpler due to a different representation of constraints, especially we can omit the normalization (in the sense of [55]) of constraint tuples[3].

[3] The normalization in [55] is different from Definition 5.12 and requires *whole relations* to be normalized

```
qe(y,(G, H)) =
  if indegree(y) = 0
    then
      return {(G − {y}, H − {y})}
    else
      let x be a node such that (x, y) ∈ E_H
      and n be the number of nodes, such that (y, z_i) ∈ E_H in
        for all 0 < i ≤ n do
          create gap graph H_i such that E_{H_i} = E_H − {(x, y)} ∪ {(x, z_i)}
            where α_{H_i}(x, z_i) := α_H(x, y) + α_H(y, z_i) + i
          create simple periodicity graph G_i such that
            G_i := G ∧ G_{x ≡_k (α_G(y) − α_H(x,y) − i)}
        return ⋃_i (if (G_i, H_i) is consistent then qe(y,(G_i, H_i)) else {})
```

Figure 5.4: Pseudocode for Algorithm 5.38.

Algorithm 5.38 (Projection) Let G be a periodicity graph, H a gap-graph, y a node in both G and H, $x \in N_H$ such that $(x, y) \in E_H$ is labeled with c (i.e., $x + c < y$), $z_1, \dots, z_l \in N_H$ such that $(y, z_j) \in E_H$ is labeled with c_j for $0 < j \leq l$, and $y \equiv_k d \in G$. We form a set of graphs (G_i, H_i) for $0 < i \leq k$ by modifying G and H as follows:

1. delete the edge (x, y) ,
2. insert edges (x, z_j) with the label $c + c_j + i$,
3. add the constraint $x \equiv_k (d - c - i)$,
4. check the resulting pair (G_i, H_i) for consistency and discard inconsistent pairs.

We apply this transformation recursively on each of the consistent pairs (G_i, H_i) until all the edges ending in y are eliminated from the gap graphs. Then we remove node y from all the resulting pairs of graphs, including all incident edges.

Lemma 5.39 *Let $\{(G_i, H_i) | i \in I\}$ be the set of graphs produced by algorithm 5.38 from the pair (G, H) with respect to a variable y. Then for all valuations θ:*

$$\theta \models \bigvee_{i \in I} (G_i, H_i) \iff \exists a \in Z.\theta[a/y] \models (G, H)$$

P r o o f: Immediate from Example 5.37 and [105], as the resulting pairs (G_i, H_i) represent exactly the consistent disjuncts in the DNF of the quantifier-free formula.

Thus, algorithm 5.38 defines a projection operation for the periodicity graphs combined with the gap graphs. To eliminate all the existential quantifiers we apply this algorithm recursively to its own output until all the existential quantifiers are eliminated from the original pair of gap- and periodicity graphs.

The projection operation produces possibly fairly large set of graphs from a single graph. This is often considered a space complexity problem. However, in the framework of Datalog it is not true: even in the standard Datalog, each relation is represented as a collection of its tuples. In our case we still manage to represent often infinite sets of tuples by a single element. The consequence operator is now defined in the standard way:

Definition 5.40 *Let P be a* Datalog$^{\equiv Z,<Z}$ *program.*

$$\begin{array}{rl} \mathrm{TP}_{\equiv,<}(I) = \{ & (A,G,H) : A \leftarrow B_1,\ldots,B_m,C_1,C_2 \in P \\ & (B_i,G_i,H_i) \in I \textit{ for all } 0 < i \leq m \\ & G' = G_1 \wedge \ldots \wedge G_m \wedge G_{C_1} \\ & H' = H_1 \wedge \ldots \wedge H_m \wedge H_{C_2} \\ & (G,H) \in \pi_{FV(A)}(G',H') \textit{ is consistent and not subsumed by } I\} \end{array}$$

where G_{C_1} is a periodicity graph corresponding to C_1 and H_{C_2} is a gap graph corresponding to C_2.

We can show that the bottom-up evaluation of a Datalog$^{\equiv Z,<Z}$ program using algorithm 4.10 (using $\mathrm{TP}_{\equiv,<}$ instead $\mathrm{TP}_{\equiv}$) is correct with respect to the standard bottom-up evaluation:

Theorem 5.41 *For any* Datalog$^{\equiv Z,<Z}$ *program P and any predicate symbol R in P*

$$\begin{array}{rll} R(c_1,\ldots,c_k) \in \mathrm{TP}^{\omega}(\emptyset) \iff (R,G,H) \in \mathrm{TP}^{\omega}_{\equiv,<}(\emptyset) & \wedge & [c_1/x_1,\ldots,c_k/x_k] \models G \\ & \wedge & [c_1/x_1,\ldots,c_k/x_k] \models H \end{array}$$

P r o o f: By simultaneous induction on the number of iterations of TP. Base case holds trivially.

Let $A(c_1,\ldots,c_k) \in \mathrm{TP}^{i+1}(\emptyset)$. Then there exists clause $A \leftarrow B_1,\ldots,B_k,C_1,C_2 \in P$ and a (ground) substitution θ such that $A\theta = A(c_1,\ldots,c_k)$, $B_i\theta \in \mathrm{TP}^i(\emptyset)$, $C_1\theta$ and $C_2\theta$ hold. Then by IH $(B_i,G_i,H_i) \in \mathrm{TP}^i_{\equiv,<}(\emptyset)$ where $\theta \models G_i$ and $\theta \models H_i$. Also, there exists a periodicity graph G_{C_1} equivalent to C_1 and a gap graph G_{C_2} equivalent to C_2. such that $\theta \models G_{C_1}$ and $\theta \models G_{C_2}$. Thus $(A,G,H) \in \mathrm{TP}^{i+1}_{\equiv,<}(\emptyset)$ by definition of the $\mathrm{TP}_{\equiv,<}$ operator.

The opposite direction is similar. The claim holds by observation that $\mathrm{TP}_{\equiv,<}$ is continuous[4].

Note that while the proof of this theorem is similar to the proof of Theorem 4.19, it is not a special case as the domains of periodicity and gap-order constraints are not disjoint.

Let I be a $(\equiv,<)$-interpretation. Then we define

$$\pi_{\equiv}(I) = \{(A,G) : (A,G,H) \in I\} \text{ and } \pi_{<}(I) = \{(A,H) : (A,G,H) \in I\}.$$

Lemma 5.42 *Let P be a* Datalog$^{\equiv Z,<Z}$ *program. Then*

$$\pi_{\equiv}(\mathrm{TP}^i_{\equiv,<}(\emptyset)) \subseteq \mathrm{TP}^i_{\equiv}(\emptyset) \textit{ and } \pi_{<}(\mathrm{TP}^i_{\equiv,<}(\emptyset)) \subseteq \mathrm{TP}^i_{<}(\emptyset)$$

for all $0 \leq i$.

P r o o f: By induction on i.

[4]The continuity of $\mathrm{TP}_{\equiv,<}$ is proven similarly to continuity of TP_C in Lemma 4.9.

Corollary 5.43 *The union of the classes of the periodicity and the gap-order constraints is constraint-compact.*

Corollary 5.44 *The bottom-up evaluation of any* Datalog$^{\equiv Z,<Z}$ *program terminates.*

To show a polynomial bound for the tuple recognition procedure for Datalog$^{\equiv Z,<Z}$ programs we use the same technique: the tuple recognition runs in PTIME (follows from Theorem 5.29 and complexity of the TEST algorithm in [73])[5].

5.4.4 Equality constraints

So far we have not mentioned the most natural class of constraints on integers—the equality constraints, which are needed to include the "standard" Datalog over integers. In [73] the equality constraints are handled by adding extra information to the gap-graphs. In the case of Datalog$^{\equiv Z}$ it can be done in the same way; in all cases we need some mechanism to represent a single constraint of the form $x = y$.

For constraint classes closed under conjunction we can use following lemma:

Lemma 5.45 *Let φ be a constraint where x and y are free variables. Then*

$$\varphi \wedge (x = y) \iff \varphi \wedge \varphi[x/y, y/x]$$

After applying this lemma we can eliminate either x or y using the quantifier elimination procedure described in the previous section. If $x = y$ is the only constraint we can just ignore it.

5.5 Conclusion

We have described the bottom-up evaluation procedures for several versions of Datalog enhanced by the constraints over theintegers and provided a general way for combining various classes of such constraints into a single language. We also proved the complexity bounds for the bottom-up evaluation algorithm.

Further directions of research:

Extensions: negation, negative gap-order constraints. The gap-order constraints allows us to specify the minimal distance between two variables. The maximal distance constraint, e.g., the maximal waiting time for changing planes in Example 5.1, can be expressed in a very limited way using an enumeration technique similar to the technique used in algorithm 5.38. Note that a general solution that adds gap order constraints with negative size of the gap (i.e., $x - 2 < y$) leads immediately to an undecidable theory—we can define the successor relation over integers as "$\mathrm{succ}(x, y) :- \ y - 2 < x, x < y$".

[5] The degree of the polynomial is higher than in standard Datalog—$O(a^2)$ for atoms of arity a (exactly $(a+1)(a+2)/2$ for Datalog$^{<Z}$ and $2a+2$ for Datalog$^{\equiv Z}$). This is because the values are assigned to *pairs* of attributes rather than to single attributes. In both cases the *exponentiated* expression is a constant with respect to the size of the program.

For the same reason it is not possible to extend the language with standard stratified negation. There is a limited version of negation that can be included in the language without such penalty. In [77] a syntactic subclass of Datalog$^{\neg,<Z}$ was introduced, for which termination can be shown. The main idea is that only unary constraints can be passed through negations—this effectively prevents us from defining the successor relation, as all the constraints, derived during the evaluation of the program are gap-order constraints with a non-negative gap-value.

Note also that for periodicity constraints alone negation is not a problem: the constraint of the form $x \not\equiv_k y + c$ can be introduced without any problems (cf. Section 5.3).

Expressiveness. In [12] the expressiveness of a number of deductive and constraint query languages is discussed. However, only monadic programs are considered. It is interesting to see, whether the expressiveness of query languages defined in this section can also be formally characterized.

Efficient Implementation. Periodicity constraints define non-convex sets. In this respect they differ from most common constraint languages. [60] describes how to adapt interval management techniques for the use with constraint databases. However, this approach works only for convex sets. Thus, the periodicity constraints call for new storage management techniques.

5.6 Related Work

There are other proposals that introduce classes of infinite integer relations into query languages. Datalog with (gap-)order constraints was introduced in [73]. In [55] the relational calculus over generalized relational databases with order constraints and linear repeating points was studied (a linear repeating point has the form $\{c + kn | n \in Z\}$ for some fixed integers c and k and is just a different notation for a periodicity constraint). However, the proposed approach was limited to first order queries and thus simple inductive queries such as the transitive closure were not expressible. It is not clear how to adapt this approach to Datalog because of no obvious termination argument. Moreover, our solution tries to keep the different classes of constraints separate as long as possible (the only 'meeting' points are the quantifier elimination and the consistency checking procedures). This arrangement provides a potentially more general framework, which could be adapted to combinations of other constraint languages over the integers.

Another approach comes from the area of temporal databases. In [23, 24] Datalog was extended with a limited use of the successor function symbol (the use is restricted to a single distinguished argument in each literal). This way it is possible to represent infinite periodic sets of integers using Horn rules. But it is not clear how to add constraints to this language and maintain its computational properties. In particular, the order constraints are not expressible in this language. Also, the unary successor symbol is used in the deductive layer on top of the database. Our approach allows a representation of the infinite relations to be stored directly in the database. Another extension of Datalog was proposed in [13]. This extension combines linear repeating points, order constraints, and unlimited use of successor function. The resulting language is very expressive—termination of query evaluation in this language cannot be guaranteed.

Summarizing: [73, 75] cannot handle periodicity constraints, [55] cannot handle recursion, [23, 24] cannot handle ordering, and [13] does not guarantee termination. Our language is thus another step towards a tractable and expressive query language for temporal databases.

Bibliographical Notes

The work on the combination of Datalog rules with integer order constraints was pursued in [73]. Integer based constraint queries have also been proposed in [55]. Most of the results about integers can be easily found in books on number theory, e.g., [48, 104] or [82] (if you read Polish). Other useful results about cyclic groups Z_n can be found in books on group theory and modern algebra in general, e.g., [10].

Chapter 6

Other Constraint Classes

This section surveys several other known constraint classes, and summarizes their properties. We show that for the majority of the constraint classes the semantics can be defined by a fixed point iteration of the TP operator, and thus we can apply the general theorems of Chapter 4.

6.1 Constraint Classes

This section surveys constraint formalisms based on the notion of the constraint class, and thus allows the application of the bottom-up evaluation algorithm based on the immediate consequence operator TP. This approach also allows the immediate use of the efficient evaluation methods for constraint databases developed in Chapter 7.

Constraint-compact classes

- The standard Datalog can be defined using the class of constraints generated from the set $\{x = a : a \in A\}$ where A is the set of all constants in the Datalog program (cf. Example 4.15).

  Allowing general equality may cause problems to the standard evaluation strategies (rules may be non-range-restricted). However, in our case we simply generate the appropriate class of constraints from the set $\{x = a : a \in A\} \cup \{x = y\}$. The evaluation remains unchanged as we use more general evaluation mechanism.

- The incorporation of more interesting constraints, e.g., the constraints over integers Z, has been discussed in Chapter 5: the *gap-order* constraints [73] are generated from the set

  $$\mathcal{C}_{<Z} = \{x < u : u \in Z\} \cup \{u < x : u \in Z\} \cup \{x + c < y : c \in Z^+\}$$

  Similarly the *periodicity* constraints are generated from

  $$\mathcal{C}_{\equiv Z} = \{x \equiv_k c : c < k, k \in \mathcal{C}_0\}$$

In Chapter 5 and [96] a closed form bottom-up evaluation procedure for

$$\mathcal{C}_{\equiv,<Z} = \mathcal{C}_{<Z} \cup \mathcal{C}_{\equiv Z}$$

was developed including all the necessary operations on the constraint class as required by Definition 4.3.

- We can also incorporate the *dense order* constraints over Q by a slight modification of constraint operations well defined in [60, 58].
- The set-order constraints were introduced in [76], and
- The boolean algebra constraints in [60],

All the above constraint classes are constraint-compact.

Other constraint classes. There are also constraint classes, where all the constraint operations are defined, but which are *not* constraint-compact, e.g., the class generated from the set $\{x + c < y : c \in Z\}$—gap-order constraints with possibly negative size of the gap [73] or the linear arithmetic constraints [60].

6.2 Abstract Interpretation

Another interesting application of constraints can be found in the area of programming languages analysis, especially those analysis that are based on the *Abstract Interpretation* [3, 32]. The main idea behind this technique is to execute a given program with respect to its formal semantics and a set of *abstract values*, rather than with respect to the concrete values the program is designed to work on. The domains of the abstract values are much simpler and often finite. Moreover, all the domains of the abstract values are designed in a way that the analysis always terminates.

The semantics of programming languages can be given in the natural form given in, e.g., [47, 80], using a set of inference rules. It is easy to see the connection between the abstract interpretation and deductive queries over a constraint database: instances of the semantic rules for every syntactic construction in the analyzed program form a set of recursive Datalog rules (cf. figure 6.2) over the appropriate domain of abstract values. The goal of the abstract interpretation is to find a (minimal) solution to these rules, i.e., to evaluate queries over a constraint database.

Note also that the domains of abstract values are always constraint-compact to guarantee termination of the analysis. The advantage of using the constraint approach is that it provides the correctness proofs of the whole analysis (provided we have a correct abstractions for the atomic values and operations).

This approach can be applied to many techniques used for compiler optimization, e.g., the *data-flow analysis* [4] or the *closure analysis* [81].

6.3 Other Formalisms

There are few other extensions of Datalog with constraints. These extensions and their evaluation algorithms are not based on fixed point iteration, and thus these formalisms cannot be directly handled using the techniques, developed through this dissertation.

- Datalog_{1S}, Datalog_{nS}: This extension of Datalog is geared towards a limited use of function symbols (1 successor, n successors). The use of the successor symbols is confined to a single distinguished argument in the individual literals of the program. Such arrangement guarantees termination of all queries [23].
- Datalog over sequences [66]. This paper introduces an extension of Datalog with *interpreted* function symbols for manipulating sequences. A syntactic restriction on the class of queries guarantees termination of query evaluation.

Bibliographical Notes

The individual results for the various constraint classes can be found using the references in Figure 6.1. A forthcoming survey of constraint query languages [78] from (which figure 6.1 was borrowed) discusses the individual results in detail.

Domain	Constraints	Relational Calculus	Datalog	Datalog$^{\neg}$
D	$=$	AC_0 [1] [58] LOGSPACE [1] [17]	PTIME [1,4] [5, 17]	PTIME [1,4] [6, 19]
Q	$\neq, \leq, <$	AC_0 [58] LOGSPACE [60, 36]	PTIME [4] [60]	PTIME [2,4] [60]
Z	$\neq, \leq, <, <_k$	NC [74]	in DEXPTIME [75] DEXPTIME-hard [76]	non-elem. [3] [77] undec. [75]
N, Z	$\equiv_k$	PTIME [105] ($\exists$ only) DEXPTIME	PTIME [4,5] [96] DEXPTIME	PTIME [4,5] DEXPTIME
B_m	$=_{B_m}$	PTIME ($\exists$ only) [60]	Π_2^p-hard [60]	undec.
$P(D)$ $P(Z)$	$\subseteq, c \in, c \notin$	PTIME ($\exists$ only) [87]	DEXPTIME-comp [76]	undec.
N, Z	$1S$	NC [74]	PSPACE-comp [23]	PSPACE-comp [23]
R	$<, +$	NC [37]	undec.	undec.
N, Z	$<, +$	in $O(2^{n^{cn}})$ [71, 11]	undec.	undec.
N, Z	lrp	in PTIME [55]	undec.	undec.
R	$+, *$	NC [60, 14, 72]	undec.	undec.
Z	$+, *$	undec. [46]	undec.	undec.

[1] safe queries [100]; the answers have to be finite relations.
[2] inflationary semantics.
[3] syntactically restricted stratified Datalog$^{\neg}$; cf. Section 5.5
[4] data complexity.
[5] for a fixed set of modulo factors (cf. Chapter 5).

This table summarizes the combined complexity of query evaluation for various classes of constraints and query languages.

Figure 6.1: Summary of Constraint classes and query evaluation [78].

We use a simple language to demonstrate the use of constraints for abstract interpretation:

$$C ::= I := J \mid C; C \mid \ \texttt{skip} \ \mid \texttt{if } B \texttt{ then } C \texttt{ else } C \mid \ \texttt{while } B \texttt{ do } C \texttt{ od}$$

where B stands for boolean expressions of the form $I < J$, $I \leq J$, or a boolean variable. To give meaning to this syntax, consider (a fragment of) the natural semantics for C:

$$\frac{\rho_0 \vdash B \Downarrow \texttt{tt} \qquad \rho_0 \vdash C_1 \Downarrow \rho_1}{\rho_0 \vdash \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \Downarrow \rho_1} \qquad \frac{\rho_0 \vdash B \Downarrow \texttt{ff} \qquad \rho_0 \vdash C_2 \Downarrow \rho_1}{\rho_0 \vdash \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \Downarrow \rho_1}$$

$$\frac{\rho_0 \vdash B \Downarrow \texttt{ff}}{\rho_0 \vdash \texttt{while } B \texttt{ do } C \Downarrow \rho_0} \qquad \frac{\rho_0 \vdash B \Downarrow \texttt{tt} \qquad \rho_0 \vdash C \Downarrow \rho_1 \qquad \rho_1 \vdash \texttt{while } B \texttt{ do } C \Downarrow \rho_2}{\rho_0 \vdash \texttt{while } B \texttt{ do } C \Downarrow \rho_2}$$

(we have shown only a fragment of the whole semantic definition that allows us to proceed with our example). The sequents in the rules have the form $\rho \vdash C \Downarrow \rho'$ where ρ and ρ' are the storage vectors before and after executing C (for commands).
Consider a procedure that naively sorts the values in three variables x, y, z:

```
procedure sort3(x, y, z) =
   f :=tt;                                        (asg1)
   while f do                                     (while)
      f :=ff                                      (asg2)
      if x ≤ y then f, x, y :=tt, y, x else skip; (if1, asg3)
      if y ≤ z then f, y, z :=tt, z, y else skip; (if2, asg4)
```

where the labels in the right column are labels of the individual syntactic constructions. The individual statements of *sort*3 are used to instantiate the individual semantic rules (written as Datalog clauses, after unfolding the rules for sequential composition):

```
sort3(X0,Y0,Z0,X1,Y1,Z1) :- asg1(_,F0), while(F0,X0,Y0,Z0,_,X1,Y1,Z1).

while(F0,X0,Y0,Z0,F0,X0,Y0,Z0) :- F0 = ff.
while(F0,X0,Y0,Z0,F4,X3,Y3,Z3) :- F0 = tt,
      asg2(F0,F1), if1(F1,X0,Y0,F2,X1,Y1), if2(F2,Y1,Z0,F2,Y2,Z2),
      while(F2,X1,Y2,Z2,F4,X3,Y3,Z3).

if1(F0,X0,Y0,F1,X1,Y1):- X0=<Y0, asg3(X0,Y0,F0,X1,Y1,F1).
if1(F0,X0,Y0,F0,X0,Y0):- X0>Y0.

if2(F0,Y0,Z0,F1,Y1,Z1):- Y0=<Z0, asg4(Y0,Z0,F0,Y1,Z1,F1).
if2(F0,Y0,Z0,F0,Y0,Z0):- Y0>Z0.

asg1(_,tt).
asg2(_,ff).
asg3(X0,Y0,_,Y0,X0,tt).
asg4(X0,Y0,_,Y0,X0,tt).
```

This is a Datalog$^{<Z}$ program and we can ask queries with respect to this program, e.g., the query $sort3(X_0, Y_0, Z_0, X_1, Y_1, Z_1)$ such that $X_0 > Y_0 > Z_0$ gives the answer $X_1 < Y_1 < Z_1$. Thus in the execution of *sort*3 with the storage vector satisfying the constraint $X_0 > Y_0 > Z_0$ always produces an output, such that $X_1 < Y_1 < Z_1$.

Figure 6.2: Application of Constraint Queries to Abstract Interpretation.

Chapter 7

Evaluation Methods

This chapter proposes an efficient method for evaluating deductive queries over general constraint databases. The method is based on a combination of top-down resolution with memoing and closed form bottom-up evaluation. In this way top-down evaluation terminates for all queries for which the bottom-up evaluation also terminates. The main advantage of the proposed method is the direct use of the information, present in the partially instantiated queries without the need for rewriting of the original program. The evaluation algorithm automatically propagates all the necessary constraints during the computation. In addition, the top-down evaluation allows the use of compilation techniques, developed for compilers of logic programming languages, which can make the query evaluation very efficient.

7.1 Introduction

We propose a new method for evaluating deductive queries over constraint databases. (cf. Chapter 4 and [60]). The evaluation of queries in constraint databases is different from the one used in standard database systems. The constraints are used as the actual representation of the data stored in the database rather than as mere restrictions of the contents of otherwise ground relations. Algorithms for query evaluation over constraint databases have to satisfy the following criteria:

1. The evaluation algorithm should terminate on all possible input queries.
2. The constraint language should encompass a wide class of constraints over particular domains.
3. Partially instantiated queries have to be evaluated efficiently.

The first requirement is especially difficult to achieve in the case of constraint databases because the extensions of constraint relations are usually infinite. There are two main approaches to satisfy the above requirements in the case of Datalog. However, neither of them seems to address all three requirements.

The first approach is based on the fixpoint (bottom-up) evaluation of the rules. (cf. Chapter 4). Here the first condition is usually met, e.g., for Datalog [100], Datalog with integer constraints in Chapter 5 and [73, 96], and sets [85]. However, the classes of allowed constraints are restricted to guarantee termination. Also, the evaluation process is not *goal-oriented* and thus the evaluation of the partially instantiated queries is quite inefficient[1].

The other approach is based on top-down resolution-based method. Here the second and third conditions are usually met. However, termination for all queries is often sacrificed (an exception is [90], where no constraints are allowed) in order to improve the expressiveness and efficiency. These methods can also take full advantage of compilation techniques developed for other logic programming languages, e.g., [91, 103]. This greatly improves the practical efficiency of query evaluation [90] in the case of Datalog. We show that similar results can be achieved in the case of constraint databases as well.

In this chapter we try to combine the advantages of both the approaches. We propose an evaluation method, *Constraint Memoing*, applicable to constraint-based extensions of Datalog (Datalog$^{\mathcal{C}}$, where $\mathcal{C}$ stands for the chosen class of constraints; cf. Chapter 4)) that has the following features:

- Termination. Constraint Memoing guarantees termination of queries for all classes of constraints that have a closed-form terminating bottom-up evaluation procedure. Also, the complexity bounds of the bottom-up procedure are preserved.
- The expressiveness of the language can be easily extended to accommodate various classes of constraints as long as every class of constraints is equipped with several elementary operations on the underlying representation of the constraints. This step is quite subtle, if the termination of queries is to be preserved. In contrast to bottom-up methods, it is also possible to extend the query language to classes of constraints, where termination is not guaranteed[2].
- The use of a top-down method allows a fully goal-oriented query evaluation, i.e., the information present in the partially instantiated queries is used to prune the search space. The efficiency achieved by this method is better than the efficiency of comparable bottom-up methods including program rewriting techniques (e.g., Magic Set Transformation; cf. Section 7.4). Also, in the future, the top-down evaluation strategy will allow a direct use of compilation techniques developed in the area of Logic Programming.

The rest of the chapter is organized as follows: Section 7.2 presents the main ideas behind the goal-oriented evaluation procedures. Section 7.3 describes the proposed evaluation method, Constraint Memoing, includes the soundness, completeness, and termination proofs, and possible optimization techniques that are specific to the processing of constraint deductive queries. Section 7.4 introduces a general Magic Set Transformation (MST$^{\mathcal{C}}$) for Datalog$^{\mathcal{C}}$ for comparison purposes. Section 7.5 studies both the analytical complexity of query evaluation using Constraint Memoing and gives results that show empirical evidence of the practicality of the proposed evaluation method. Section 7.6 concludes the presentation with possibilities of further improvements and directions for further research.

[1] Application of program transformation techniques does not solve this problem in general.

[2] Even in cases when termination cannot be guaranteed, the algorithm reduces the possibility of non-termination.

7.2 Goal-oriented Evaluation Strategies

There are several standard improvements to the naive bottom-up query evaluation algorithm, e.g., the semi-naive Algorithm [100]. However, such strategies fail to consider the information contained in a partially instantiated query (i.e., they are not goal-oriented). There are two major approaches to solving this problem in the framework of standard Datalog:

1. Rewrite the program using the Magic Set Transformation (MST) [8, 68] and subsequently evaluate the transformed program bottom-up, and

2. Adopt a top-down evaluation strategy [29, 92] based on the resolution principle [65].

In this section we consider mainly the top-down resolution-based method. However, the MST transformation for the constraint deductive query languages is also introduced for comparison purposes. It is well-known that the standard top-down strategies, e.g., SLD-resolution [29], despite their efficiency, have a major drawback as query evaluation procedures: they lead to nonterminating computations even in the situations, where the bottom-up algorithms are guaranteed to terminate. Note also that breadth-first traversal of a SLD-tree does not guarantee termination in general.

The drawback is caused by the occurrence of infinite paths in SLD search trees. This has been observed in several papers, e.g., [92, 93] and an alternative to SLD-resolution was proposed under various names. The main idea consists of *remembering* sets of answers for already resolved subgoals. This method also guarantees termination for function-free logic programs [90]. We extend this method to *constraint* deductive queries while preserving the termination and complexity bounds of bottom-up evaluation.

7.3 Top-down evaluation for $\text{Datalog}^{\mathcal{C}}$

In Chapter 4 the bottom-up evaluation of $\text{Datalog}^{\mathcal{C}}$ was defined in terms of elementary operations over a specific class of constraints. This section shows how a top-down query evaluation procedure (SLG-resolution [92]) can be refined using the same operations to handle constraint queries. This approach allows us to build a very efficient top-down evaluation procedure for every class of constraints that has a closed form bottom-up evaluation procedure. Moreover, the termination property of the bottom-up algorithm is preserved.

The modification of SLG resolution for Constraint Memoing ($\text{SLG}^{\mathcal{C}}$ resolution) is defined by the following set of rewriting rules:

Definition 7.1 ($\text{SLG}^{\mathcal{C}}$ rewriting rules) *Let*

$$\mathsf{root}(G;C), \mathsf{body}(G;B_1,\ldots,B_k;C), \mathsf{goal}(G;B,C';B_2,\ldots,B_k;C), \text{ and } \mathsf{ans}(G,A)$$

where G is an atom, $B_1,\ldots,B_k$ are literals, and C, C', A are constraints in the class $\mathcal{C}$, be *nodes from which we build SLG-trees using the following rules:*

Action/Node | Children | Conditions

$\boxed{\text{Clause resolution}}$

$$\mathsf{root}(G;C) \begin{cases} \mathsf{body}(G;B_1^1,\ldots,B_{k_1}^1;C\wedge D^1) & \text{for all } 0<i\le l \text{ such that}\\ \qquad\qquad\vdots & G\leftarrow D^i,B_1^i,\ldots,B_{k_i}^i\in P\\ \mathsf{body}(G;B_1^l,\ldots,B_{k_1}^l;C\wedge D^l) & \text{and } C\wedge D^i \text{ is satisfiable}\end{cases}$$

$\boxed{\text{Query projection}}$

$$\mathsf{body}(G;B_1,\ldots,B_k;C) \begin{cases} \mathsf{goal}(G;B_1,C_1;B_2,\ldots,B_k;C) & \\ \qquad\qquad\vdots & \text{for all } C_i\in\exists_{B_1}.C\\ \mathsf{goal}(G;B_1,C_l;B_2,\ldots,B_k;C) & \end{cases}$$

$\boxed{\text{Answer propagation}}$

$$\mathsf{goal}(G;B,C';B_2,\ldots,B_k;C) \begin{cases} \mathsf{body}(G;B_2,\ldots,B_k;C\wedge A_1) & \text{for all } A_i\in\textsc{ans}(B,C'')\\ \qquad\qquad\vdots & \text{where } C'\le_{\mathcal{C}} C''\\ \mathsf{body}(G;B_2,\ldots,B_k;C\wedge A_l) & \text{and } C\wedge A_i \text{ is satisfiable}\end{cases}$$

$\boxed{\text{Answer projection}}$

$$\mathsf{body}(G;\ ;C) \begin{cases} \mathsf{ans}(G;A_1) & \\ \quad\vdots & \text{for all } A_i\in\exists_G.C\\ \mathsf{ans}(G;A_l) & \end{cases}$$

where G, B_i^j are atoms, $A_i, C, C', C'', C_i, D^i \in \mathcal{C}$, and $\textsc{ans}(B, C'')$ is the set of answers collected from the leaves of the SLG-tree rooted by (B, C''), as introduced in Notation 7.4.

A SLG-tree *is a tree built from a node* $\mathsf{root}(G;C)$ *by finite application of the above rules. A* SLG-forest *is a set of SLG-trees.*

Note that the *Answer propagation* and *Answer projection* rules have to *cooperate* every time a new answer[3] is produced in a specific SLG-tree, it has to be propagated to all the nodes that have already been resolved using answers from this particular tree. Also, the *Answer propagation* rule is responsible for creating new SLG-trees in the SLG-forest when no tree with a root node that subsumes the goal to be resolved can be found.

The main difference between SLG and SLG$^{\mathcal{C}}$ is in two additional rules: the *Query Projection* and the *Answer Projection*. The *Query Projection* rule is responsible for determining what are the goals to be resolved by answer resolution (i.e., what is the goal-constraint pair to be looked for among the already computed answers). The *Answer Projection* rule is responsible for storing the computed answers for a given goal for subsequent lookup and propagating them to the appropriate goal nodes. Note the essential use of constraint projection that allows to determine the relevant constraint for every atom.

Also, the SLG-resolution can handle negation using several additional rules. However, presence of negation together with constraints often leads to non-termination [73]. Thus, our current proposal allows only positive programs. Adding negation is briefly discussed in Section 7.6.

The SLG$^{\mathcal{C}}$ rewriting rules are used for the query evaluation as follows:

[3] I.e., an answer not subsumed by any answer in the same tree generated earlier.

Definition 7.2 *Let* $\mathrm{SLG}(G,C)$ *be the SLG-forest that is generated from the query* (G,C,P) *as follows:*

1. *create a SLG-forest containing a single tree* $\{\mathsf{root}(G;C)\}$.
2. *expand the leftmost*[4] *node using the rules in Definition 7.1 as long as they can be applied.*
3. *return the set* $\textsc{ans}(G,C)$ *as the answer to the query.*

Let $\textsc{slg}(G,C)$ *be a SLG-tree rooted by the node* $\mathsf{root}(G;C)$.

We also order the answers (i.e., the nodes $\mathsf{ans}(G;C)$) in the SLG-forest according to time when a particular answer is derived:

Definition 7.3 *Let* $\mathsf{ans}(G;A)$ *and* $\mathsf{ans}(G';A')$ *be leaves in the SLG-forest generated by the SLG-rules (cf. Definition 7.1). Then we say that* $\mathsf{ans}(G;A)$ is older than $\mathsf{ans}(G';A')$ *if the node* $\mathsf{ans}(G;A)$ *is generated before the node* $\mathsf{ans}(G';A')$.

Notation 7.4 *Let* $\textsc{ans}(G,C)$ *be the set of all* A *such that* $\mathsf{ans}(G;A) \in \textsc{slg}(G,C)$ *and if an older* $\mathsf{ans}(G;A') \in \textsc{slg}(G,C)$ *then* $A \not\leq A'$.

Lemma 7.5 *Let* $\mathsf{ans}(G;A)$ *be a leaf of the SLG-tree* $\textsc{slg}(G,C)$. *Then for every application of the answer propagation rule along the path* $\mathsf{root}(G;C) \to \ldots \to \mathsf{ans}(G;A)$ *if* $\mathsf{ans}(G';A')$ *was the answer propagated by the rule then* $\mathsf{ans}(G';A')$ *is older than* $\mathsf{ans}(G;A)$.

P r o o f: Immediate from the Definitions 7.1, 7.2 and 7.3.

Soundness and completeness of Constraint Memoing are proven by reduction to soundness and completeness of the bottom-up evaluation (cf. Algorithm 4.10).

Note that the set $\textsc{ans}(G,C)$ may not be unique and depends on the order in which the nodes $\mathsf{ans}(G;A)$ are generated. However, it is sufficient for our purposes that the set of valuations $\|\textsc{ans}(G,C)\|$ is unique.

Lemma 7.6 *Let* $\textsc{slg}(G,C)$ *be a SLG-tree and* ϑ *a valuation. Let* $\vartheta \in \|\textsc{ans}(G,C)\|$. *Then* $\vartheta \models C$.

P r o o f: By induction on the height of the SLG-tree $\textsc{slg}(G,C)$.

To prove correctness of the algorithm, we show that all the derived answers are also derived in the bottom-up computation:

Lemma 7.7 *Let* (G_0,C_0,P) *be a query. Then for every* $\textsc{slg}(G,C) \in \mathrm{SLG}(G_0,C_0)$ *and every valuation* ϑ

$$\vartheta \in \|\textsc{ans}(G,C)\| \supset G\vartheta \in \mathrm{TP}^{\omega}(\emptyset)$$

P r o o f: Induction on the "age" of answers:

[4] Assuming the SLG-trees are ordered in the same order they were created. However, the choice of selection strategy does not affect the soundness/completeness of the algorithm.

Let $\mathsf{ans}(G; C') \in \textsc{slg}(G, C)$ such that $\vartheta \models C'$. Then there is a path

$$\mathsf{root}(G; C) \to \mathsf{body}(G; B_1, \ldots, B_k; C_1) \to \ldots \to \mathsf{ans}(G; C') \in \textsc{slg}(G, C)$$

that starts with a *Clause Resolution* step using a clause $G \leftarrow D, B_1, \ldots, B_k \in P$ (cf. Figure 7.1). By Definition 7.1 $C' \in \exists_G.C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$ where A_i is an answer propagated from the SLG-tree $\textsc{slg}(B_i, C_i')$. Thus there exists θ an extension of ϑ, such that $\theta \models C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$. Clearly, $\theta \models A_i$ for $0 < i \leq k$. By the induction hypothesis (as all the answers $\mathsf{ans}(B_i; A_i)$ used along this path have been computed before $\mathsf{ans}(G; C')$ by Lemma 7.5) we have $B_i\theta \in \mathrm{TP}^\omega(\emptyset)$. By definition of TP and the fact that $\theta \models D$ we have $G\theta = G\vartheta \in \mathrm{TP}^\omega(\emptyset)$.

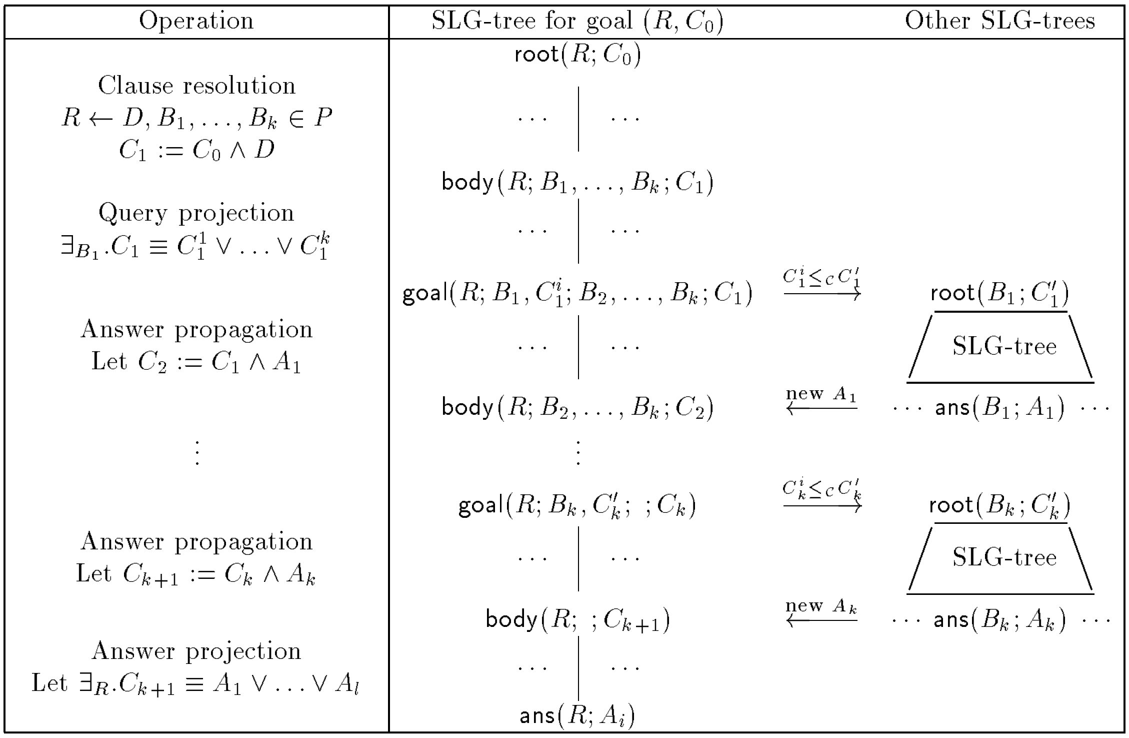


Figure 7.1: $\mathrm{SLG}^{\mathcal{C}}$ Evaluation of goal R w.r.t. constraint C_0.

Thus all answers—not only for the main query, but also for all subqueries represented by the remaining SLG-trees in the SLG-forest generated from the main query—are sound.

Lemma 7.8 *Let G be an atom and $C_1, C_2 \in \mathcal{C}$ constraints. Then*

$$C_1 \leq C_2 \supset \| \textsc{ans}(G, C_1) \| \subseteq \| \textsc{ans}(G, C_2) \|$$

P r o o f: Immediate from the definition of $\textsc{ans}$.

```
tc(X,Y):- tc(X,Z), tc(Z,Y).                              e(a,b). e(b,c).
tc(X,Y):- e(X,Y).                                        e(c,b). e(c,d).
```

```
 (1)  root(tc(X,Y);[X = b])
 (2)   ├body(tc(X,Y);tc(X,Z),tc(Z,Y);[X = b])
 (4)   │  └goal(tc(X,Y);tc(X,Z),[X = b];tc(Z,Y);[X = b])          get results for X = b
 (8)   │     ├body(tc(X,Y);tc(Z,Y);[X = b, Z = c])                answer from node (7)
 (9)   │     │  └goal(tc(X,Y);tc(Z,Y),[Z = c];;[X = b, Z = c])    get results for X = c (new tree)
(17)   │     │     ├body(tc(X,Y);;[X = b, Z = c, Y = b]           answer from node (16)
(19)   │     │     │  └ans(tc(X,Y);[X = b, Y = b])                new answer for X = b
(30)   │     │     ├body(tc(X,Y);;[X = b, Z = c, Y = c]           answer from node (29)
(31)   │     │     │  └ans(tc(X,Y);[X = b, Y = c])
(42)   │     │     └body(tc(X,Y);;[X = b, Z = c, Y = d]           answer from node (39)
(43)   │     │        └ans(tc(X,Y);[X = b, Y = d])                new answer for X = b
(20)   │     ├body(tc(X,Y);tc(Z,Y);[X = b, Z = b])                answer from node (18)
(21)   │     │  └goal(tc(X,Y);tc(Z,Y),[Z = b];;[X = b, Z = b])    get results for X = b
(22)   │     │     ├body(tc(X,Y);;[X = b, Z = b, Y = c]           answer from node (7)
(24)   │     │     │  └ans(tc(X,Y);[X = b, Y = c])
(23)   │     │     ├body(tc(X,Y);;[X = b, Z = b, Y = b]           answer from node (18)
(25)   │     │     │  └ans(tc(X,Y);[X = b, Y = b])
(45)   │     │     └body(tc(X,Y);;[X = b, Z = b, Y = d]
(51)   │     │        └ans(tc(X,Y);[X = b, Y = d])
(44)   │     └body(tc(X,Y);tc(Z,Y);[X = b, Z = d])                answer from node (41)
(46)   │        └goal(tc(X,Y);tc(Z,Y),[Z = c];;[X = b, Z = d])    get results for X = d (new tree)
 (3)   └body(tc(X,Y);e(X,Y);[X = b])
 (5)      └goal(tc(X,Y);e(X,Y),[X = b];;[X = b])
 (6)         └body(tc(X,Y);;[X = b, Y = c])
 (7)            └ans(tc(X,Y);[X = b, Y = c])                      new answer for X = b

(10)  root(tc(X,Y);[X = c])
(11)   ├body(tc(X,Y);tc(X,Z),tc(Z,Y);[X = c])
(13)   │  └goal(tc(X,Y);tc(X,Z),[X = c];tc(Z,Y);[X = c])          get results for X = c
(18)   │     ├body(tc(X,Y);tc(Z,Y);[X = c, Z = b]                 answer from node (16)
(26)   │     │  └goal(tc(X,Y);tc(Z,Y),[Z = b];;[X = c, Z = b]     get results for X = b
(27)   │     │     ├body(tc(X,Y);[X = c, Z = b, Y = c]            answer from node (7)
(29)   │     │     │  └ans(tc(X,Y);[X = c, Y = c]                 new answer for X = c
(28)   │     │     ├body(tc(X,Y);[X = c, Z = b, Y = b]            answer from node (18)
(33)   │     │     │  └ans(tc(X,Y);[X = c, Y = b]
(46)   │     │     └body(tc(X,Y);[X = c, Z = b, Y = d]
(53)   │     │        └ans(tc(X,Y);[X = c, Y = d]
(32)   │     ├body(tc(X,Y);tc(Z,Y);[X = c, Z = c]                 answer from node (29)
(34)   │     │  └goal(tc(X,Y);tc(Z,Y),[Z = c];;[X = c, Z = c]     get results for X = c
(35)   │     │     ├body(tc(X,Y);[X = c, Z = c, Y = b]            answer from node (16)
(37)   │     │     │  └ans(tc(X,Y);[X = c, Y = b]
(35)   │     │     ├body(tc(X,Y);[X = c, Z = c, Y = c]            answer from node (29)
(38)   │     │     │  └ans(tc(X,Y);[X = c, Y = c]
(42)   │     │     └body(tc(X,Y);[X = c, Z = c, Y = c]
(54)   │     │        └ans(tc(X,Y);[X = c, Y = c]
(44)   │     └body(tc(X,Y);tc(Z,Y);[X = c, Z = d]
(54)   │        └goal(tc(X,Y);tc(Z,Y),[Z = d];;[X = c, Z = b]
(12)   └body(tc(X,Y);e(X,Y);[X = c])
(14)      └goal(tc(X,Y);e(X,Y),[X = c];;[X = c])
(15)         ├body(tc(X,Y);;[X = c, Y = b])
(16)         │  └ans(tc(X,Y);[X = c, Y = b])                      new answer for X = c
(39)         └body(tc(X,Y);;[X = c, Y = d])
(40)            └ans(tc(X,Y);[X = c, Y = d])                      new answer for X = c

(45)  root(tc(X,Y);[X = d])
(46)   ├body(tc(X,Y);tc(X,Z),tc(Z,Y);[X = d])
(48)   │  └goal(tc(X,Y);tc(X,Z),[X = d];tc(Z,Y);[X = d])
(47)   └body(tc(X,Y);e(X,Y);[X = d])
(49)      └goal(tc(X,Y);e(X,Y),[X = d];;[X = d])
```

The First column indicates the order, in which the individual nodes have been created, the second column shows the SLG-trees generated, and the last column describes how answers have been generated and propagated among the SLG-trees.

Figure 7.2: Annotated SLG evaluation for query $tc(X,Y)$ where $X = b$.

The next thing to show is that the algorithm computes all the answers to the given query. This is a little bit more complicated because the algorithm does not compute all the answers to an *uninstantiated* query as the bottom-up evaluation does. However we have the following lemma:

Lemma 7.9 *Let* (G_0, C_0, P) *be a query. Then for every* $\textsc{slg}(G,C) \in \mathrm{SLG}(G_0, C_0)$ *and every valuation* ϑ

$$G\vartheta \in \mathrm{TP}^\omega(\emptyset) \wedge \vartheta \models C \supset \vartheta \in \| \textsc{ans}(G,C) \|$$

P r o o f: By induction on stages of TP. Let $G\vartheta \in \mathrm{TP}^i(\emptyset)$ and $\vartheta \models C$. The claim holds vacuously for $i = 0$. Let $i > 0$. Then there is a clause $G \leftarrow D, B_1, \ldots, B_k \in P$ and an extension θ of ϑ such that $\theta \models D$ and $B_i\theta \in \mathrm{TP}^{i-1}(\emptyset)$. We can construct a path in $\textsc{slg}(G,C)$ that ends with a node $\mathsf{ans}(G; A')$ such that $\vartheta \models A'$. Using the assumption $\vartheta \models C$ we have $\theta \models C \wedge D$ (this corresponds to the application of the *Clause Resolution* rule). Thus, $\theta \models C_1'$ for at least one element C_1' of $\exists_{B_1} C \wedge D$. By the inductive hypothesis $\theta \in \| \textsc{ans}(B_1, C_1') \|$. This fact in turn yields a node $\mathsf{ans}(B_1; A_1) \in \textsc{slg}(B_1, C_1')$ such that $\theta \models A_1$. Again, using the previous assumptions, $\theta \models C \wedge D \wedge A_1$. In general, let C_i' be an element of $\exists_{B_i}.C \wedge D \wedge A_1 \wedge \ldots \wedge A_{i-1}$. Clearly, by $i-1$ applications of the induction hypothesis, $\theta \models C_i'$. Then by induction hypothesis on B_i we have $\theta \in \| \textsc{ans}(B_i, C_i') \|$. This exactly corresponds to an application of the *Query Projection* and *Answer Propagation* rules from Definition 7.1. After k steps we have $\theta \models C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$. Thus, $\vartheta \models A'$ for some element A' of $\exists_G.C \wedge D \wedge A_1 \wedge \ldots \wedge A_k$ (this is achieved using the *Answer Projection* rule). Thus $G\vartheta \in \textsc{ans}(G,C)$.

In the actual algorithm, the application of the *Answer Propagation* rule does not necessarily use the tree $\textsc{slg}(B_i, C_i')$ for answer resolution. However, if it uses different tree $\textsc{slg}(B_i, C_i'')$ then it is always the case that $C_i' \leq_{\mathcal{C}} C_i''$. Thus, by Lemma 7.8 $\| \textsc{ans}(B_i, C_i') \| \subseteq \| \textsc{ans}(B_i, C_i'') \|$ and no answers can possibly be lost.

By composing the four lemmas above we have:

Theorem 7.10 (Soundness and Completeness) *Let* (G, C, P) *be a query tuple. Then for all valuations* ϑ

$$G\vartheta \in \mathrm{TP}^\omega(\emptyset) \wedge \vartheta \models C \iff \vartheta \in \| \textsc{ans}(G,C) \|$$

P r o o f: Soundness follows from Lemmas 7.7 and 7.6. Completeness is proven in Lemma 7.9.

The soundness and completeness proof is based on the reduction to the fixpoint computation on *ground* instances. However, to prove the termination of query evaluation (in both the bottom-up and top-down cases) a finite encoding of a potentially infinite result of the evaluation is needed (cf. Chapter 4 and [73, 96]).

Theorem 7.11 (Termination) *If the class* $\mathcal{C}$ *is constraint-compact then the* $SLG^{\mathcal{C}}$ *evaluation terminates for all queries* (G, C, P).

P r o o f: Let $\mathcal{C}$ be a constraint-compact class of constraints. Then:

1. The number of trees in the SLG-forest $\mathrm{SLG}(G, C)$ is finite, as there are only finitely many predicate symbols $G' \in P$ and for every predicate symbol the set

$$\{C_i : \mathsf{root}(G', C_i) \in \mathrm{SLG}(G, C)\}$$

is finite by Definitions 4.4 and 7.1.

2. Every **root** node has finitely many children, as there are only finitely many clauses in the Datalog$^{\mathcal{C}}$ program P.

3. Every **body** node has finitely many children, as the set $\exists_G.C$ is finite for any $C \in \mathcal{C}$.

4. Every **goal** node as only finitely many children, as there are only finitely many elements in the set $\mathrm{ANS}(G', C')$ for any atom G' and $C' \in \mathcal{C}$ by Definition 4.4.

5. Every $\mathrm{SLG}(G, C)$ has finite depth, because of finite number of subgoals in the bodies of each clause in P.

Together, the rules in Definition 7.1 can be applied only finitely many times.

Thus the termination for Constraint Memoing is guaranteed in all cases when the bottom-up algorithm terminates computing a finite interpretation $\mathrm{TP}_{\mathcal{C}}^{\omega}(\emptyset)$. Moreover, it is usually easy to *decompose* the original bottom-up evaluation procedure and extract the elementary operations on constraints needed for Constraint Memoing (Definition 4.3).

7.3.1 Optimizations

To reduce the overhead introduced by the SLG$^{\mathcal{C}}$ resolution (in comparison to standard SLD resolution) we explore several possibilities:

Solving more general goals than necessary:

Action/Node	Children	Conditions
$\boxed{\text{Query projection}}$		
$\mathsf{body}(G; B_1, \ldots, B_k; C)$	$\left\{\begin{array}{c} \mathsf{goal}(G; B_1, C_1; B_2, \ldots, B_k; C) \\ \vdots \\ \mathsf{goal}(G; B_1, C_l; B_2, \ldots, B_k; C) \end{array}\right.$	for $\exists_{B_1}.C \supset C_1 \vee \ldots \vee C_l$ for some $C_1, \ldots, C_l$

This modification may reduce the number of SLG-trees in the SLG-forest (if $|\exists_{B_1}.C| < l$). However, the propagation of constraints *at the time of goal resolution* is reduced. The soundness and completeness properties are preserved by Lemma 7.8. The termination is guaranteed similarly to Theorem 7.11.

In [45] the following version of this modification was presented:

Action/Node	Children	Conditions
$\boxed{\text{Query projection}}$		
$\mathsf{body}(G; B_1, \ldots, B_k; C) \rightarrow$	$\mathsf{goal}(G; B_1, \mathrm{true}; B_2, \ldots, B_k; C)$	none

In this case, there is only one SLG-tree per predicate symbol. On the other hand, no constraints are propagated at the time of goal resolution—the constraints are used merely to restrict the returned answers. Thus the computation essentially computes all answers to an *uninstantiated* query, similarly to the bottom-up algorithm, and the performance suffers[5].

Mixed SLG and SLD resolution (by memoing only a subset of the predicate symbols present in the program).

Action/Node	Children	Conditions
$\boxed{\text{Non-tabled resolution}}$		
$\mathsf{body}(G; B_1, \ldots, B_k; C)$	$\begin{cases} \mathsf{body}(G; B_1^1, \ldots, B_k; C \wedge D^1) \\ \vdots \\ \mathsf{body}(G; B_1^l, \ldots, B_k; C \wedge D^l) \end{cases}$	for B_1 not tabled goal $B_1 \leftarrow D^i, B_1^i, \ldots, B_{k_i}^i \in P$ and $C \wedge D^i$ satisfiable

This is a different way of reducing the number of SLG-trees generated by the algorithm: SLG-trees are generated only for a subset of the predicate symbols in P. The remaining symbols are always resolved using the program clauses, similarly to SLD-resolution. Again, soundness and completeness are preserved (by simple modification of Theorem 7.10). Termination is guaranteed if and only if at least one literal is resolved by the $\mathrm{SLG}^{\mathcal{C}}$ resolution for every cycle in the dependency graph of P (this follows by an easy extension of Theorem 7.11). Otherwise, an infinite branch may appear in some of the SLG-trees. This may lead to non-termination similarly as in the case of SLD-resolution.

Also, as there is only a bounded number of SLD resolution steps between any two $\mathrm{SLG}^{\mathcal{C}}$ resolution steps, the bodies of the clauses can be *unfolded* in the bodies of their callers. This transformation completely eliminates the need for SLD resolution steps.

Program transformation similar to factoring The previous folding transformation may introduce unnecessary recomputation of conjunctions of goals. This can be avoided using the *factoring* technique that *folds* common parts of bodies of the clauses and creates separate clauses. Note that the recomputation is avoided by making the heads of such clauses tabled.

The last two optimizations are based on program transformation. However, in contrast to the Magic Set transformation, these two transformations are completely query-independent.

7.4 Magic Set Transformation for $\mathcal{C}$ ($\mathrm{MST}^{\mathcal{C}}$)

This section describes a simple version of the program transformation approach to the goal-oriented query evaluation in constraint deductive databases—the Magic Set Transformation. The transformation has to be slightly modified in the context of constraint databases:

[5]The performance is approximately the same as evaluating the uninstantiated query first.

Definition 7.12 *Let x be a variable and $C \in \mathcal{C}$. We say that x is* free with respect to C *if for a valuation θ such that $\theta \models C$ the valuation $\theta[x/a] \models C$ for all $a \in Dom(x)$. Otherwise we say that x is* bound in C. *We denote* bound(C) *the set of variables bound in C.*

Note that this definition does not consider the mutual dependencies between different variables. In the case of ground Datalog this was not an issue: a variable could be either free or bound to a single constant. We take an *optimistic* approach to MST: we use only two adornments $\{b, f\}$ with the following meaning: if an argument is adorned by f then it is going to be free at the time of the goal processing, if it is adorned by b then it *may* be bound by some constraint. Note that the b adornment does not guarantee that the variable is bound:

Example 7.13 Consider a goal $p(x)$ and a constraint $x < y$. Then x in p is adorned by b but in fact it is not constrained at all.

However, to avoid this situation, a much more complicated analysis of the constraint queries is needed, including the analysis of the data stored in the constraint database, which may not be feasible in general. In the case of Datalog (i.e., the constraint class generated from the set $\{x = a : a \in A\}$) the MST$^{\mathcal{C}}$ transformation reduces to the standard MST for Datalog [8].

Definition 7.14 (Adornment) *Let G be an atom of arity k. An* adornment *A for this atom is a string over $\{b, f\}$ of length k. We say that the i-th argument of G is adorned by b (f) if the i-th position in A is b (f), respectively.*

We say that A is an adornment of G with respect to $C \in \mathcal{C}$ *if the i-th position of A is b if the i-th argument of A is bound in C or the i-th position of A is f if the i-th argument of A is free in C.*

Let $G \leftarrow D, B_1, \ldots, B_n \in P$ be a clause and A an adornment of G. Then an adornment of the clause with respect to G and A *is the set of adornments $\{A_{B_1}, \ldots, A_{B_n}\}$ such that A_{B_i} is an adornment for B_i. The adornment for the atoms in the body of the clause is constructed as follows: the j-th argument of B_i is adorned by b in A_{B_i} if:*

1. *the j-th argument of B_i occurs in the head of the clause G as an argument adorned by b in A, or*
2. *j-th argument of B_i is a variable bound in D, or*
3. *j-th argument of B_i occurs as a variable in a subgoal preceding B_i.*

Otherwise the j-th argument of B_i is adorned by f.

A magic atom *for an atom G and an adornment A is the atom* magicG^A *that has only those arguments of G adorned by b in A, i.e., the arity of* magicG^A *is less or equal to the arity of G depending on how many arguments of G are adorned by b. Arguments adorned by f are removed.*

For simplicity, only left-to-right SIPS[6] is used (rule 3 of the previous definition). This corresponds to the selection rule used in Constraint Memoing. In both cases, different

[6]Sideways Information Passing Style.

selection rules may improve the efficiency of query evaluation [70]. However, in the case of MST, the SIPS is fixed during the *program transformation* phase and there are technical difficulties with combining different SIPS in one program. In the case of SLG$^{\mathcal{C}}$ evaluation, the selection rule can be adjusted during the evaluation process dynamically while preserving correctness of the answers.

Algorithm 7.15 (MST program transformation)

$$\begin{aligned}
\mathrm{mst}(G,A,P) = \{\, & G^A \leftarrow D, \mathrm{magic}G^A, B_1^{A_{B_1}}, \ldots, B_k^{A_{B_k}} \\
& \mathrm{magic}B_1^{A_{B_1}} \leftarrow D, \mathrm{magic}G^A \\
& \quad \vdots \\
& \mathrm{magic}B_k^{A_{B_k}} \leftarrow D, B_1^{A_{B_1}}, \ldots, B_{k-1}^{A_{B_{k-1}}}, \mathrm{magic}G^A \\
& \qquad \text{for } G \leftarrow D, B_1, \ldots, B_k \in P \,\} \\
\cup & \bigcup_{0<i\leq k} \mathrm{mst}(B_i, A_{B_i}, P)
\end{aligned}$$

where $\{A_{B_1}, \ldots, A_{B_k}\}$ *is the adornment of* $G \leftarrow D, B_1, \ldots, B_k$ *with respect to* G *and* A *and* $\mathrm{magic}G^A, \mathrm{magic}B_1^{A_{B_1}}, \ldots, \mathrm{magic}B_k^{A_{B_k}}$ *are the magic atoms for* $G, B_1, \ldots, B_k$*, respectively.*

Definition 7.16 (MST for $\mathcal{C}$) *Let* (G, C, P) *be a query. A* MST$^{\mathcal{C}}$ *of this query is the query* (G^A, C, P') *where*

$$P' = \mathrm{mst}(G, A, P) \cup \bigcup_{C_i \in \exists_G . C} \{\mathrm{magic}G^A \leftarrow C_i\}$$

for A *being an adornment for* G *with respect to* C.

The only difference between the classical version of the MST and our version is in the definitions of the *free* variable and the *seed* of the magic computation, the clauses $\mathrm{magic}G^A \leftarrow C_i$, derived by the quantifier elimination procedure from the original query.

Theorem 7.17 (Soundness and completeness of the MST$^{\mathcal{C}}$ transformation)
For every Datalog$^{\mathcal{C}}$ *query* (G, C, P_0) *the query* (G^A, C, P_1) *obtained by applying the* MST$^{\mathcal{C}}$ *on* (G, C, P_0) *satisfies the following condition:*

$$G\theta \in \mathrm{TP}_{P_0}^{\omega}(\emptyset) \iff G\theta \in \mathrm{TP}_{P_1}^{\omega}(\emptyset)$$

for any ground θ*, such that* $\theta \models C$.

P r o o f: only if part: Let $G\theta \in \mathrm{TP}_{P_0}^{\omega}(\emptyset)$. Then there exists a proof tree T for $G\theta$ with inference rules from P_0, where the inference rules are exactly the clauses of the program and axioms are the atomic clauses. We can easily replace each inference a rule in T by rule from P_1, as for each adornment we created a copy of the original rule. If we assume all magic predicates to be true (i.e., facts) in the tree we have a valid proof tree for $G\theta$ where all rules are from P_1.

What remains to be shown is, that in each application of an inference rule from T the magic predicate is true—derivable from the transformed program. Note that the magic predicates just restrict, for which θ the rule is true—so it is sufficient to show that the restriction is not too severe: this can be done by induction on distance from the root of T.

Base case: In the root inference we simply use the seed magic$G^A \leftarrow C_i$ defined as a restriction of the query's arguments to the instantiated ones. This obviously allows to derive all the tuples implied by the original constraint in the partially instantiated query.

Induction: Assume all magicB_i's to be true for all the applications of the inference rule with distance at most k from the root of T. Let the k-th inference be of the form

$$Q^A \leftarrow D, \text{magic}Q^A, B_1^{A_1}, \ldots, B_k^{A_k}$$

The MST$^{\mathcal{C}}$ has created magic a rule for magic$B_i^{A_i}$ literal of the form

$$\text{magic}B_i^A \leftarrow D, B_1^{A_1}, \ldots, B_{i-1}^{A_{i-1}}, \text{magic}Q^A$$

Because all the literals in the body of this rule are true by the induction hypothesis or because they are contained in T, the head of this clause is also true by soundness of the inference (remember that the arguments of magic$B_i^{A_i}$'s are just restrictions of arguments of B_i's). Thus magic$B_i^{A_i}$'s are true for all nodes with distance $k+1$ from the root.

Together we have a proof tree of $G\theta$ with inference rules from P_1, where the magic predicates are considered true atoms; this is not a proof tree in P_1, but by completeness there must exist one.

if part: Let $G\theta \in \mathrm{TP}_{P_1}^{\omega}(\emptyset)$. Then there exists a proof tree T for $G\theta$ with inference rules from P_1, where the inference rules are exactly the clauses of the program and axioms are the EDB relations. Each of the inference rules in P_1 follows the pattern:

$$Q^A :- D, \text{magic}Q^A, B_1^{A_1}, \ldots, B_k^{A_k}$$

where magicQ^A is the magic predicate for A and $B_i^{A_i}$ are adorned literals.

We are not interested in expanding the magicB^A predicates (we know that they are true because all antecedents of a rule applied in a proof tree must be true).

It is easy to see that when we project out all the adornments and magic predicates, we obtain a proof tree of G that uses only rules from P. Thus $G\theta \in \mathrm{TP}_{P_0}^{\omega}(\emptyset)$.

However, it is easy to find cases where MST$^{\mathcal{C}}$ as a static method cannot predict the binding patterns precisely enough and thus its performance is inferior to the SLG$^{\mathcal{C}}$ strategy:

Example 7.18 Consider the following fragment of a Datalog$^{\mathcal{C}}$ program:

$$q_0 \rightarrow \ldots, q(X,Y), r(X,Y), \ldots$$

$$r(X,Y) \rightarrow p(X,Z).$$
$$r(X,Y) \rightarrow p(Z,Y).$$

The MST$^{\mathcal{C}}$ adorns the goal $r(X,Y)$ in the body of the first clause with the adornment bb (cf. rule 3 of Definition 7.14). Thus the clauses used for resolving `r(X,Y)` are

$$r^{bb}(X,Y) :- \text{magic}r^{bb}, p^{bf}(X,Z).$$
$$r^{bb}(X,Y) :- \text{magic}r^{bb}, p^{fb}(Z,Y).$$

Test cases for (ground) Datalog:

Transitive closure (tc)	chain(N)	cycle(N)	tree(N)
`tc(X,Z):- tc(X,Y),tc(Y,Z).`	`edge(1,2).`	`edge(1,2).`	`edge(1,2).`
`tc(X,Y):- edge(X,Y).`	`...`	`...`	`...`
	`edge(N-1,N).`	`edge(N-1,N).`	`edge(N-1,2N-2).`
		`edge(N,1).`	`edge(N-1,2N-1).`

Queries: `tc(X,Y)`, `tc(1,Y)`, `tc(1,32)` for tc+chain(32), tc+cycle(32), and tc+tree(32).

Test cases for Datalog with gap-order constraints over Z:

Monotonic Path (path)	
`path(A,C,X,Z):- X<Y<Z, path(A,B,X,Y), path(B,C,Y,Z).`	
`path(A,B,X,Y):- edge(A,B,X,Y).`	
chain with unit edges (echain(N))	chain with equidistant nodes (nchain(N))
`edge(1,2,X,Y):- X<Y.`	`edge(1,2,X,Y):- 0<X<2, 1<Y<3.`
`...`	`...`
`edge(N-1,N,X,Y):- X<Y.`	`edge(N-1,N,X,Y):- N-2<X<N, N-1<Y<N+1.`

Queries: `path(A,B,X,Y)` and `path(A,B,X,Y), 5<X,Y<11` for path+echain(16)
`path(A,B,X,Y)` and `path(A,B,X,Y), -1<X,X+14<Y,Y<16` for path+nchain(16)

Figure 7.3: Test Programs

However, in the case, $q(X,Y) = X < Y$, the bottom-up evaluation of the transformed program has to evaluate both $p^{bf}(X,Z)$ and $p^{fb}(Z,Y)$ instead of evaluating $p^{ff}(X,Z)$, which would be sufficient, as both the projections of $X < Y$ yield the trivial constraint *true*. This is correctly realized by the SLG$^{\mathcal{C}}$ algorithm, and only one SLG-tree is created.

7.5 Performance

In this section, an analytical complexity bound for the Constraint Memoing evaluation is provided. The complexity of the top-down evaluation depends on the particular class of constraints $\mathcal{C}$; our analysis is based on the relative comparison to the (time- and space-) complexity of the bottom-up evaluation procedure. The analytical results are confirmed by experimental results that show the performance gain achieved by Constraint Memoing.

7.5.1 Theory

We show that the complexity of the SLG$^{\mathcal{C}}$ algorithm is no worse than complexity of the bottom-up query evaluation:

Theorem 7.19 *Let* $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$ *be the result of the bottom-up evaluation of the query* (G, C, P) *and* $f(n)$ *a function such that* $|\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)| \in O(f(|P|))$. *Then*

1. *the SLG-forest* $\mathrm{SLG}(G, C)$ *has at most* $O(f^2)$ *nodes.*

$I, J, \Delta := \emptyset, \emptyset, \emptyset$
repeat
  $J := \mathrm{TP}'_{\mathcal{C}}(I, \Delta)$
  $I, \Delta := I \cup J, J - I$
while $\Delta \neq \emptyset$

where $\mathrm{TP}'_{\mathcal{C}}$ is the same as $\mathrm{TP}_{\mathcal{C}}$, but in addition requires that for each clause

$$A \leftarrow D, B_1, \ldots, B_k$$

at least one of the named constraints (B_i, C_i) is in the set Δ. This arrangement avoids unnecessary recomputation of already known results while preserving correctness.

Figure 7.4: Semi-naive Bottom-up evaluation Algorithm.

2. there are at most $O(f^2)$ applications of the $SLG^{\mathcal{C}}$ rules in the evaluation of the query.

P r o o f: (1) follows from the observation that every SLG-tree in the SLG forest has at most $O(f)$ leaves and thus also at most $O(f)$ nodes as the height of the trees is fixed by the number of subgoals in the clauses of P, and there are at most $O(f)$ different SLG trees in the SLG-forest as the size of $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$ limits the number of possible roots of the SLG-trees. (2) follows immediately from (1) as every application of a rule creates at least one new node in the SLG-forest.

A careful implementation needs to store only a single path (of fixed length) in every SLG-tree in the SLG forest. Thus, the space requirements can be reduced to $O(f)$. On the other hand, the quadratic number of rule applications cannot be avoided in general. However, by analysis of the bottom-up algorithm the number of applications of *clauses* in P is also quadratic with respect to f (the bottom-up algorithm can recompute the same element of the interpretation several times, even if it is added only once to the interpretation $\mathrm{TP}^{\omega}_{\mathcal{C}}(\emptyset)$).

7.5.2 Implementation

We implemented an experimental version of the following deductive query evaluation algorithms:

1. Naive Bottom-up: a straightforward implementation of algorithm 4.10,
2. Semi-naive Bottom-up: a modification of Algorithm 4.10, see Figure 7.4,
3. Semi-naive Bottom-up with $\mathrm{MST}^{\mathcal{C}}$, and
4. $\mathrm{SLG}^{\mathcal{C}}$.

The implementation of each of the evaluation algorithms is independent of the underlying class of constraints. For each constraint class we need to provide the elementary operations on the representations of constraints (cf. Definition 4.3), and a few additional auxiliary operations:

`constraint_new(N,C)`: `C` is the representation of the constraint *true* over `N` variables. This operation is used to create a *fresh* representation of constraints, present during the evaluation of the individual clauses of the Datalog$^{\mathcal{C}}$ program.

`constraint_qe(G,C,Cn)`: Let `G` be an atom. Then `Cn` is the finite set (list) of constraints that is equivalent to the constraint `C` after all variables not in `G` are eliminated. These operations are used in both the *query projection*, where we project the constraint on the free variables of the goal to be resolved, and the *answer projection*, where we eliminate all variables not in the head of the clause.

`constraint_subsumes(C1,C2)` is the algorithm for subsumption checking. The goal succeeds if `C1` subsumes `C2`. We can assume that the constraints `C1` and `C2` are over the same set of variables, as we use this operation for deciding, if a new named constraint has been derived by the particular method.

`constraint_and(G,C,CO,CN)`: This operation computes the constraint conjunction of the constraint `CO` with the constraint `C` where all the variables in `C` are renamed with respect to the variables of the atom `G`. This operation is used when a constraint derived by a subgoal of a clause is "and"-ed to the overall constraint for the variables in the body of the clause. The operation produces only consistent constraints; if the conjunction is not satisfiable, the operation fails.

`constraint_var(V,C)` This operation is used solely by the MST$^{\mathcal{C}}$ transformation to determine the adornments of variables (cf. Section 7.4). The last two operations are just for the convenience of the user of the system:

`constraint_pp(C)` allows "pretty printing" of the results of the evaluation, and

`constraint_read(T,G,C)` allows entering the constraints as formulas, rather than as the actual representation as Prolog terms.

In addition, we need to specify the Datalog$^{\mathcal{C}}$ programs that we intend to evaluate. Figure 7.5 depicts the structure of the experimental system.

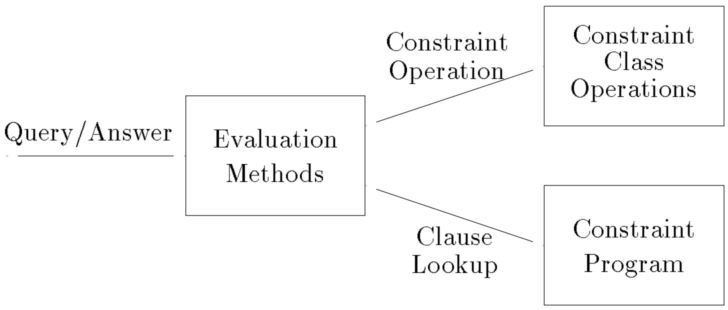


Figure 7.5: Structure of the experimental system.

7.5.3 Experimental Results

Table 7.6 summarizes the running time (in milliseconds) for the test programs in Figure 7.3. The examples of partially instantiated queries are those where the optimization achieves the

Query	Data	Naive Bottom-up	Semi-naive Bottom-up	Semi-naive with MST	Top-down
`tc(X,Y)`	tc,chain(24)	61934	31766	60134	27634
`tc(1,Y)`	tc,chain(24)	61283	31583	41333	2533
`tc(1,24)`	tc,chain(24)	61166	31567	59517	3766
`tc(X,Y)`	tc,cycle(24)	331134	273768	536466	238567
`tc(1,Y)`	tc,cycle(24)	330034	273233	278850	14100
`tc(1,24)`	tc,cycle(24)	329701	272616	306134	15917
`tc(X,Y)`	tc,tree(24)	12450	7300	15084	5800
`tc(1,Y)`	tc,tree(24)	12384	7267	13083	1450
`tc(1,24)`	tc,tree(24)	12333	7166	19967	2966
`path(A,B,X,Y)`	path,echain(16)	28317	17950	52685	14250
`path, 5<X,Y<11`	path,echain(16)	27633	17450	6832	2450
`path(A,B,X,Y)`	path,nchain(16)	40617	25283	78048	24468
`path, -1<X+14<Y<16`	path,nchain(16)	41050	25565	47347	11116

Figure 7.6: Running times of test queries for various evaluation procedures (in msec).

least effect. This can be seen from the graphs in Figures 7.7 and 7.8. The graphs show the time needed to find a path (always only one exists) between nodes x and y (represented by the axes x and y). The z axis is represents the elapsed time (in milliseconds). The results show that while the implementations of the various evaluation methods are comparable (the results on uninstantiated queries are approximately the same), the evaluation of instantiated queries is much more efficient using Constraint Memoing. This boost is inherent to the top-down evaluation method as it is not caused by more sophisticated implementation.

7.6 Conclusion

We have proposed a practical approach to query evaluation in generalized constraint databases. Both the analytical and the empirical results show that Constraint Memoing is no worse than a comparable bottom-up method and in many cases the practical performance is much better even when using very naive implementation. The performance of Constraint Memoing can be boosted by utilizing the compilation techniques developed in [91] and performance similar to ground Datalog can be expected.

Future research in this area will focus on following issues:

Compilation of constraints. To achieve an efficient implementation of Constraint Memoing, data structures for efficient representation of the constraints have to be developed. There are two main differences that distinguish the top-down processing of constraint deductive queries from a general-purpose logic programming languages to be addressed:

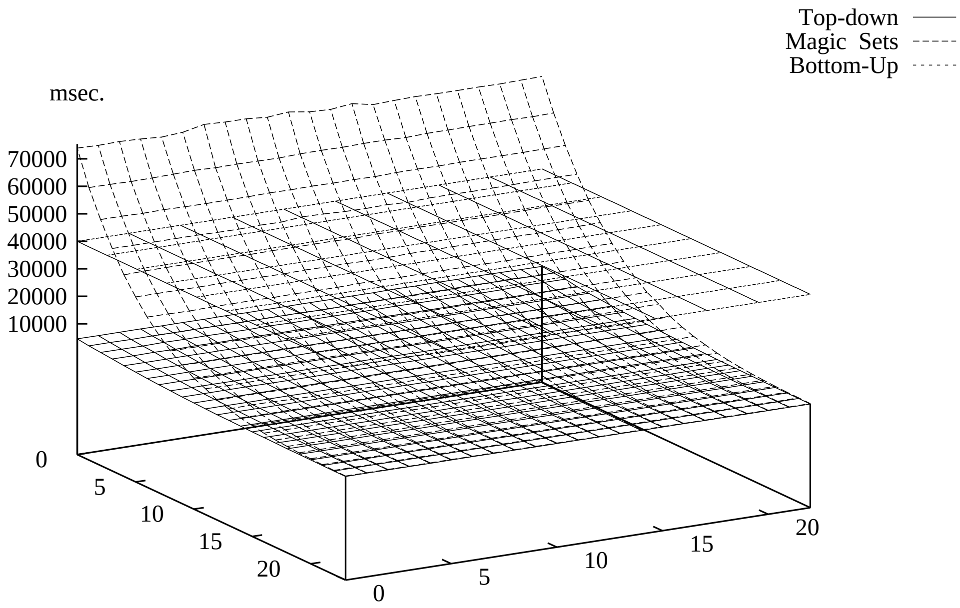


Figure 7.7: Datalog over 24-element chain (`tc, chain(24)`)

- In most cases, the size of the constraint representation is bounded with respect to the arity of a literal. However, general Logic Programming engines allow unbounded terms to be built. Exploring this property may lead to an efficient *stack*-based implementation of the evaluation procedure without the need for a heap.

- On the other hand, classical Logic Programming assumes that every variable is either free or bound to a single term, and this binding can be changed only by backtracking. This assumption is no longer valid in the presence of constraints as more restrictive conditions may be derived after a variable is originally bound. Also, the constraints specify complex relations *between* individual variables, which is not possible in the standard approach.

Development of such a representation enables building of very efficient query evaluation engines based on partial evaluation of the atomic constraints in a given class, similar to the WAM abstract code [103].

Analysis of binding patterns. Similar to the MST transformation, the queries can be analyzed to determine the *flow of information* in clause bodies [70]. This is a considerably more complicated task in the presence of constraints: it is no longer sufficient to focus on single variables; the relationships between groups of variables have to be considered (as noted in Section 7.4). Also, the assumption that all EDB relations are ground (i.e.,

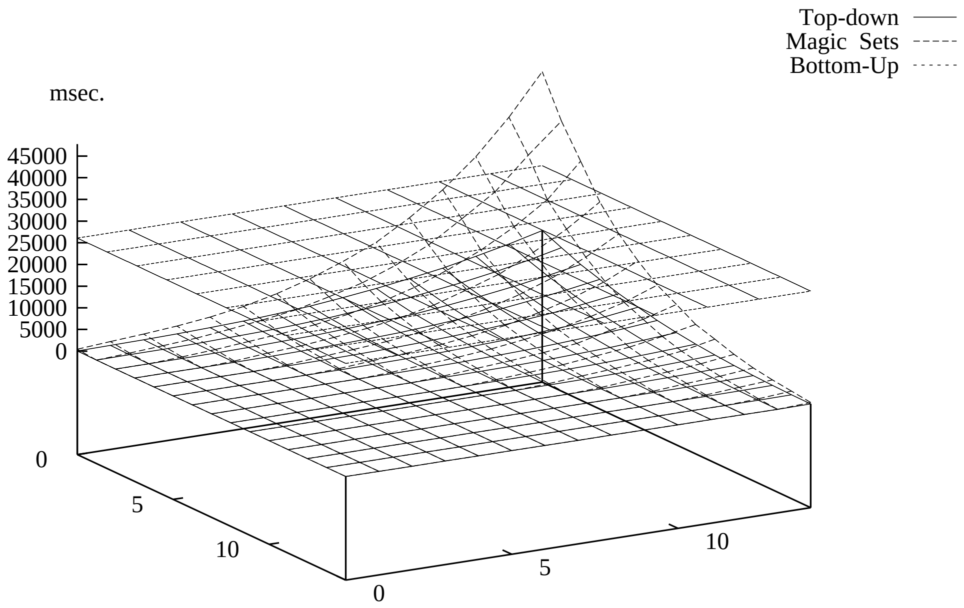


Figure 7.8: Datalog with Gap-order constraints, 16-element chain (`path,nchain(16)`)

after resolution of an EDB goal all variables are bound to constants) is no longer valid—the generalized relations store representation of sets of tuples that may be infinite. Such analysis can be used for several purposes: query optimization, e.g., MST-like rewriting, optimization of access to the constraint database (indexing), goal reordering, etc.

Interface to an existing RDBMS. As the constraints can be finitely encoded, their representation can be stored as tuples in a standard relational database system. However, the query evaluation has to be carried out with respect to the meanings of such encoding (i.e., to perform, e.g., join, of two constraint relations, we cannot use the join operation of the underlying RDBMS directly). We propose that the top-down evaluation procedure be used as a front-end built on top of a standard relational DBMS. The techniques proposed above can be directly applied here.

Negation. Adding negation to Datalog$^{\mathcal{C}}$ in such a way that termination is preserved, is a nontrivial task: even stratified negation often leads immediately to Turing completeness. Essentially, adding negation and preserving the termination of queries requires the constraint class to be closed under negation (complementation) while preserving the constraint-compactness. This condition is easy to satisfy in the case of finite constraint class. However, for constraints over infinite domains, e.g., integers, this is often not possible.

Storage and access methods. To achieve an efficient implementation of constraint databases, new storage management techniques have to be developed: Access methods suitable for fast retrieval of the stored information, efficient updates of generalized relations, indexing techniques [59], etc.

Benchmarks. The performance of various implementations of Logic Programming languages (e.g., Prolog) is often judged by the performance on a standard benchmarks (e.g., `nrev`). We propose to develop similar suite of benchmarks for measuring the performance of query evaluation methods for constraint databases. The benchmarks should be independent of the particular class of constraints. Such a test suite would allow one to compare the performance of various query evaluation methods.

7.7 Related Work

Recently there have been several other attempts to make query evaluation efficient in the presence of constraints. There are two main directions of this research:

1. The first direction has its roots in the (deductive) database community: In [61, 68, 86, 89] techniques for pushing constraints present in the query are proposed. However, the goal of these methods is to *preprocess* the query, i.e., the goal and the rules, with respect to the given constraints, for a subsequent bottom-up evaluation. We present a completely different evaluation strategy where the constraints are propagated dynamically without the need for the preprocessing of the query. Also, in the standard database approach, the constraints are considered only as *conditions* that restrict the otherwise ground answers. Constraint Memoing uses constraints as a tool for *representing* both the data computed by the queries and stored in the database itself. This dramatically increases the expressive power of the query language while preserving the termination and efficiency.

2. The other direction is pursued in the area of Logic Programming: In [45, 54, 63] top-down evaluation for constraint logic programs is proposed. However, in all cases, general constraint-solving procedures are used. Thus these methods are not directly useful for query evaluation in constraint databases as the termination cannot be guaranteed. The closest to our work is [45]. However, the method proposed there allows only propagation of constants (i.e., constraints of the form $x = a$); the *constraint* part of the query is essentially computed bottom-up. Our approach allows full propagation of all possible constraints during the whole evaluation process.

In [100] the bottom-up approach equipped with a query transformation phase is shown to be no worse than the top-down approach for restricted classes of Datalog programs over ground relations. We show that top-down approach is no worse than the bottom-up approach in the worst case, and in many empirical examples the top-down evaluation is much faster than the bottom-up evaluation of the same query.

Bibliographical Notes

An introduction to the topic of query processing can be found from many sources. An in depth analysis of the bottom-up evaluation can be found in the second volume of Ullman's book [100]. An overview of different deductive query evaluation strategies with brief description of each of the methods can be also found in [9]. Here both the top-down and bottom-up strategies are discussed and their relative performance is analyzed. An introduction to the top-down query evaluation (and the area of Logic Programming in general) can be found in [65]. Finally, there is a vast amount of articles on the various implementation techniques for Logic Programming languages based on resolution. Many of the ideas that are used in modern compilers of such languages can be found in [7].

Chapter 8

Conclusion

In this dissertation, we have investigated several questions concerning the design and implementation of Temporal Databases (and other databases intended to handle inherently infinite information). In particular we have shown that:

- Restrictions on the database schema, even when they look very natural, may lead to unrepairable problems, when query languages need to be designed.
- Such restrictions are not crucial as finite representations of subclasses of infinite databases can be developed in the framework of constraint databases.
- We have an efficient implementation of query evaluation techniques over such finite representations.

We also discussed directions of further development at the ends of each chapter of the dissertation. However, there are many other areas that need to be investigated (and solutions have to be found) before commercially viable applications based on this technology can be developed:

- We need more expressive constraint classes. Here we run into problems with termination of all Datalog$^{\mathcal{C}}$ queries (cf. Figure 6.1). However, in [77] a syntactic subclass of Datalog$^{\mathcal{C}}$ queries for Datalog$^{\neg,<Z}$ was introduced, for which termination is guaranteed. Is it possible to restrict Datalog$^{\mathcal{C}}$ queries for other classes of constraints in order to achieve termination (decidability)? Is there a general approach to restricting the class of allowed Datalog$^{\neg}$ programs that guarantees termination with respect to a given constraint class?
- Another important feature of a practical database system is the addition of *aggregation* operators to the language. However, it is fairly difficult to add these operators with a clean declarative semantics to first-order query languages. There are two main approaches to solving this problem:
  1. Logic-based approach: usually based on addition of limited iteration operators over the contents of the relations, or

2. Complex-object approach: allows relations to be nested in other relations, i.e., the 1NF is not enforced any more. However, such extension makes the query evaluation of even first-order languages over the appropriate structures extremely hard, if possible at all.

None of the current approaches solves the problem of aggregation satisfactorily. Thus, further development in this area is needed.

Bibliography

[1] Abiteboul, S., Herr, L., Van den Bussche, J. Temporal Connectives versus Explicit Timestamps in Temporal Query Languages (unpublished report). A preliminary version in Proc. *International Workshop of Temporal Databases*, Zürich, Switzerland, 1995.

[2] Abiteboul, S., Hull, R., Viannu, V. Foundations of Databases. Addison Wesley, 1995.

[3] Abramsky, S., Hankin, C. Abstract Interpretation of Declarative Languages. Ellis Horwood, 1987.

[4] Aho, A. V.,Sethi, R., Ullman, J. D. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, MA, 1986.

[5] Aho, A. V., Ullman, J. D. Universality of data retrieval languages. In *Proc. ACM Symp. on Principles of Programming Languages*, pp. 110–117, 1979.

[6] Apt, K. R., Blair, H., Walker, A. Towards a theory of declarative knowledge. In Minker, J. (ed.) *Foundations of Deductive Databases and Logic Programming*, pp 89–148, Morgan Kaufman, Los Altos, CA, 1988.

[7] Ait-Kaci, H. Warren's Abstract Machine: A Tutorial Reconstruction. MIT-Press 1991.

[8] Bancilhon F., Maier D., Sagiv Y., Ullman J. D. Magic sets and other strange ways to implement logic programs. In *Proc. ACM Symposium on Principles of Database Systems*, 1986.

[9] Bancilhon F., Ramakrishnan, R. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. of ACM SIGMOD '86*, 16–52, 1986.

[10] Birkhoff, G., MacLane, S. Algebra. The MacMillan Co., 1967.

[11] Berman, L. Precise bounds for Presburger arithmetic and the reals with addition. In *Proc. 18th IEEE Symposium on Foundations of Computer Science*, pp. 95–99, 1977.

[12] Baudinet, M., Chomicki, J., Wolper, P. Temporal Deductive Databases. In *Temporal Databases: Theory, Design, And Implementation*, Tansel, A. et al. (editors). Benjamin Cummings 1993.

[13] Baudinet, M., Níezette, M., Wolper P. On the Representation of Infinite Temporal Data and Queries. *10th ACM Symposium on Principles of Database Systems,* pp 280–290, 1991.

[14] Ben-Or, M., Kozen, D., Reif, J. The Complexity of elementary Algebra and Geometry. *JCSS*, 32:251–264, 1986.

[15] Bohlen, M. H., Chomicki, J., Snodgrass, R. T., Toman, D. Querying TSQL2 Databases with Temporal Logic. To appear in Proc. EDBT'96.

[16] Bol, R, Degerstedt, L. The Underlying Search for Magic Templates and Tabulation. Proc. *International Conference on Logic Programming*, 1993.

[17] Chandra, A. K., Harel, D. Computable Queries for Relational Data Bases. *Journal of Computer and System Sciences,* 21:156–178, 1980.

[18] Chandra, A. K., Harel, D. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences,* 25:156–178, 1982.

[19] Chandra, A. K., Harel, D. Horn Clause Queries and Generalizations. *J. Logic Programming*, 2(1):1-15, 1985.

[20] Chomicki J. Temporal Query Languages: a Survey. Proc. *International Conference on Temporal Logic*, July 1994, Bonn, Germany, Springer-Verlag (LNAI 827), pp. 506–534.

[21] Chomicki J. History-less Checking of Dynamic Integrity Constraints. *IEEE International Conference on Data Engineering*, Phoenix, AZ, 1992.

[22] Chomicki J. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. In *ACM Transactions on Database Systems*, (20) 2, pp. 149–186. 1995.

[23] Chomicki, J., Imieliński, T. Temporal Deductive Databases and Infinite Objects. Proc. *7th ACM Symposium on Principles of Database Systems,* pp 61–73, 1988.

[24] Chomicki, J., Imieliński, T. Finite Representation of Infinite Query Answers, In *ACM Transactions on Database Systems*, pp. 181–223, (18) 2, 1993.

[25] Chomicki, J.,N iwinski, D. On the Feasibility of Checking Temporal Integrity Constraints. Proc. *12th ACM Symposium on Principles of Database Systems,* pp 202–213, 1993. (full version to appear in JCSS).

[26] Chomicki, J., Toman, D. Implementing Temporal Integrity Constraints Using an Active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, Special section on Temporal and Real-time Databases, Vol. 7, No. 4, 1995.

[27] Clifford J., Croker A. The Historical Relational Data Model (HRDM) and Algebra based on Lifespans. In *Proceedings of the International Conference on Data Engineering*, pages 528–537, Los Angeles, CA, February 1987.

[28] Clifford J., Croker A., Tuzhilin A. On Completeness of Historical Relational Query Languages. *ACM Transactions on Database Systems*, Vol. 19, No. 1, pp. 64–116, 1994.

[29] Clocksin, W. F., Mellish, C.S. Programming in Prolog. Springer Verlag 1987.

[30] Codd, E. F. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[31] Codd, E. F. Relational completeness of database sublanguages. In Rustin, R.(ed.) *Courant Computer Science Symposium 6: Data Base Systems*, pp. 65–98, Prentice-Hall, 1972.

[32] Cousot P., Cousot R. Abstract Interpretation Frameworks. *J. of Logic and Computation*, 2–4, 1992.

[33] Enderton, H. B. A Mathematical Introduction to Logic. Academic Press, 1972.

[34] Ehrenfeucht, A. An application of games to the completeness problem for formalized theories. *Fund. Math.*, 49:129–141, 1961.

[35] Ershov, I. ,L. Mathematical logic. Moscow, Mir, 1984.

[36] Ferrante, J., Geiser, J. R. An Efficient Decision Procedure for the Theory of Rational Order. *Theoretical Computer Science*, 4:227–233, 1977.

[37] J. Ferrante, C. Rackoff. A Decision Procedure for the First Order Theory of Real Addition with Order. *SIAM J. Comp*, 4:1:69–76, 1975.

[38] Fraisse, R. Sur les classifications des systemes de relations. *Publ. Sci. Univ. Alger*, 1:1, 1954.

[39] Gabbay D. Expressive Functional Completeness in Tense Logic. In Mönnich U. *Aspects of Philosophical Logic*, 91–117, 1981.

[40] Gabbay D. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In Banieqbal B., Barringer B., and Pnuelli A. (ed.) *Temporal Logic in Specification*, vol. 398, pp.409–448, Springer Verlag, LNCS 398, 1989.

[41] Gabbay D., Pnueli A., Shelah S., Stavi J. On the Temporal Analysis of Fairness. Proc. *ACM Symposium on Principles of Programming Languages*, 1980.

[42] Gabbay D., Hodkinson I., Reynolds M. Temporal Logic. Mathematical Foundations and Computational Aspects. Vol. 1. *Oxford Logic Guides* 28, Oxford Science Publications, 1994.

[43] Gabbay D., McBrien P. Temporal Logic & Historical Databases. Proc. *International Conference on VLDB*, 1991.

[44] Gantmakher, F. R. The theory of matrices. New York, Chelsea Pub. Co. 1959.

[45] Gao, H., Warren, D. S. A Powerful Evaluation Strategy For CLP Programs. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 1993.

[46] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematic und Physik.* vol. 38, 173–198, 1931.

[47] Gunter, C. Foundations of Programming Languages, MIT Press, Cambridge, MA, 1992.

[48] Hardy, G. H. An Introduction to the Theory of Numbers. Oxford University Press, 1979.

[49] Henkin, L., Monk, J. D., Tarski, A. Cylindric Algebras. North Holland, 1971.

[50] Immerman N., Kozen D. Definability with Bounded Number of Variables. Information and Computation 83, pp.121–139, 1989.

[51] Jaffar J., Maher, M. J. Constraint Logic Programming: A Survey. J. Logic Programming 1994, 19. 20:503–581.

[52] Jensen, C. S., Snodgrass, R. Temporal Specialization and Generalization. *IEEE Transactions on Knowledge and Data Engineering*, 1993.

[53] Jaffar, J., Lassez, J. L. Constraint Logic Programming. Proc. *14th ACM Symposium on Principles of Programming Languages*, pp 111–119, 1987.

[54] Johnson, M. Memoization in Constraint Logic Programming. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 1993.

[55] Kabanza, F., Stevenne, J-M., Wolper, P. Handling Infinite Temporal Data. *Journal of Computer and System Sciences*, (51) 1, 149–186, 1995.

[56] Kamp J.A.W. Tense Logic and the Theory of Linear Order. PhD thesis, University of California, Los Angeles, 1968.

[57] Kamp J.A.W. On the Formal Properties of 'now'. *Theoria* 37:227–273, 1971.

[58] Kanellakis, P. C., Goldin, D. Q. Constraint Programming and Database Query Languages. In *Proc. 2nd TACS*, 1994.

[59] Kanellakis, P. C., Ramaswamy, S., Vengroff, D. E., Vitter, J. S. Indexing for Data Models with Constraints and Classes. Proc. *ACM Symposium on Principles of Database Systems,* pp 233–243, 1993.

[60] Kanellakis, P. C., Kuper, G. M., Revesz, P.Z. Constraint Query Languages, *Journal of Computer and System Sciences*, (51) 1, 26–52, 1995.

[61] Kemp, D. B., Stuckey, P. J. Analysis based constraint query optimization. Proc. *International Conference on Logic Programming*, 1993.

[62] Kreisel, G., Krivine, J. L. The elimination of quantifiers. *Elements of Mathematical Logic*, North-Holland, 1967.

[63] Lim, P., Stuckey, P.J. A Constraint Logic Programming Shell. PLILP 1990.

[64] Lipeck U.W., Saake, G. Monitoring Dynamic Integrity Constraints Based on Temporal Logic. *Information Systems*, 12(2):255–269, 1987.

[65] Lloyd, J. Foundations of Logic Programming. Springer-Verlag, 1987.

[66] Mecca, G., Bonner, A. Sequences, Datalog, and Transducers. in *Proc. 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*,pp. 23–35, 1995.

[67] Mendelson, E. Introduction to mathematical logic. Princeton, N.J., Van Nostrand 1964.

[68] Mumick, I. S. Query Optimization in Deductive and Relational Databases. PhD thesis, Stanford University, 1991.

[69] Prior, A. Time and Modality. Oxford University Press, 1957.

[70] Ramakrishnan, R. Magic Templates. A spellbinding approach to logic programs. J. Logic Programming 1991, 11:189–216.

[71] Reddy, C.R., Loveland D.W. Presburger Arithmetic with Bounded Quantifier Alternation. *Proc. ACM Symp. on Theory of Comp.*, 320-325, 1978.

[72] Renegar, J. On the Computational Complexity and Geometry of the First-order Theory of the Reals. *Journal of Symbolic Computation*, parts I–III, 13(3):255–330, 1992

[73] Revesz, P. Z. A Closed Form for Datalog Queries with Integer Order. Proc. *3rd International Conference on Database Theory*, pp 187–201, Springer-Verlag LNCS 470, 1990.

[74] Revesz, P. Constraint Query Languages. PhD Dissertation, Brown University, 1991.

[75] Revesz, P. Z. A Closed Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints. *Theoretical Computer Science*, vol. 116, no. 1, 117–149, 1993.

[76] Revesz, P. Z. Datalog Queries of Set Constraint Databases. In *Proc. 5th International Conference on Database Theory*, 423–438, 1995.

[77] Revesz, P. Safe Stratified Datalog with Integer Order Programs. In *Proc. First International Conference on Constraint Programming*, Montanari U., Rossi F. eds., Springer-Verlag LNCS 976, Cassis, France, September 1995.

[78] Revesz, P. Constraint Query Languages. A Survey. (unpublished manuscript), 1995.

[79] Rosenstein J.G. Linear Orderings. Academic Press, New York, 1982.

[80] Schmidt, D. A. The Structure of Typed Programming Languages, MIT Press, Cambridge, MA, 1994.

[81] Schmidt, D. A. Natural-Semantics-Based Abstract Interpretation. In Proc. Static Analysis Symposium 1995, Springer LNCS 983.

[82] Sierpiński, W. Teoria Liczb, Part II, Warszaw, 1959.

[83] Snodgrass R. T. The Temporal Query Language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[84] Snodgrass R.T., editor. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 674+xxiv pages, 1995.

[85] Srivastava, D., Ramakrishnan, R., Revesz, P. Z. Constraint Objects. Proc. *Intl. Workshop on Principles and Practice of Constraint Programming*, 218–228, 1994.

[86] Srivastava, D., Ramakrishnan, R. Pushing Constraint Selections. J. Logic Programming 1993:16:361–414, 1993.

[87] Srivastava, D., Ramakrishnan, R., Revesz, P. Constraint Objects. In *Proc. 2nd Workshop on Principles and Practice of Constraint Programming*, 274–284, 1994.

[88] Stavi J. Functional Completeness over Rationals. Unpublished, Bar-Ilan University, Ramat-Gan, Israel, 1979.

[89] Stuckey, P. J., Sudarashan, S. Compiling Query Constraints. Proc. *ACM Symposium on Principles of Database Systems,* 1994.

[90] Swift, T., Warren, D. S. Analysis of SLG-WAM Evaluation of Definite Programs. Proc. *1994 International Logic Programming Symposium*, MIT Press, 1994, 219–235.

[91] Swift, T., Warren, D. S. An Abstract Machine for SLG Resolution: Definite Programs. Proc. *1994 International Logic Programming Symposium*, MIT Press, 633–652, 1994.

[92] Swift, T., Warren, D. S., Chen, W. Operational semantics of SLG evaluation. Tech. report, SUNY at Stony Brook, 1994.

[93] Tamaki, H., Sato, T. OLD-Resolution with Tabulation. Proc. *3-rd International Conference on Logic Programming*, 84–98, 1986.

[94] Tansel A., Clifford J., Gadia S., Jajodia S., Segev A., Snodgrass R. Temporal Databases. Theory, Design, and Implementation. Benjamin Cummings 1993.

[95] Tarski, A. Sur les ensembles définissables de nombres réels, *Fundamenta Mathematicae*, 17:210–239, 1931 (English translation in *Logics, Semantics, Metamathematics,* second edition, Hackett, 1983).

[96] Toman, D., Chomicki, J., Rogers D. S. Datalog with Integer Periodicity Constraints. Proc. *1994 International Logic Programming Symposium*, 189–203, MIT Press, 1994.

[97] Toman, D., Niwinski, D. First-Order Temporal Queries Inexpressible in Temporal Logic. Presented at the Workshop on Theory of Constraint Databases, PODS 1995. (TR UNL-CSE-95-08). To appear in Proc. EDBT'96.

[98] Toman, D. Top-Down beats Bottom-Up for Constraint Based Extensions of Datalog. Proc. *1995 International Logic Programming Symposium*, 98–112, MIT Press, 1995.

[99] Tuzhilin A., Clifford J. A Temporal Relational Algebra as a Basis for Temporal Completeness. Proc. *International Conference on VLDB*, 1990.

[100] Ullman J. D. Principles of Database and Knowledge-base Systems, Vol. 1,2. Computer Science Systems, 1989.

[101] Vardi, M. Y. The Complexity of Relational Query Languages. In *ACM SIGACT Symposium on Theory of Computing,* pp 137–146, 1982.

[102] Vardi M.Y. A Temporal Fixpoint Calculus. In *ACM Symposium on Principles of Programming Languages*, 1988.

[103] Warren, D. H. D. An Abstract Prolog Instruction Set, *Tech. Report 309*, SRI International, AI Center, Palo Alto, 1983.

[104] Weil, A. Basic number theory. 3ed. Springer-Verlag, 1974.

[105] Williams, H. P. Fourier–Motzkin Elimination Extension to Integer Programming Problems. In *Journal of Combinatorial Theory* (A) 21, pp 118-123, 1976.

List of Publications

Refereed journal papers

1. Jan Chomicki and David Toman. Implementing Temporal Integrity Constraints Using an Active DBMS. IEEE Transactions on Knowledge and Data Engineering, Special section on Temporal and Real-time Databases, Vol. 7, No. 4, August 1995.

2. Luděk Matyska and David Toman. Implementation Techniques for Prolog. Informačné systémy 1, 29-59, 1991 (in Czech).

Refereed conference submissions

1. David Toman and Damian Niwinski. First-Order Temporal Queries Inexpressible in Temporal Logic. Presented at the Workshop on Theory of Constraint Databases, PODS 1995 (TR UNL-CSE-95-08). To appear in proc. EDBT'96.

2. David Toman. Top-Down beats Bottom-Up for Constraint Based Extensions of Datalog. Proc. *1995 International Logic Programming Symposium*, 98–112, MIT Press, 1995.

3. Michael H. Bohlen, Jan Chomicki, Richard T. Snodgrass, and David Toman. Querying TSQL2 Databases with Temporal Logic. To appear in proc. EDBT'96.

4. David Toman, Jan Chomicki, and David S. Rogers. Datalog with Integer Periodicity Constraints. Proc. *1994 International Logic Programming Symposium*, 189–203, MIT Press, 1994.

5. David Toman and Jan Chomicki. Implementing Temporal Integrity Constraints Using an Active DBMS. Proc. RIDE'94 Active Database Systems, Houston, TX, pp.87-95, 1994.

6. Luděk Matyska, Adriana Jergová, and David Toman. The Register Allocation in WAM. Proc. 8th ICLP, Paris, pp.142-156, MIT Press, Cambridge, MA, 1991.

7. Luděk Matyska and David Toman. Implementation of Prolog Compiler, Proc. Compilers, Praha, pp.81-104, 1990 (in Czech).

8. Luděk Matyska and David Toman. Implementation techniques for Logic Programming, Proc. SOFSEM'89, pp.141-162, 1989 (Invited Tutorial in Czech)

9. Luděk Matyska and David Toman. Prolog for AI Applications, Proc. AI'89, Praha, pp.191-198, 1989.

10. David Toman. Implementation of freeze in WAM, Proc. LOP'89, pp.95-94, 1989 (in Czech, English version TR MU Brno, 1989).

11. Luděk Matyska and David Toman. Prolog and WAM—A Case Study of implementation, Proc. LOP'88, pp.65-88, 1988.

Other publications

1. David Toman. Query Evaluation in Constraint Databases. Tutorial proposal for the Second International Conference on Principles and Practice of Constraint Programming (CP96).

2. David Toman. Point-based vs. Interval-based Temporal Query Languages. Submitted to ACM PODS'96.

3. Luděk Matyska, Adriana Jergová, Petr Mejzlík, Václav Račanský, David Toman, and Jiří Zlatuška. WAM Based Efficient Implementation of Logic Programming Languages Final Report. IBM Academic Initiative research project. 1992.

4. David Toman. Operational Semantics of Prolog, Semantics of WAM. MS thesis, Masaryk University Brno, 1992.

FOUNDATIONS OF TEMPORAL QUERY LANGUAGES

by

DAVID TOMAN

Mgr., Masaryk University, Brno, Czech Republic, 1992

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1995

Abstract

Temporal Databases are repositories of information dependent on time. The major difference from standard, e.g., relational database systems, is the need of storing possibly infinite objects, e.g., time spans.

In recent years, there have been numerous proposals that introduce time into standard relational systems. Unfortunately, most of the attempts have been based on ad-hoc extensions of existing database systems and query languages, e.g., TQUEL and TSQL. Such extensions often create many problems, when precise semantics needs to be developed, if one exists at all. In a recent survey by J. Chomicki, a clean way of defining temporal databases based on logic was proposed. This methodology views temporal databases as multi-sorted, finitely representable first-order structures. Query languages then became formulas in suitable logics over the vocabulary of such structures. This method has been quite successful, as most of the existing proposals are subsumed by this approach with only minor adjustments for a few special cases.

In this dissertation we try to address several issues connected with this approach and the development of Temporal Databases:

- In most of the current implementations of Temporal Databases a seemingly natural restriction on the structure of relations is imposed—all the relations stored in the database have exactly one or a bounded number of attributes that represent time, e.g., TSQL or TQUEL. Such a restriction naturally corresponds to the single dimensional modeling of time. However, it also raises an obvious question about the generality of such a solution. We show that even in the first-order case, imposing such discipline on a query language affects the overall expressive power and rules out many interesting queries over the database.

- How to store and handle infinite objects, e.g., time spans. Here we use the idea of generalized databases and encode some of the infinite objects using formulas of appropriate logic. We generalize the common requirements on such encodings that make them suitable for query evaluation. We also propose a new class of infinite objects (periodic sets) together with efficient procedures that allow deductive queries to be evaluated over such objects.

- How to efficiently evaluate queries over such encodings. We propose a new query evaluation procedure based on a combination of the resolution method proposed for logic programs with the fixed point computation used for deductive query evaluation. Both theoretical complexity bounds and experimental measurements show the performance gains achieved by using this method.