THE COMPLEXITY OF QUERYING INDEFINITE INFORMATION: DEFINED RELATIONS, RECURSION AND LINEAR ORDER

BY RONALD VAN DER MEYDEN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Graduate Program in Computer Science

Written under the direction of

L.T. McCarty

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

October, 1992

© 1992
Ronald van der Meyden
ALL RIGHTS RESERVED

ABSTRACT OF THE DISSERTATION

The Complexity of Querying Indefinite Information: Defined Relations, Recursion and Linear Order

by Ronald van der Meyden, Ph.D.

Dissertation Director: L.T. McCarty

This dissertation studies the computational complexity of answering queries in logical databases containing indefinite information arising from two sources: facts stated in terms of defined relations, and incomplete information about linearly ordered domains.

First, we consider databases consisting of (1) a DATALOG program and (2) a description of the world in terms of the predicates defined by the program as well as the basic predicates. The query processing problem in such databases is related to issues in database theory, including view updates and DATALOG optimization, and also to the Artificial Intelligence problems of reasoning in circumscribed theories and sceptical abductive reasoning.

If the program is non-recursive, the meaning of the database can be represented by Clark's Predicate Completion, and standard first order theorem proving technology may be used to evaluate queries. However, with recursive definitions such databases are intrinsically second order, and query processing is not even semi-decidable. Nevertheless, the basic queries, which do not contain defined predicates, are decidable. We show that under certain conditions querying this second order form of indefinite information is no more complex than querying indefinite information expressible in first order logic. We also consider the influence of negation and inequality on complexity.

Next, we study databases containing basic atomic facts and facts asserting order relations between points in a linearly ordered domain. Incomplete information about a linearly ordered domain means that the data provide only a partial order, and a query answering requires reasoning about all the compatible linear orders. We show the complexity of this inference problem is in general intractable, but identify a variety of natural conditions under which queries may be answered in polynomial time.

Finally, we consider the effect of combining the two sorts of indefinite information: we study databases containing facts defined using recursive rules with linear order constraints. Applications of such rules include reasoning about concurrent, repetitive actions. In general, even the basic queries are undecidable in this context, but by restricting definitions to a reasonable class, we are able to recapture decidability. Further, under a constraint of "bounded concurrency," query processing is in polynomial time.

Acknowledgements

Debts to many people have been incurred in the course of this work. Most of all, I would like to thank Thorne McCarty for his supervision. The research reported here was germinated in an attempt to answer the questions he posed, even if my attempts to answer them did not always lead in directions he had intended: he never did get me to share in his enthusiasm for intuitionistic logic. He was always available to discuss my latest results and challenge me to refine and extend them. His careful comments on my drafts did much to improve my presentation of this work.

The influence of Tomasz Imielinski on this dissertation will be evident to all familiar with his work. Thanks to Tomasz for his seminars, in which I learnt a great deal of database theory, and for the lively discussions we had during my tenure at Rutgers. I would also like to express my gratitude to the other faculty at Rutgers whose courses and seminars gave me the grounding to pursue this research, particularly Eric Allender, Alex Borgida and Ann Yasuhara.

Thanks also to Eric Allender, Alex Borgida, Tomasz Imielinski, and Alberto Mendelzon, as members of my dissertation committee, and to the anonymous reviewers of a submission to *Theoretical Computer Science* for their comments on the work reported in this dissertation, which led to improvements in its presentation.

To my fellow students, particularly Soraya Abad-Mota, Tony Bonner, Jan Chomicki, Mukesh Dalal, Nitin Indhurkhya, Sunil Mohan, and Kumar Vadaparty, I extend my thanks for their contributions to the intellectual and social environment at Rutgers.

While a student at Rutgers, I was supported at various times by the Barker Graduate Scholarship (a Sydney University travelling scholarhip), a Rutgers Graduate School Excellence Fellowship, and by Teaching Assistantships and Visiting Lecturer positions in the Rutgers Department of Computer Science. Thanks also to Naftaly Minsky for supporting me during the summer of 1988 with a Research Assistantship.

Dedication

To my parents.

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

1. Introduction 1

2. A Survey of Indefinite Information and Complexity 6

2.1. Indefinite versus Definite Information 6

2.2. Complexity of Query Processing 12

3. Recursively Indefinite Databases 18

3.1. Indefinite Information from Defined Relations 18

3.2. Formal Preliminaries 32

3.3. Decidability 36

3.4. Adorned Glue Types 46

3.5. Complexity: Upper Bounds 53

3.6. Lower Bounds 64

3.7. Decidable Intensional Queries 74

3.8. Discussion 76

4. Negation and Inequality 78

4.1. Introduction 78

4.2. Preliminaries 79

4.3. Negating Base Predicates 83

4.4. Negating Defined Predicates 92

4.5. Inequality 95

5. Linear Order 102

5.1. Introduction 102

5.2. Definitions and Preliminary Results 114

5.3. Order Databases with n-ary Predicates 123

5.4. Conjunctive Monadic Queries 129

5.5. Disjunctive Monadic Queries 139

5.6. Discussion 148

6. Recursion and Linear Order 155

6.1. Introduction 155

6.2. Semantic Preliminaries and Basic Results 159

6.3. Glue Types for Order Databases 163

6.4. Decidability of Regular Rules 171

6.5. Regular Databases with Bounded Concurrency 178

6.6. Discussion 183

7. Conclusion 188

References 192

Vita 201

Chapter 1

Introduction

This dissertation studies the computational complexity of answering queries in logical databases containing indefinite information arising from two sources: facts stated in terms of recursive defined relations, and incomplete information about linearly ordered domains.

First, we introduce recursively indefinite databases. These are databases containing facts stated in terms of predicates defined by means of recursive rules. Consider the predicate **path**, defined by the recursive rules

$$\mathbf{path}(x, y) :- edge(x, y)$$
$$\mathbf{path}(x, y) :- edge(x, z), \mathbf{path}(z, y)$$

That is, **path** is the transitive closure of the relation *edge*. Suppose we are told, of some unspecified graph, the single fact $\mathbf{path}(a, b)$, that is, that there exists a path from node a to node b. Then we have somewhat indefinite information: we know that there exists a sequence of edges from a to b, but we do not know the nodes this sequence traverses, nor even its length. Informally, what we know is the infinite disjunction "there exists a sequence of length one from a to b OR there exists a sequence of length two from a to b OR .....". Nevertheless, we are able to draw some inferences from this information. For example, we know that there must exist an edge from *some* node to b, although we cannot say exactly which node.

Formally, we consider databases consisting of two components: (1) a Datalog program (that is, a logic program without function symbols) which defines certain abstract predicates in terms of more primitive basic predicates, and (2) a description of the world consisting of a set of ground atoms in the defined as well as the basic predicates. Defined facts result in indefiniteness because there may be many distinct configurations of

basic facts supporting any given defined fact, corresponding to different clauses of the definition. We describe a semantics of such databases which requires that in all models the extension of defined relations correspond to that assigned by the least fixpoint operator of the program, applied to the basic facts holding in the model.

The query problem for databases containing defined facts is related to various questions in database theory, including view updates, Datalog optimization and reasoning about logic programs. There are also connections to problems in Artificial Intelligence, including sceptical abductive reasoning and reasoning in circumscribed theories.

In the case that the program is negation-free and non-recursive, the meaning of such a database is that obtained by replacing the program by Clark's Predicate Completion, which turns the rules, considered as sufficient conditions, into necessary and sufficient conditions. Thus, the database is equivalent to a first-order theory and standard theorem proving technology may be used to evaluate queries. However, databases containing recursive definitions are, in general, intrinsically second order. In the case of the example above this is reflected in the well-known fact that transitive closure may not be expressed in first order logic.

This greater expressiveness comes at a cost: query processing becomes undecidable, in fact not even semi-decidable. This is not unexpected, in view of the fact that recursive definitions express infinitely disjunctive information, as illustrated by the example. What is more surprising, is that the basic queries, that is, those queries which do not contain defined predicates, are decidable. One can even prove decidability of a slightly larger class of queries, in which predicates defined by monadic Datalog programs may also occur. A central concern of this dissertation is to analyze the complexity of this decidable class of queries.

We analyze query complexity as a function of various parameters, including size of the data, query size, arity of predicates and linearity or non-linearity of rules. One of the measures of complexity we use is data complexity, the complexity of answering a fixed query as the data varies. While the complexity is in general intractable, a surprising result is that under certain very reasonable conditions the data complexity of querying this second order form of indefinite information is no higher than the data complexity

of querying most types of indefinite information expressible in first order logic.

One of the themes of the dissertation is to identify under what more general conditions the basic queries remain decidable in recursively indefinite databases. One of the generalizations we consider is the influence of negation and inequality ($\neq$). Here the results are somewhat negative: basic queries containing inequality are undecidable even under very restricted conditions. For definitions using negation we show that the decidability of basic queries is retained if only basic predicates are negated in the definitions, but this leads to an exponential increase in complexity. However, if the definitions contain two levels of negation, basic queries are once more undecidable.

The second form of indefinite information we study involves databases containing basic atomic facts and facts of the form $u < v$ asserting order relations between unknown points in a linearly ordered domain. In applications dealing with linearly ordered domains, it is often the case that only some of the order relations between points in the data are known, so that the data provide only a partial order. If we wish to provide a sound and complete query answering service we are faced with the inferential problem of determining what holds under all the compatible linear orders.

For example, suppose we know that Mrs Indira Gandhi's Prime Ministership of India preceded Mr Morarji Desai's Prime Ministership, although we are not sure of their precise periods of tenure. Later, we meet someone who asserts with some certainty that Mrs Gandhi was Prime Minister after Desai, but is also unsure of the exact dates. Then we can draw the conclusion that someone was Prime Minister of India twice. Formally, we know the following set of facts:

$$PM(\text{Gandhi},t_1), PM(\text{Desai},t_2), t_1<t_2,$$
$$PM(\text{Desai},u_1), PM(\text{Gandhi},u_2), u_1<u_2$$

where t_1, t_2, u_1 and u_2 are constants referring to unknown points in time. Notice that the data is only partially ordered. For example, we do not know which of the relations $u_1<t_1, u_1 = t_1$ or $u_1>t_1$ holds. The conclusion is obtained by reasoning as follows. If Gandhi was not Prime Minister twice, then it must be the case that $t_1 = u_2$. But this would mean that Desai was Prime Minister twice. Thus, in any case, someone held office more than once.

Applications that involve query processing on this sort of data include sequencing, scheduling, and non-linear planners in AI. There are also connections to database query containment, specifically to the containment problem for queries containing inequalities.

We analyze the complexity of querying data of this form, establishing complexity results that show that, in general, query processing is intractable. Moreover, this intractability persists even if one considers monadic predicates only. Data complexity for queries containing only monadic predicates does turn out to be in polynomial time. Unfortunately, the proof we present of this result is nonconstructive, and does not yield an explicit PTIME algorithm. However, by adding variety of natural restrictions we are able to obtain explicit PTIME algorithms.

Finally, we return to recursively indefinite information, to consider the effect of combining the two sorts of indefinite information introduced in this dissertation: we study databases containing facts defined using recursive rules with order constraints over a linearly ordered domain. Applications of such rules include reasoning about concurrent, repetitive actions. We show that, in general, the combination of recursion and linear order makes even the basic queries undecidable. However, by restricting definitions to a very reasonable class, the regular rules, which are able to express many examples of interest, we are able to recapture decidability. Further, we show that under a constraint of "bounded concurrency," query processing may be done in polynomial time in databases containing recursive defined facts.

The structure of the thesis is as follows. Chapter 2 explains what we mean by indefinite information, and describes some types of indefinite information that have previously been considered in the literature, as well as the assumptions that are usually made in connection with indefinite information. We also introduce here the measures of query complexity we will consider, and review some known results on the complexity of querying indefinite data. Chapter 3 studies the complexity of querying indefinite information arising from facts stated in terms of relations defined using recursive Datalog programs. Chapter 4 considers inequality, and recursive definitions involving negation. Chapter 5 concerns the second type of indefinite information we study: indefiniteness arising from incomplete information about linear order. Finally, Chapter 6 deals with

the combination of recursion and linear order. Chapter 7 lists open problems and suggests possible lines of future research.

Chapter 2

A Survey of Indefinite Information and Complexity

Before introducing the novel types of indefinite information we will study, we explain in this chapter what we mean by indefinite information, by means of two properties that clearly mark out a state of information as indefinite rather than definite. The types of information we study in this dissertation will satisfy these properties. We also survey the types of indefinite information previously studied, and discuss various assumptions frequently adopted in treatments of indefinite information. Next, we introduce the ways in which we will measure query complexity, and present some known results about the complexity of querying indefinite information.

2.1 Indefinite versus Definite Information

First of all, what is *indefinite* information, and what distinguishes it from *definite* information? Broadly speaking, we refer to a piece of information as definite if it gives us unequivocal knowledge about the world. A prototypical instance of definite information is the statement "Acme Co. supplies green widgets for $2 each and Nadir Inc. supplies blue whatsits for $20 each" Given this information, we are able to form a concrete, though incomplete, picture of the world, that enables us to give precise answers to certain questions. For example, asked "What is the colour of widgets supplied by Acme?" we may answer straightforwardly "green", and asked "Who supplies whatsits" we may answer "Nadir". Questions that we are not able to give precise answers to, like "What is the colour of widgets supplied by Nadir?" can be answered "don't know," without danger that we are not telling all we know. Contrast this with the information we have if we know "Either Acme or Nadir supplies green widgets" On the basis of this knowledge we cannot give a definitive answer to the question "Who supplies green

widgets?" While we could answer with "don't know," this answer would not tell all that we know: a more informative answer would be "either Acme or Nadir."

We take this behaviour with respect to queries to be indicative of whether or not an informational state is definite or indefinite. More precisely, suppose that we have a binary relation $\models$ of entailment, such that $D \models \Phi$ holds when the data D entail the query Φ. Then we will consider the data D to express definite information under the following circumstances:

1. Whenever $D \models \Phi_1 \vee \Phi_2$, either $D \models \Phi_1$ or $D \models \Phi_2$.

2. Whenever $D \models \exists \mathbf{x}[\Phi(\mathbf{x})]$, there exists constants $\mathbf{a}$ such that $D \models \Phi(\mathbf{a})$.

The second of these properties is often referred to as the "definite answer property." It states that we may always find a witness when an existential query is entailed by the data.

Examples of definite information are ground atomic facts such as $P(a)$, and conjunctions of such facts such as $P(a) \wedge R(b, c)$. In fact, an even larger syntactic class of formulae of first order logic may be classified as expressing definite information: the *definite Horn clauses* well known from logic programming theory [76]. These are universally quantified implications of the form $\forall \mathbf{xy}[P(\mathbf{x}) \Leftarrow B(\mathbf{x}, \mathbf{y})]$ where $P(\mathbf{x})$ is an atomic formula and $B(\mathbf{x}, \mathbf{y})$ is a conjunction of atomic formulae. The definite answer property plays an important role in the foundations of logic programming, since it assures that answers to queries may be given using a single answer substitution rather than a disjunction of substitutions.

On the other hand, if we are given data expressed using disjunction and existential quantification, then the two properties do not hold. For example, $A \vee B \models A \vee B$, but neither $A \vee B \models A$ nor $A \vee B \models B$, so that property P1 fails. Similarly, if we are told $\exists x[A(x)]$ then we may conclude $\exists x[A(x)]$, but there does not exist a particular constant a for which it follows that $A(a)$ holds. Thus the definite answer property P2 fails when the data is expressed using existential quantification. The definite answer property may also fail on the basis of disjunctive information. For example, $A(a) \vee A(b) \models \exists x[A(x)]$, but there is no definite answer.

There is a semantic reason underlying the fact that the two properties P1 and P2 hold for definite Horn rules, and that is that it is possible to associate with a set of such rules a unique Herbrand model, minimal in the ordering of containment on the set of Herbrand interpretations. This means that queries may very efficiently be answered by testing whether they are satisfied in this minimal model.

On the other hand, indefinite information is semantically marked by the existence of more than one minimal model. For example, the theory $\{A(a) \vee A(b)\}$ has the two minimal Herbrand models $\{A(a)\}$ and $\{A(b)\}$. Similarly, the theory $\{\exists x[A(x)]\}$ has a minimal models $\{A(c)\}$ for every constant c. It follows from this that it is not possible to answer queries on indefinite data by consideration of a single model. One expects, therefore, that indefinite information is more complex to query than definite information. As we will see in the next section, this is indeed the case.

Many different types of indefinite information have been studied. Much of this work has been motivated by attempts to model null values in databases, beginning with Codd [20]. Null values have been placed on a solid semantic footing by Reiter [109], who shows how they may be represented in logical databases using existential quantifiers. Many classes of database containing null values have been considered, from the simple Codd tables, in which each null value is independent of the others, to tables containing "marked" nulls [58], which may occur at several places in the database, with the intention that each occurrence refers to the same unknown object, to more complex tables in which tuples containing null values are subject to a variety of local and global inequality ($\neq$) constraints [1]. OR-objects, studied by Imielinski and Vadaparty [59], are a "range-restricted" sort of null value. Instead of taking its possible values from the entire domain of the database, each OR-object has associated with it a particular domain. A feature of this literature has been the consideration of the complexity of query processing, and we will sketch some of these results in the next section.

Indefinite information has also been considered in the context of theorem proving and logic programming, under the rubric of "non-Horn programs" [91, 131] and "near-Horn programs" [77, 78]. These are logic programs consisting of clauses of the form $[A_1(\mathbf{x}) \vee \ldots \vee A_n(\mathbf{x})] \Leftarrow B(\mathbf{x}, \mathbf{y})$ where the A_i are atoms and B is a conjunction of

literals. The emphasis in this work has been to devise theorem proving and query answering techniques which operate efficiently in practice, rather than achieve optimal complexity.

The distinction we have drawn between definite and indefinite information is sometimes expressed in the database literature using the terms "incomplete" instead of "indefinite" and "complete" instead of "definite." We prefer to use these terms for a different distinction, referring to the quantity of information we have about the world, rather than to its quality. A state of complete information, for us, is one of total information about the world (or some aspect thereof), so that every question about the world (or about that aspect of the world) can be answered either "yes" or "no" on the basis of this information. Suppose for example, that we know that (1) Acme supplies green widgets, and nothing else, (2) Nadir supplies blue whatsits, and possibly some other things. Then we have complete information about what is supplied by Acme. Asked "Does Acme supply whatsits?" we may give the precise answer, "no". However, we have incomplete information about what is supplied by Nadir: we can only answer the question "Does Nadir supply widgets?" with "don't know."

Because having complete information corresponds to knowing that (part of) the world is in a single possible state, it is one way of having definite information. On the other hand, incomplete information need not be indefinite. For example, although our information about Nadir is incomplete, it is not indefinite. Asked "Does Nadir supply either widgets or whatsits?" we may give the definite answer "Yes, they supply whatsits."

A device which is frequently used to turn definite, but incomplete information into complete information is Reiter's "Closed World Assumption" [108, 109]. The motivation for this assumption is the observation that in most domains the number of negative facts significantly outnumber the number of positive facts. This means that the domain can most succinctly be represented by explicitly representing the positive facts only, and leaving the negative facts to follow implicitly. That is, we assume that if the truth of a fact does not follow from the database, it must be false. Formally, applying the Closed

World Assumption to a logical theory T results in the database

$$CWA(T) = T \cup \{\neg P \mid P \text{is an atom such that } T \models P\}$$

in which we have added all the implicit negative facts. Reiter shows that if the theory T consists of a set of Horn clauses, then the closed world database is consistent. Clearly the Closed World Assumption yields complete information, since for each atom P we must have either $T \models P$ or $T \models P$ and in the latter case we have $CWA(T) \models \neg P$.

When the Closed World Assumption is applied to a theory representing indefinite information, it often produces as result an inconsistent theory. A simple example of this is the theory consisting of a single disjunct $T = \{A \vee B\}$. Suppose we are dealing with a propositional language with propositional letters $\{A, B, C\}$. Then since neither A nor B nor C is individually entailed by this theory, the Closed World Assumption requires that we add the negations of these facts, yielding the theory $CWA(T) = \{A \vee B, \neg A, \neg B, \neg C\}$, which is clearly inconsistent.

Since one would still like to avoid having to state a large number of negative facts even when dealing with indefinite information, it is desirable to modify the CWA so as to apply to indefinite states of information. This has been done by Minker [90], who formulate the Generalized Closed World Assumption. Intuitively, the GCWA operates by restricting the set of models of a theory to those in which the set of atoms holding cannot be further reduced while satisfying the theory. That is, whereas the CWA associates to a definite state of information its unique minimal model, the GCWA associates to an indefinite state of information the set of all its minimal models. For example, the GCWA associates with the theory $T = \{A \vee B\}$ the two minimal models $\{A, \neg B, \neg C\}$ and $\{\neg A, B, \neg C\}$. The Generalized Closed World Assumption yields a consistent theory whenever it is applied to a consistent theory, so it overcomes the problem of inconsistency we noted above for the Closed World Assumption. Observe that since $GCWA(T)$ has two distinct and incomparable models, we are still dealing with an indefinite state of information. Syntactically, this is marked by the fact that $GCWA(T)$ entails $A \vee B$, but separately entails neither A nor B. However, $GCWA(T)$ has some consequences not entailed by the original theory, for example $\neg C$ and $\neg A \vee \neg B$.

Thus we might say that the result of applying the Generalized Closed World Assumption is a more complete, but still indefinite, state of information.

Note that the extra consequences obtained from the Generalized Closed World Assumption all involve negation. If Φ is a "positive formula", not involving negation (we will give a more precise description of this class of formulae later) then $T \models \Phi$ if and only if $GCWA(T) \models \Phi$, since every minimal model of T is still a model of $GCWA(T)$.

In this dissertation we will focus on queries which are positive formulae, so we will not adopt any form of closure assumption intended to turn the incomplete states of information represented by our databases into more complete states of information. Thus, we adopt what is sometimes referred to as the "Open World Assumption": for the semantics of a database represented by a theory T, we take the set of all models of T, not some reduced set of models. However, it turns out that even when dealing with positive queries it is convenient to deal with minimal models when constructing decision procedures, so we will still have a use for these. Further, as we have just remarked, when dealing with positive queries, either assumption will do.

Closely related to the "Closed World Assumption" is the "Domain Closure Assumption" which states that the objects referred to in the database are all the objects that exist. Formally, this assumption amounts to the inclusion of an axiom of the form

$$\forall x [x = c_1 \vee x = c_2 \vee \ldots \vee x = c_n]$$

where the c_i are all the constants in the database. We will not make this assumption at any stage in our work.

Another assumption that is frequently adopted in the context of logical databases is the "Unique Names Assumption", which states that distinct constants in the logical theory refer to distinct objects. We implicitly assumed something of this nature when we gave the answer "no" to the question "Does Acme supply whatsits?" on the basis of the information that Acme supplies widgets only. If whatsits and widgets are the same thing, then this answer is incorrect. Formally, the Unique Names Assumption corresponds to including in the logical theory the formula $c_i \neq c_j$ for each pair of distinct constants (i.e., $i \neq j$). We will consider the effect of this assumption on our

databases in Chapter 3, but its effect will be shown to involve only minor modifications of our query answering procedures, so we will neglect it thereafter.

Finally, we remark here that the fact that information is expressed using ground atoms does not necessarily imply that we have definite information. This occurs when there is certain background information, not explicitly stated, through which these atomic formulae imply indefinite information. We will present two examples of this in this dissertation. In Chapter 3 we will see how indefiniteness results when the predicates used in these atoms are defined in terms of more primitive predicates. In Chapter 5, we show that incomplete information about linearly ordered domains results in indefiniteness. In this case the indefiniteness results from background information we have about linear order.

2.2 Complexity of Query Processing

A central concern in this dissertation will be the complexity of querying the various types of indefinite information we introduce. In this section we introduce the measures of complexity we will use. We also summarize in this section some of what is already known about the complexity of querying various types of indefinite information.

The class of queries we consider throughout this dissertation is the class of *positive existential* formulae of first order logic. Formally, the positive existential formulae are defined as follows:

1. Any atomic formula $P(\mathbf{x})$ is positive existential.

2. If $\Phi_1(\mathbf{x}$ and $\Phi_2(\mathbf{y}$ are positive existential, then so are the formulae $\Phi_1(\mathbf{x}) \wedge \Phi_2(\mathbf{y})$, $\Phi_1(\mathbf{x}) \vee \Phi_2(\mathbf{y})$ and $\exists y[\Phi_1(\mathbf{x})]$.

A positive existential query is said to be *conjunctive* if it is formed without the use of disjunction.

We will be concerned only with the "Yes-No" query answering problem. That is, all our queries will be *closed* formulae Φ, rather than formulae $\Phi(\mathbf{x})$ with free variables. Thus, we are interested in being able to determine if the data D entail the query Φ

(written $D \models \Phi$) rather than in computing a set of answers $\mathbf{x}$ for which $\Phi(\mathbf{x})$ holds. This assumption is frequently made when dealing with indefinite information, since as we discussed in the previous section, indefinite data may entail a query of the form $\exists x[\Phi(x)]$ without entailing $\Phi(a)$ for any particular value a. A fully adequate account of how to answer open queries in indefinite databases would involve "disjunctive answers" of the form $a_1 \vee a_2 \vee \ldots \vee a_k$. While there have been investigations of how to provide such answers [37], this problem is generally avoided, as a first approximation. It is far from clear that an account of disjunctive answers is even possible in the context of the recursively indefinite databases, to be introduced in the next chapter. We note, however, that the problem of computing the set of definite answers $\{\mathbf{a} \mid D \models \Phi(\mathbf{a})\}$ for the query $\Phi(\mathbf{x})$ straightforwardly reduces to the "Yes-No" query answering problem.

We will measure "Yes-No" query complexity by placing query problem within a complexity class, and proving both upper and lower bounds. We represent a query problem as a set S and look for a complexity class C such that $S \in C$. This provides an upper bound for the complexity of the query problem. We will also seek lower bounds, in order to show that the class C provides a tight characterization of the complexity of the problem S. This is done by proving completeness of S for the class C, i.e., showing that S is in some sense as hard as all the problems in C.

The most obvious way to represent a query problem as a set is as follows: given a class $\mathcal{D}$ and a class $\mathcal{Q}$ of queries, we define the answer set

$$AS_{\mathcal{D},\mathcal{Q}} = \{\langle D, \Phi\rangle \mid D \in \mathcal{D}, \Phi \in \mathcal{Q}, D \models \Phi\}$$

In determining the complexity of this set, we ask "given as input a database D and a query Φ in the appropriate classes, how complex is it to determine if the query is entailed by the data?" Following Vardi [123], we call this measure of complexity the *combined complexity* of the query problem.

While combined complexity provides us with an overall view of the complexity of querying a certain class of databases, it is not always the most realistic measure of complexity. It turns out that even conjunctive queries in definite relational databases have NP-hard combined complexity, and hence are unlikely to be practical according to

this measure! This is contrary to the fact that this class of data and these queries are in broad use. The explanation for this is that combined complexity uses a very poor model of the sorts of query problems that are encountered in practice.

In database applications, one finds that the size of the database can grow to be very large, as data accumulates over time. Queries, on the other hand, are composed on-line, and are therefore likely to be very small compared with the size of the database. The hardest instances of the query problem generally involve queries roughly equal in size to the data, and these instances contribute to combined complexity. Thus, combined complexity may be high only because of problem instances that are extremely unlikely to occur in practice.

A way to take the imbalance in query size and data size into account is *data complexity*, which measures the complexity of answering a fixed query as the data varies. Formally, the complexity of a particular query Φ with respect to a class of databases $\mathcal{D}$ is the complexity of the answer set $AS_{\mathcal{D}}(\Phi) = \{D \mid D \in \mathcal{D}, D \models \Phi\}$. We will also speak of the data complexity of a class of queries $\mathcal{Q}$ with respect to a class of databases $\mathcal{D}$. This is the maximal data complexity with respect to $\mathcal{D}$ of queries in $\mathcal{Q}$.

Data complexity may be understood in a number of different ways. Most obviously, data complexity is the appropriate measure of complexity in situations in which it is necessary to ask the same query repeatedly as the data varies. Here, since the query does not change, the only contribution to complexity comes from the size of the data. However, data complexity is still a useful measure of complexity when one is interested in ad hoc queries. It is reasonable to assume a fixed bound on the size of ad hoc queries (the length of the largest query that may be written by the average person in an hours time, say). There will be only a finite number of queries of this size, albeit a large number. The combined complexity of answering queries from this finite set will be equal to the data complexity of the most complex query in the set. Thus, the data complexity of the infinite class of all queries will provide an upper bound on the complexity of queries from this finite set.

We should warn however, that data complexity has its peculiarities, and low data complexity does not always imply that a class of queries is practical. First, the constants

of proportionality involved in data complexity results may be extremely large for all but the smallest queries. More seriously, we will see an example in Chapter 5 in which it is possible to establish linear time data complexity, but without explicitly supplying an algorithm that answers queries with this low complexity. Until such an explicit algorithm can be found, the linear time data complexity characterization has no utility.

For completeness, we also consider the contribution to complexity due to the size of the query. The answer set of a *database* D with respect to a class $\mathcal{Q}$ of queries is the set

$$AS_Q(D) = \{\Phi \mid \Phi \in Q, D \models \Phi\}$$

of queries satisfied by the database. The *expression complexity* of a database is the complexity of the set $AS_Q(D)$. This is a measure of the complexity of query answering as a function of the size of the query.

Let us now briefly summarize some known results about the complexity of querying a variety of types of definite and indefinite information, in order to provide a yardstick against which the results we present in this dissertation may be measured. We confine ourselves to results for first order positive existential queries, both because this is the class of queries we study in this dissertation and because complexity results on more general classes of queries are rather sparse (the work of Vardi [123], who considers arbitrary second order queries, being one of the few exceptions.)

Table 2.1 shows results for the complexity of positive existential first order queries in a variety of types of data. Each entry provides a complexity class for which the corresponding query problem is complete with respect to log-space transformations. In the case of data complexity, this is to be interpreted as follows:

1. For every positive existential query Φ, the data complexity of Φ is in the class indicated.

2. There exists a query Φ with data complexity complete for the class indicated.

That is, it is not necessarily the case that every query has data complexity complete for the class indicated: some queries may have lower complexity. A similar interpretation applies to expression complexity. The first row shows results (from [14, 122]) for definite

	Data complexity	Expression complexity	Combined complexity
Relational Databases	LOGSPACE	NP	NP
"Disjunctive" Databases	co-NP	NP	Π_2^p

Table 2.1: Complexity of positive existential queries

relational databases, that is, databases containing atomic formulae only. We have already mentioned in motivating data complexity that combined complexity in such databases is NP-hard. Notice however, that data complexity in such databases is in LOGSPACE, so (happily) according to this measure querying relational databases is tractable. We interpret this result as an indication that data complexity is a more reasonable more measure of query complexity than combined complexity.

Once one moves from definite databases to databases containing even the most limited forms of indefinite information, the tractability of data complexity is lost (provided P$\neq$co-NP). The second row of Table 2.1 shows the complexity of positive existential queries in a variety of logical databases containing disjunctions or null values. These include Codd tables (subject to the Domain Closure Assumption), tables containing null values subject to inequality constraints, OR-object databases, and logical databases containing disjunctive formulae. In some cases, the lower bounds require that queries contain $\neq$ constraints: we refer the reader to the references [1, 57, 59, 123] for precise statements of these results in the various cases. Notice that indefinite information of these forms results in a "jump" in data complexity from LOGSPACE to co-NP, as well as a jump in combined complexity from NP to Π_2^p. While these increases are not known to be strict, it is evident that querying indefinite information is in some sense more complex than querying definite information. Whereas we have efficient algorithms for querying definite information, no such algorithms are known in the case of most sorts of indefinite information.

The sorts of indefinite information studied in this dissertation, particularly recursively indefinite information, are much more expressive than the disjunctive sorts of

indefinite information described above, and the complexity results we obtain will involve complexity classes much higher than NP and co-NP. However, as we will see, under some reasonable restrictions, it will be possible to show that the data complexity does not increase beyond co-NP. Although this is likely to be intractable, we consider such results to be promising, in the sense that they show that there is not an increase in complexity in the move from very simple forms of indefiniteness to highly complex forms of indefiniteness.

Chapter 3

Recursively Indefinite Databases

3.1 Indefinite Information from Defined Relations

In this chapter[1] we introduce the first type of indefinite information we will study. Suppose we have a set of definitions, intended to define the meaning of a set of abstract terms (the *defined relations*) in terms of more primitive terms (the *basic relations*). Given a set of such definitions, it is frequently the case that data comes expressed not just using the basic predicates, but also in terms of the defined predicates. As we now show, this rapidly produces indefinite information.

Example 3.1.1: View Updates: In most database applications it is necessary for reasons of security and user interests to provide different classes of users with different *views* of the database. Consider a database that contains records of the form $Employee(Name, Salary, HealthPlan, DentalPlan)$. While the latter two attributes are the responsibility of the Personnel office and the employee, they are probably of little interest to the CEO. It is desirable to present her instead with the relation $\mathbf{Emp}(Name, Salary)$, which is obtained by deleting the irrelevant attributes from each tuple in the database. More formally, the relation **Emp** may be said to be defined by the rule

$$\mathbf{Emp}(Name, Salary) : -Employee(Name, Salary, HP, DP) \qquad (3.1)$$

Here we may consider the predicate *Employee* to be a basic predicate and the predicate **Emp** to be a defined predicate.

[1] The material presented in this chapter is based on [89]

Since one of the purposes of views is to relieve the user from the complexities of how the data is actually represented in the database, a view should behave as if it were itself the database. In particular, it is desirable to allow the user to update a view. This leads to what is known as the "view update problem": updates of a view in general underdetermine the corresponding update of the underlying database. In general, many different underlying database states correspond to the view state produced by the view update.

Consider the effect of the CEO's decision to hire Joe Average at $ 20K. When she inserts the tuple **Emp**(Joe Average, $20K$) into her view, it is necessary to reflect this update in the relation *Employee* by inserting a tuple into this relation with attributes $Name$ = Joe Average and $Salary = 20K$. However, the view update gives us no information as to the other two attributes. That is, the fact **Emp**(Joe Average, $20K$) gives us the indefinite information $\exists xy[Employee(\text{Joe Average}, 20K, x, y)]$. □

The approach generally taken to the view update problem is to try to eliminate the indefinite information. In certain circumstances, it is possible to use extra domain semantics to resolve the ambiguities resulting from view updates. For example, [6] exploits the idea of a *complementary view*, which together with the view being updated fully determines the database state. Under the reasonable assumption that the complementary view is unaffected by the update, this means it is possible to compute a unique updated database as the result of a view update.

Unfortunately, even when extra domain semantics is exploited, view updates still frequently underdetermine the underlying database state. In the example, it would be an error to insert particular values for the missing attributes into the database, since the null value acts as a signal to the personnel department that there is a matter to be resolved with the new employee: he must complete the paperwork necessary to join the health and dental plans of his choice. Thus, there are clear cases in which rather than trying to eliminate indefiniteness, we must learn to live with it.

Let us suppose all updates are *insertions* into views. That is, only updates which add to the information in the database are permitted. Then a simple solution to the

view update problem is as follows: instead of trying to reflect a view update in the basic relations, simply store the update. In the example, this means that we store, besides the tuples originally in the database, the new fact **Emp**(Joe Average, $20K$), together with the rule (3.1) which enables us to interpret this fact in terms of the basic relations.

In general, this approach to view updates involves managing informational states consisting of the original set B of basic facts, the definitions Σ of the views and the new facts D stated in terms of the defined relations. One of the central concerns of this work is: how complex is it to query indefinite information expressed in this form? That is, we deal with the following inferential problem: given that the basic facts B and the defined facts D hold, where the latter are interpreted according to the definitions Σ is the query Φ necessarily true? In particular, we focus on definitions Σ expressed using *recursive* logic programs. As we shall see, this leads to some highly indefinite information, which we call *recursively indefinite information.* Accordingly, we call the triples $\Sigma, [B, D]$ *recursively indefinite databases.*

However, let us first consider some more mundane examples. Example 3.1.1 shows that defined facts may express information involving existential quantification. Defined facts also lead to disjunctive information. For, suppose we have a definition consisting of a number of clauses

$$\begin{aligned} \mathbf{A} &: -A_1 \\ &\vdots \\ \mathbf{A} &: -A_n \end{aligned}$$

where the A_i are basic propositions. Then the defined fact **A** is true just when one of the clauses is true. That is, **A** expresses the disjunction $A_1 \vee \ldots \vee A_n$.

In fact, when the definitions are non-recursive, defined facts are equivalent to a first order formula involving existential quantification and disjunctions. To see this, let us consider how to compute the possible states of the basic relations that are expressed by the defined facts. Consider the definitions

$$\begin{aligned} \mathbf{R}(x, z) &: -\mathbf{S}(x, y), A(y, z) \\ \mathbf{S}(x, y) &: -B(x, y) \\ \mathbf{S}(x, y) &: -C(x, y) \end{aligned}$$

of the predicates $\mathbf{R}$ and $\mathbf{S}$. Suppose we are told the fact $\mathbf{R}(a, b)$. Then we may reason as follows: The only way that we know for the fact $\mathbf{R}(a, b)$ to hold is when the body of the first rule is satisfied with $x = a$ and $z = b$. That is, we must have $\exists y[\mathbf{S}(a, y) \wedge A(y, b)]$. This formula itself contains a defined fact $\mathbf{S}(a, y)$, but we know that this will hold if either $B(a, y)$ or $C(a, y)$. This means that the defined fact $\mathbf{R}(a, b)$ is equivalent to the formula $\exists y[B(a, y) \wedge A(y, b)] \vee \exists y[C(a, y) \wedge A(y, b)]$.

Notice that in "expanding out" the meaning of a defined atom, we apply the rules in the reverse of the usual deductive direction. The term "abduction" was introduced by the philosopher C.S. Peirce [99] to refer to this sort of inference from consequents of rules to their antecedents, or more generally, from effects to their causal explanations. Abduction has become a central concern of artificial intelligence since the early work of Pople [103], motivated by medical diagnosis. In this application, defined predicates correspond to observable symptoms, and basic predicates correspond to the underlying medical condition. The rules state known causal links between diseases and their symptoms. Given a set of observed symptoms, it is necessary to reason backwards using the rules, constructing a hypothetical explanation consisting of a set of diseases that would result in the symptoms observed.

Motivated by diagnostic problems in a variety of domains, formalizations of abductive inference have been developed in a number of distinct logical frameworks [102, 101, 72]. Most of this work focuses on the problem of computing a "preferred" explanation. Examples of the sort of preference criteria that have been considered include minimality of the set of basic atoms in the explanation [12], explanation with the highest probability [98], and minimality of the set of "abnormal" components [110].

Most closely related to our work in this chapter are the logic programming formulations of abduction, which note that explanations may be obtained as the "dead ends" of attempted linear resolution refutations [103, 26, 34]. These authors frequently consider the abduced explanation as suggesting a "discriminating experiment" which will generate further information. Thus, in some sense they take a sceptical attitude. The inference problem we study corresponds to the sceptical attitude to abduction, in that we accept only those conclusions that hold under *all* the possible explanations of the

defined facts.

When dealing with theories stated as logic programs, the semantics of abductive inference seems to be generally understood in terms of Clark's predicate completion [19]. Abduction is of course an unsound pattern of logical inference, but it appears that on the sceptical view it becomes sound when one adds the formula

$$\mathbf{R}(\mathbf{x}) \Rightarrow \{B_1 \vee B_2 \vee \ldots \vee B_n\} \tag{3.2}$$

which asserts that each defined fact $\mathbf{R}(\mathbf{x})$ is explained by one of the rules $\mathbf{R}(\mathbf{x}) :- B_i$ for $i = 1 \ldots n$ in the definition of the predicate $\mathbf{R}$. Clark used exactly these formulae, (plus some others intended to restrict the class of models to Herbrand interpretations) to give a semantics for negation as failure in logic programming. When dealing with non-recursive definitions, these rules give an adequate formalization of sceptical abductive reasoning. However, matters are not so clear in the recursive case, as we now show.

Example 3.1.2: Homo ergo Deus: Dave Cart, a computer scientist with a philosophical bent, offers the following proof of the existence of God: "A human is either someone created in the image of a God, or the child of a human. I am human. Therefore, a God exists." Formally, the definition of **human** is given by the rules

$$\mathbf{human}(x) :- Made\text{-}in\text{-}image(x, y), God(y)$$
$$\mathbf{human}(x) :- Parent(x, y), \mathbf{human}(y)$$

If we know $\mathbf{human}(Dave)$ then, so the argument goes, it follows that $\exists x[God(x)]$, since in ascending Dave's family tree we must eventually reach a human without parent, who must have been created in the image of a god. Is Dave's argument valid? □

Under the completion interpretation of the definitions, the conclusion does not follow. For, consider a model in which the only facts holding are $\mathbf{human}(Dave)$ and $Parent(Dave, Dave)$. Clearly this model satisfies the definition above, as well as the rule

$$\forall x[\mathbf{human}(x) \Rightarrow \{\exists y[Parent(x, y) \wedge \mathbf{human}(y)]\}$$
$$\vee \exists y[Made\text{-}in\text{-}image(x, y) \wedge God(y)]\}]$$

Thus, the model satisfies the completion. But there is no God in this model. It might be objected to this that no-one can be their own parent, but this objection is easily met: consider the model

$$\mathbf{human}(Dave), Parent(Dave, a_1), \mathbf{human}(a_1), Parent(a_1, a_2), \mathbf{human}(a_2), \ldots$$

in which we have an infinite chain of Dave's ancestors. Once again it is readily verified that this satisfies the completion, but does not involve the existence of a God.

Clearly this is not what Dave intended. His rebuttal to these counterexamples would be that in the two models described, he does not deserve to be called human. No derivation, using the definitions of the predicate **human**, supports the humanity of Dave, given the basic facts about Gods, creation and parenthood in these models. On this way of understanding the assertion $\mathbf{human}(Dave)$, the models above do not count as counter-examples. In fact there can be no such counterexample under this interpretation, because the fact $\mathbf{human}(Dave)$ must be supported in each model by a finite derivation, which must begin with the existence of a deity.

This illustrates that there are two inequivalent interpretations that can be given to recursive definitions. The first of these is to assign to the database $\Sigma, [B, D]$ the first order theory $Comp(\Sigma, \mathcal{Q}) \cup B \cup D$, in which we add the completion axiom (3.2) for each defined relation. The second, and the one we adopt, is to interpret each defined fact in D to require the existence of a "derivation" from the basic facts, using the rules Σ. While the completion interpretation may be reasonable for some applications, we believe that the second interpretation accords better with our intuitions. We describe in Section 3.2 a semantics for recursively indefinite databases which is based on the least fixpoint semantics for logic programs. Informally, this semantics restricts the set of permitted models to those in which the interpretation of the defined relations is that obtained by applying the least fixpoint operator associated with the program to the basic relations. That is, this semantics requires that the interpretation of the defined relations is the minimal possible interpretation satisfying the rules, given the set of basic facts holding in the model. The following are some more examples of the sort of inference that holds under this semantics, but not under the completion semantics.

Example 3.1.3: Let G be a directed graph some of whose nodes are colored red or green. A 'red-green path' is any path through G each of whose nodes is colored either red or green. The following logic program defines the relation $\mathbf{rgpath}(x, y)$: 'there exists a red-green path from x to y'

$$\mathbf{rg}(x) : -red(x)$$
$$\mathbf{rg}(x) : -green(x)$$
$$\mathbf{rgpath}(x, y) : -\mathbf{rg}(x), edge(x, y), \mathbf{rg}(y)$$
$$\mathbf{rgpath}(x, y) : -\mathbf{rg}(x), edge(x, z), \mathbf{rgpath}(z, y).$$

Suppose all that we know about the graph is that there is a red-green path between nodes a and b and that a is colored red and b is colored green. Then we have highly indefinite information about the graph: while we know that a path exists, we know nothing about its length or about the colours of the intermediate nodes. Nevertheless, this is sufficient information for us to conclude that the graph contains an edge from a red node to a green node, as a simple inductive proof shows. Thus, on the basis of the information $red(a), green(b), rgpath(a, b)$, the query

$$\exists xy(red(x) \wedge green(y) \wedge edge(x, y))$$

should be answered 'true'. The information about the defined predicate $rgpath$ gives us extra information about the basic predicates $red, green, edge$. □

Example 3.1.4: Two nodes a and c in a graph are 'equidistant' to a node b if there exists a path p from a to b and a path q from b to c such that the lengths of p and q are equal. The nodes a and c are equidistant to b just when the atom $\mathbf{eq}(a, b, c)$ is derived by the Datalog program

$$\mathbf{eq}(x, y, z) : -edge(x, y), edge(y, z)$$
$$\mathbf{eq}(x, y, z) : -edge(x, x'), \mathbf{eq}(x', y, z'), edge(z', z).$$

If we know then $\mathbf{eq}(a, b, c)$ and $\mathbf{eq}(c, d, f)$ the query

$$\exists x(\mathbf{eq}(b, c, x) \vee \mathbf{eq}(x, c, d))$$

should be answered 'true'. For, suppose that we have paths of length n from a to b and from b to c, as well as paths of length m from c to d and from d to f,

respectively. If $n \leq m$ then $\mathbf{eq}(b, c, x)$ for some x along the path from c to d. The converse relation gives the other disjunct. □

Under the "minimization" semantics just described, recursively indefinite databases are closely related to problems in non-monotonic reasoning. The constraint that the defined predicates are to be interpreted as being computed from the basic predicates using the program may be stated in terms of McCarthy's notion of circumscription [82, 83] as follows. Let $\Sigma(\mathcal{P}, \mathcal{Q})$ be a first order theory containing the sequences of predicates $\mathcal{P} = (P_1 \ldots P_n)$ and $\mathcal{Q} = (Q_1 \ldots Q_m)$. Then the *circumscription* $Circum(\Sigma, \mathcal{Q})$ of the predicates $\mathcal{Q}$ in Σ is the following formula of second order logic:

$$\Sigma(\mathcal{P}, \mathcal{Q}) \wedge \forall \mathcal{Q}' [\mathcal{Q}' < \mathcal{Q} \Rightarrow \neg \Sigma(\mathcal{P}, \mathcal{Q}')]$$

where the order $\leq$ is defined by $\mathcal{Q}' \leq \mathcal{Q}$ when $\forall x (Q_i'(x) \Rightarrow Q_i(x))$ for all $i = 1 \ldots n$. This formula asserts that the denotation of the predicates $\mathcal{Q}$ cannot be decreased while satisfying the theory.

Circumscription of logic programs has been previously studied by Kolaitis and Papadimitriou [65]. They show that the circumscription of a logic program holds in a model just when the defined relations denote the result of applying the least fixpoint operator associated to the program to the base relations. Thus, the minimization semantics for recursively indefinite databases corresponds to interpreting the database $\Sigma, [B, D]$ by the second order theory $Circum(\Sigma, \mathcal{Q}) \cup B \cup D$, instead of the first order theory $Comp(\Sigma, \mathcal{Q}) \cup B \cup D$ given by the completion semantics. The connection between querying recursively indefinite databases and reasoning with circumscribed logic programs may therefore be stated by means of the equivalence

$$\Sigma, [B, D] \models \Phi \text{iff } Circum(\Sigma, \mathcal{Q}) \models \left[\bigwedge B, D\right] \Rightarrow \Phi$$

where $\bigwedge B, D$ denotes the conjunction of the basic and defined facts in D.

Since the completion semantics is first order, we may use any theorem proving method for first order logic to answer queries under this semantics. However, the circumscription is a second order formula, so there is no immediately applicable proof theory for the minimization semantics. A number of authors have considered using

inductive proofs to reason about logic programs [31, 61] and these results are applicable to this problem. In certain cases these inductive proofs can be automated using a version of second order intuitionistic logic programming [84, 85]. However, this line of research has been able to provide only soundness results.

Most of the research on circumscription has concentrated on finding conditions under which the circumscription of a theory is equivalent to a first order theory [74, 75], so that first order theorem proving technology may be applied to circumscribed theories. Kolaitis and Papadimitriou [65] show that the circumscription of a Datalog program (a logic program without function symbols) is first order just in case the program is bounded. Since boundedness of programs is undecidable [39], this seems a very negative result. It is of practical use only for restricted classes of programs for which boundedness is decidable, and even here fails to provide a proof theory for all such programs.

The results of this chapter provide a different approach to reasoning in circumscribed theories. We will show that there is a broad class of queries for which query answering is *decidable* in recursively indefinite databases for *all* circumscribed Datalog programs. This improves upon the approach of finding first order equivalents in two ways: we are able to handle more programs, and we are able to provide decision procedures where reduction to first order form gives only a complete proof theory.

The queries we consider will be first order positive existential formulae. If a query contains occurrences of defined predicates, it will be called *intensional*. *Basic* queries will be those that contain only basic predicates. Unfortunately, the problem of answering arbitrary intensional queries is undecidable. However, if we restrict to basic queries, this problem becomes decidable. One can also show decidability for a slightly larger class of queries, the *monadic* queries. These queries may contain basic predicates and intensional predicates which are defined by a Datalog program containing only monadic defined predicates (e.g., the program of Example 3.1.2) The central concern of this chapter will be to analyze the complexity of various query problems obtained as restrictions of the following result:

Theorem 3.1.5: For monadic queries φ, and for arbitrary Datalog programs Σ, the problem $\Sigma, [B, D] \models \varphi$ is decidable.

As we show in Section 3.7, this decidability result can be shown to follow from a very general result of Courcelle [25] concerning context-free graph grammars. Our main contribution is to analyze the exact complexity of various decision problems. The following is an example of an inference involving a monadic query.

Example 3.1.6: Let **human** be the monadic intensional predicate defined by the program of Example 3.1.2. Define the predicate **ancestor** by

$$\mathbf{ancestor}(x, y) :- Parent(x, y)$$
$$\mathbf{ancestor}(x, z) :- Parent(x, y), \mathbf{ancestor}(y, z)$$

Then given the facts $\mathbf{human}(Lucy)$ and $\mathbf{ancestor}(Dave, Lucy)$, it follows that $\mathbf{human}(Dave)$. □

We have stated our topic of interest as being to reason about indefinite information, but the problems we study are intimately related to optimization problems for database queries. A database query $Q_1(\mathbf{x})$ is said to be *contained* in a query $Q_2(\mathbf{x})$ if for every database instance D, the answer of Q_1 on D (i.e., the set of tuples $\mathbf{a}$ such that $D \models Q_1(\mathbf{a})$) is contained in the answer to Q_2 on D. There is a well-known reduction of query containment for conjunctive queries to the problem of querying a database [14], that involves asserting the query Q_2 as a database, and asking the query Q_1 in this database. This method is illustrated by the following example:

Example 3.1.7: The query $Q_1(x) = \exists yz[A(x, y) \wedge A(z, y)]$ is contained in the query $Q_2(x) = \exists y[A(x, y)]$. This may be seen as follows. First, replace the variable x by the constant X. Next, assert the query $Q_2(X)$, replacing the existential variable y by the constant Y, obtaining the database $D = \{A(X, Y)\}$. Now ask the query $Q_1(X)$ in this database. Since this query is satisfied in the database D with the substitution $y = Y, z = X$, it follows that Q_1 is contained in Q_2. □

This reduction is readily modified to yield a necessary and sufficient condition for containment of Datalog queries, provided we "assert" the query Q_2 as an indefinite database. Formally, suppose we are given a Datalog program Σ containing defined predicates $\mathbf{Q_1}(\mathbf{x})$ and $\mathbf{Q_2}(\mathbf{x})$. Let $\mathbf{X}$ be new constants. Then it is straightforward to show

Proposition 3.1.8: The answer $\{x \mid \Sigma^{\infty}(D) \models \mathbf{Q}_1(x)\}$ is contained in the answer $\{x \mid \Sigma^{\infty}(D) \models \mathbf{Q}_2(x)\}$ for every database D (consisting of basic predicates), if and only if $\Sigma, [\{\mathbf{Q}_2(X)\}, \emptyset] \models \mathbf{Q}_1(X)$.

Thus, querying indefinite data corresponds in a precise way to containment of Datalog queries. We will in fact use this correspondence to apply some undecidability results about Datalog to our database query problems. But, we will give something back in return. Notice that if the query $\mathbf{Q}_2$ is nonrecursive, then the data $\mathbf{Q}_2(\mathbf{X})$ asserted by the reduction is in fact first order expressible, and we can answer the query by reasoning in a finite number of models. This means that containment of an arbitrary query in a nonrecursive query is decidable, as has been previously noted [121]. The decidability of basic queries implies that the converse containment is also decidable. Thus, we obtain the following result[2].

Theorem 3.1.9: It is decidable to determine, given an arbitrary Datalog query $\mathbf{Q}_1$ and a *non-recursive* Datalog query $\mathbf{Q}_2$ whether $\mathbf{Q}_1$ is equivalent to $\mathbf{Q}_2$.

One of the optimization techniques that have been proposed for Datalog programs is to detect the equivalence of recursive queries to nonrecursive queries, since the latter may be more simply evaluated, and have an extremely well-developed theory of optimization. Unfortunately, it is known that it is not decidable whether a recursive query is equivalent to a non-recursive query [39], even for very restricted classes of recursive queries. Theorem 3.1.9 yields a compromise (although a weak one): given a hypothesis that states that a particular non-recursive query is equivalent to a recursive query, it is possible to automatically verify this hypothesis.

We now sketch the complexity results we establish in this chapter. In order to do so, we first need to describe how we apply the definition of the complexity types introduced in Section 2.2, since we are now dealing with definitions as well as facts in the database. Databases $\mathbf{D}$ now consist of the program Σ together with the basic and defined facts. We would like to measure the contribution to query complexity of a variety of parameters of the rules in terms of which the intensional facts are defined.

[2]This result has been independently noted by Chaudhuri and Vardi [16].

This is done by instantiating the definitions of the complexity measures as follows.

Data complexity was defined so as to be parameterized with respect to a class $\mathcal{D}$ of databases. When dealing with recursively indefinite databases, these classes will correspond to various restrictions on the programs permitted to occur as definitions in the database. One of the restrictions we treat will be to consider for each program Σ, the class of databases using only definitions Σ. That is, we fix the program, and write $AS_\Sigma(\varphi)$ for the answer set in this case.

Note that restricting the class of programs corresponds to assuming that there is an *a priori* fixed set of possible types of recursively indefinite information permitted in the database. In particular, the restriction to a single program is realistic if the incomplete information arises from updates to recursively defined views (Example 3.1.1), of which there will be a fixed number. Data complexity with respect to a fixed program is a particularly apt measure for this application, since one crucial problem is how to compute the effect of an insertion in one of the fixed set of views in the remaining views.

Why it is meaningful to consider data complexity for indefinite databases which may vary their definitions perhaps requires some explanation. First, we remind the reader of the discussion above in which we argued that the fact that queries occurring in practice are likely to be bounded in size indicates data complexity as the appropriate measure. Second, let us observe that fixing the set of definitions means that not just the types of recursively indefinite information that may be expressed in the database is limited, but also the types of simply disjunctive (first-order) information. A fixed set of definitions is not even able to express the class of OR-object databases. It is possible to represent the fact that the OR-object a ranges over the k possible values $DOM(a) = \{v_1, \ldots, v_k\}$ by asserting the defined fact $\mathbf{value}(a, v_1, \ldots, v_k)$, given the definition

$$\begin{aligned}
&\mathbf{value}(x, x, y_2, y_3, \ldots, y_k) :- .\\
&\mathbf{value}(x, y_1, x, y_3 \ldots, y_k) :- .\\
&\qquad\vdots\\
&\mathbf{value}(x, y_1, y_2, y_3 \ldots, x) :- .
\end{aligned}$$

However, this will not do to represent OR-objects ranging over a set of $k+1$ or more values. It appears that applications in which it is necessary to represent OR-objects whose domains may be of arbitrary size (or more generally, arbitrary first order disjunctive information) will require that the set of definitions be allowed to vary.

There are other justifications for allowing the set of definitions to vary. It is conventional in "adaptable interfaces" to allow the user to construct his own vocabulary for interaction with the system. In the context of a knowledge base, this would mean that the user can construct his own defined relations, then use these to state his knowledge. Consider also knowledge bases with long lives: as the demands of the application change, it will be necessary to introduce new defined relations. Knowledge Bases representing legislation, cases and administrative practice [9] are one example application in which complex definitions are subject to frequent change. Finally, it is of technical interest in understanding the sources of complexity in query processing to study the effect of various restrictions on the query problem.

When analyzing the complexity of querying databases which may vary their definitions we will consider the following classes C of programs. The class $Prog$ will contain all programs. The class $Arity(k)$ contains all programs whose defined predicates have arity no greater than k and whose rules contain only a fixed set of k constants. A bound on the arity of defined terms is a very reasonable restriction, since the definitions one typically finds in logic programs have very low arity. (Notice though, that the construction used above to represent OR-objects requires rules of unbounded arity.) We will see that this restriction results in the drop of data complexity to unexpectedly low (co-NP) levels. The restriction on the number of constants allowed to occur in rules is a technicality required to achieve this result.

The final class of definitions we consider is the class $Linear$ consisting of all linear programs, that is, programs that contain only one occurrence of a defined relation in each rule. Most recursive definitions encountered in practice are linear, (in fact the usefulness of non-linear recursive rules is often disputed) so once again this is a reasonable restriction. We will see that linearity in general results in a drop in complexity. We will also say that a recursively indefinite database $\mathbf{D}$ is linear if its

	Complexity Type		
Program type	Data	Expression	Combined
Fixed Σ	co-NP	N/A	2-EXPTIME
arity(k)	co-NP	2-EXPTIME	2-EXPTIME
Linear	PSPACE	EXPSPACE	EXPSPACE
all Datalog	EXPTIME	2-EXPTIME	2-EXPTIME

Table 3.1: Complexity of monadic queries

program is linear.

Combined complexity and expression complexity are parameterized with respect to a class of queries. We will consider the classes *Basic* of basic queries and *Monadic* of monadic queries. It will turn out however, that this choice does not affect the complexity result obtained.

Table 3.1 summarizes the complexity results of this chapter. Each entry indicates a class for which the corresponding problem is complete under logspace reductions. These characterizations apply to both basic and monadic queries: in each case the lower bound result can be achieved using basic queries. As we discussed in Section 2.2, the data complexity of databases with most first order forms of disjunctive information is co-NP complete. We show that provided that the program is constrained by bounding the arity of defined predicates, the data complexity for monadic queries in recursively indefinite databases remains in co-NP. This surprising result shows that if one is prepared to live with the complexity of dealing with incomplete information, there is no extra cost in admitting the additional expressiveness of recursively indefinite information.

This (relatively) low complexity does not persist for programs of unbounded arity, however, nor does it apply to the other types of complexity. Note that even with respect to a fixed program the combined complexity may be double exponential time complete. Note also that the results for expression complexity and combined complexity are the same. Thus, the dominant contribution to the complexity of query answering comes from the length of the query.

The structure of this chapter is as follows. Section 3.2 formally defines recursively indefinite databases and the classes of queries we study. In Section 3.3 we show that the general problem of answering intensional queries is undecidable, and establish connections with the theory of context-free graph grammars. In Section 3.4 we introduce the ideas underlying the decidability of monadic queries. Section 3.5 establishes upper bounds on the complexity of monadic queries and Section 3.6 discusses lower bounds. In Section 3.7 we return to intensional queries and show that there exist other interesting classes of intensional queries which are decidable. Section 3.8 concludes by discussing some related technical literature.

3.2 Formal Preliminaries

Let $A, B, C \ldots$ and $\mathbf{R}, \mathbf{S}, \mathbf{T} \ldots$ be disjoint collections of predicate symbols. The former will be called *basic predicates*, the latter *defined predicates.* We use the convention throughout that defined predicates are indicated in bold-face. An *atom* is an expression of the form $P(t_1, \ldots, t_n)$ where P is an n-ary predicate symbol and the t_i are either variables or objects from some domain. Atoms whose predicate is a defined predicate will be said to be *defined atoms.* Atoms whose predicate is a basic predicate will be called *basic atoms.* An atom is *ground* if it contains no variables. An *interpretation* of a set of predicates will be a set S of ground atoms in those predicates. The *domain* $dom(S)$ of an interpretation is the set of objects occurring in some atom of S. A *structure* is a triple $M = \langle i, I, J \rangle$ consisting of an interpretation I of the basic predicates, an interpretation J of the defined predicates plus a function i mapping some set $Const$ of constant symbols to the domain $dom(M) = dom(I) \cup dom(J)$. The union of structures having the same function i interpreting constants is defined by

$$\langle i, I, J \rangle \cup \langle i, I', J' \rangle = \langle i, I \cup I', J \cup J' \rangle.$$

Satisfaction of a formula of first order logic in a structure is defined as usual.

A *Datalog program* with constants from $Const$ is a set of rules r of the form

$$r: \qquad R :- P_1, \ldots, P_n$$

where R is a defined atom and the P_i are either defined or basic atoms, and all the constant symbols which occur in the rule are members of the set $Const$. That is, Datalog programs are logic programs without function symbols. See Examples 3.1.2, 3.1.3 and 3.1.4 for examples of Datalog programs. A Datalog program is *linear* if the body of each rule contains at most one defined atom. If $M = \langle i, I, J \rangle$ is a structure then an *instance* of the rule r with respect to M is any rule obtained from r by replacing each occurrence of a constant $c \in Const$ with $i(c)$ and then substituting an element of $dom(M)$ for each variable of the rule.

Logic programs are usually interpreted over Herbrand interpretations, i.e., structures in which the ground terms of the language of the program are the objects of the domain, and in which each term is interpreted as itself. We will want to deal with equality constraints, so we give a slightly non-standard definition of the semantics of Datalog programs. Each program Σ acts as an operator (also denoted by Σ) on structures $M = \langle i, I, J \rangle$ via the definition

$$\Sigma(M) = \langle i, I, J' \rangle$$

where a defined atom R is an element of J' just in case there exists an instance of a rule of Σ of the form $R :- P_1, \ldots, P_n$ such that each of the P_i is an element of M. This definition generalizes the standard fixpoint operator on Herbrand interpretations of van Emden and Kowalski [32] to general structures. We write

$$\Sigma^\infty(M) = \bigcup_{k<\omega} \Sigma^k(M)$$

for the union of all the finite iterates of Σ.

A *recursively indefinite database* will be a pair consisting of a Datalog program Σ together with a tuple $[B, D]$ of *finite* interpretations of the basic and defined predicates respectively. Recursively indefinite databases will be interpreted in two ways. The *standard* semantics associates to a recursively indefinite database the set of models

$$Mod(\Sigma, [B, D]) = \{\langle i, I, J \rangle \mid i : dom(B \cup D) \rightarrow dom(I \cup J), \\ \langle i, I, J \rangle \models B \cup D \text{ and } \langle i, I, J \rangle = \Sigma^\infty(i, I, \emptyset)\}.$$

That is, a structure $\langle i, I, J \rangle$ is a model of the database just in case all of the basic and defined facts of the database hold there and the interpretation of the defined relations J

is exactly that which is obtained by applying the fixpoint operator to the basic relations I. The *unique names* semantics assigns to the database the set of models

$$Mod_{un}(\Sigma,[B,D]) = \{\langle i,I,J\rangle \in Mod(\Sigma,[B,D]) \mid i(c_1) \neq i(c_2) \text{ for all constants } c_1 \neq c_2\}.$$

Intuitively, the unique names semantics insists that distinct constants refer to distinct objects, whereas in the standard semantics distinct constants may refer to the same object.

Example 3.2.1: Let $\mathbf{D}$ be the recursively indefinite database with Σ equal to the program

$$\mathbf{R}(x,y) :- Edge(x,y).$$
$$\mathbf{R}(x,y) :- \mathbf{R}(x,z), Edge(z,y).$$

defining $\mathbf{R}$ as the transitive closure of $Edge$ and with

$$B = \{Edge(a,b), Edge(b,c)\}$$
$$D = \{\mathbf{R}(a,b), \mathbf{R}(b,c), \mathbf{R}(a,c)\}$$

The structure $M = \langle i,I,J\rangle$ where $I = B$, $J = D$ and $i(x) = x$ for $x \in \{a,b,c\}$ is a model of the database $\mathbf{D}$, i.e., $M \in Mod(\mathbf{D})$. On the other hand, if we let M' be the structure obtained from M by putting

$$J = \{\mathbf{R}(a,b), \mathbf{R}(b,c), \mathbf{R}(a,c), \mathbf{R}(c,a)\}$$

then M' is no longer a model of $\mathbf{D}$. This is because $\Sigma^{\infty}(i,I,\emptyset) \not\models \mathbf{R}(c,a)$, i.e., there is no way to "derive" the atom $\mathbf{R}(c,a)$ from the facts I using the program Σ. Thus, the interpretation of the predicate $\mathbf{R}$ in the model M' is different from that assigned by the program Σ. □

The class of queries we consider will be the *positive existential* sentences of first order logic, over the set of basic and defined predicates. A formula is positive existential if it is an atom, or is formed from positive existential formulae by conjunction, disjunction or existential quantification. A query will be said to be *basic* if it contains only basic predicates. Otherwise the query is called *intensional*.

Associated with the two types of semantics are two consequence relations defined by

$$\mathbf{D} \models \varphi \text{ iff } M \models \varphi \text{ for all } M \in Mod(\mathbf{D})$$

and similarly

$$\mathbf{D} \models^{un} \varphi \text{ iff } M \models^{un} \varphi \text{ for all } M \in Mod_{un}(\mathbf{D}).$$

We note that elements of the domain of the database may occur as constant symbols in the query φ.

It is possible for a recursively indefinite database to be inconsistent, i.e., to have an empty set of models. A defined predicate $\mathbf{R}$ is *empty* in a program Σ if for all structures $M = \langle i, I, \emptyset \rangle$, the structure $\Sigma^{\infty}(M)$ supports no fact of the form $\mathbf{R}(a_1, \ldots, a_n)$. Clearly, if $\mathbf{R}$ is empty in Σ, then $Mod(\mathbf{D})$ is empty for any database $\mathbf{D} : \Sigma, [B, D]$ in which D contains defined atoms in the predicate $\mathbf{R}$. It will follow from results in the next section that $\mathbf{D}$ is inconsistent with respect to the standard semantics just in case some defined predicate occurring in the set D is empty in Σ. This means that inconsistency of recursively indefinite databases with respect to the standard semantics can be tested in polynomial time, since this is the complexity of predicate emptiness [125]. For the unique names semantics, things are a little more subtle.

Example 3.2.2: Suppose that A, B are basic predicates and that a, b are constant symbols. Let Σ be the program with rules

$$\mathbf{R}(x) :- \mathbf{S}(x, b)$$
$$\mathbf{S}(x, x) :- B(x).$$

Suppose that $\mathbf{D}$ is the database $\Sigma, [B, D]$ with basic facts $B = \{A(b)\}$ and defined facts $D = \{\mathbf{R}(a)\}$ and let $M = \langle i, \{A(c), B(c)\}, \{\mathbf{R}(c), \mathbf{S}(c, c)\} \rangle$ where $i(a) = i(b) = c$. Then the structure M is an element of $Mod(\mathbf{D})$, but not an element of $Mod_{un}(\mathbf{D})$, because a and b do not refer to distinct objects. In fact, the database $\mathbf{D}$ is inconsistent under the unique names semantics. Note that the atom $\mathbf{R}(a)$ holds in a model only if $\mathbf{S}(a, b)$ holds there. This in turn can only be the case if $a = b$ and $B(b)$ holds. This also shows that $\mathbf{D} \models \exists x[A(x) \wedge B(x)]$. □

We will not give an explicit algorithm for testing inconsistency with respect to the unique names semantics, but one is implicit in our decision procedures of Section 3.5. Given a database $\mathbf{D}$, choose some 0-ary basic predicate P not occurring in $\mathbf{D}$. Then D is inconsistent with respect to the unique names semantics if and only if $\mathbf{D} \models^{un} P$. Thus,

the procedures we provide for query answering may be used to test for inconsistency of databases.

We will want to discuss intensional queries containing only monadic defined predicates whose definition involves only monadic predicates. This is formulated as follows. Suppose that $\mathcal{P}$ is a set of monadic defined predicates. A program Π in which all defined predicates are in the set $\mathcal{P}$ will be called a *monadic program defining* $\mathcal{P}$. A *monadic query* will be a pair (Π, φ) consisting of an intensional query φ in which all defined predicates are from the set $\mathcal{P}$ and a monadic program Π defining $\mathcal{P}$. Satisfaction of monadic queries is defined on databases $\mathbf{D} : \Sigma, [B, D]$ in which the predicates $\mathcal{P}$ do not occur in the heads of rules of Σ by

$$\Sigma, [B, D] \models (\Pi, \varphi) \text{ if and only if } \Sigma \cup \Pi, [B, D] \models \varphi$$

The condition that no predicate in $\mathcal{P}$ occurs in the head of a rule of Σ is required to ensure that Σ does not modify the definition of the predicates $\mathcal{P}$. It can always be guaranteed simply by renaming the predicates defined by Σ (or Π).

A *homomorphism* $h : M \rightarrow M'$ from a model $M = \langle i, I, J \rangle$ to the model $M' = \langle i', I', J' \rangle$ is a mapping $h : dom(M) \rightarrow dom(M')$ such that for all constant symbols $c \in Const$ we have $h(i(c)) = i'(c)$ and for each atom $A(a_1, \ldots, a_n) \in M$ we have $A(h(a_1), \ldots, h(a_n)) \in M'$. We write $M \leq M'$ when there exists such a homomorphism. The following fact about homomorphisms is well known:

Proposition 3.2.3: Suppose $M \leq M'$. Then for every closed positive existential formula φ, if $M \models \varphi$ then $M' \models \varphi$.

3.3 Decidability

In this section we consider the decidability of queries on recursively indefinite databases. We first introduce the notion of expansions of a database, making use of terminology from the field of graph grammars. We show that the problem of query answering can be reduced to checking the query on all expansions of a database. We then show that binary intensional queries are undecidable, even when the program contains only linear rules.

Hypergraph edge replacement grammars, introduced by Habel and Kreowski [47, 48], are generalizations of context free grammars, generating sets of hypergraphs instead of sets of strings. We present Courcelle's [22] slightly more general version of these grammars. Let $\mathcal{A}$ be an alphabet with a rank function $\tau : \mathcal{A} \to \mathbf{N}$. A *hypergraph* on $\mathcal{A}$ is a tuple $\langle V, E, lab, vert, src \rangle$ where V is a set of vertices, E is a set of 'hyperedges', $lab : E \to \mathcal{A}$ associates a label from $\mathcal{A}$ with each hyperedge, $vert : E \to V^*$ associates to each hyperedge a sequence of vertices and $src \in V^*$ is a sequence of *sources* or 'distinguished vertices' of the hypergraph. We will refer to vertices not equal to any source as *internal* vertices. The *arity* of an edge e is the length of $vert(e)$. We require that each edge e has arity equal to $\tau(lab(e))$. If σ is a sequence then $\sigma[i]$ denotes the i-th element of σ. Thus $src[i]$ refers to the i-th source. If the length of the sequence of sources of a hypergraph is n the hypergraph is called an n-graph and n is called the *type* of the hypergraph. We write $Graph(\mathcal{A})$ for the set of hypergraphs with labels in $\mathcal{A}$.

The operation of *gluing* a k-graph G_2 to an n-graph G_1 at a site given by a sequence σ of vertices of G_1 of length k is defined as follows. Let G_j be a hypergraph $\langle V_j, E_j, lab_j, vert_j, src_j \rangle$ for $j = 1, 2$. We assume that V_1 and V_2 are disjoint, as are E_1 and E_2; if not, rename the vertices and edges to make this the case. Let $\sim$ be the equivalence relation on the set $V_1 \cup V_2$ generated by the equivalences $\sigma[i] \sim src_2[i]$ for $i = 1, \ldots, k$. We write $\{v\}$ for the equivalence class of v. The composition $(G_1, \sigma) \circ G_2$ is the hypergraph $G = \langle V, E, lab, vert, src \rangle$ with vertices $V = (V_1 \cup V_2)/\sim$ and edges $E = E_1 \cup E_2$. Edges retain their original labels, i.e., $lab = lab_1 \cup lab_2$. If $e \in E_j$ and $vert_j(e)$ has length m then so does $vert(e)$, and $vert(e)[i] = \{vert_j(e)\}$ for $i = 1, \ldots, m$ and $j = 1, 2$. The type of G is n and $src[i] = \{src_1[i]\}$. If G_1 and G_2 are both k-graphs, then we will write simply $G_1 \circ G_2$ for the result of gluing G_2 to G_1 at the site given by the sequence of sources of G_1, that is, for the hypergraph $(G_1, src_1) \circ G_2$.

Using the operation of gluing we may define the *substitution* of an edge by a hypergraph. Suppose G_1 is an n-graph and G_2 is a k-graph and let e be an edge of G_1 of arity k. The n-graph $G_1[G_2/e]$ resulting from the substitution of G_2 for the edge e is obtained by deleting the edge e from G_1 and gluing to the result the hypergraph G_2

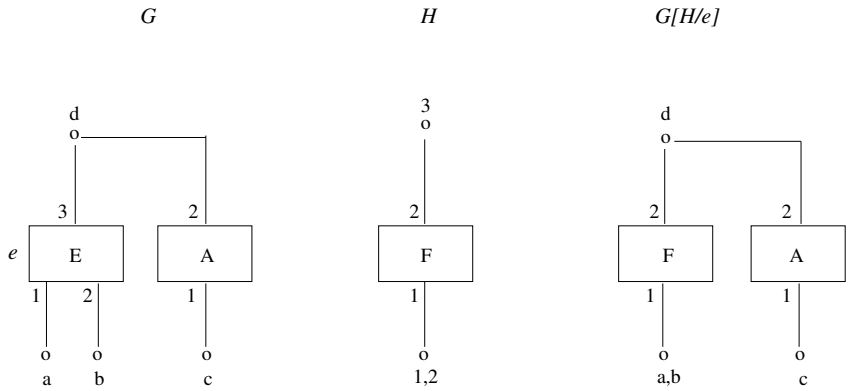

Figure 3.1: Substituting a graph for an edge

at the site $vert(e)$. If θ is a function mapping the defined edges $e_1, \ldots, e_n$ of G to the hypergraphs $H_1, \ldots, H_n$ then we will say that the result of the sequence of substitutions $G[H_1/e_1] \ldots [H_n/e_n]$ is the result of *applying the substitution* θ to the hypergraph G, and write $G\theta$ for this result. Note that, up to renaming of vertices and edges, the result of a sequence of such operations is independent of the order in which they are applied.

Example 3.3.1: Figure 3.1 shows the result of substituting the hypergraph H for the edge e of the hypergraph G. Here edges are represented by rectangles and vertices by circles. The numbering on the lines emanating from edges indicates the sequence of vertices of the edge. Thus $vert(e) = \langle a, b, d \rangle$. The hypergraph H has sources indicated by the numbers 1–3 labelling the vertices of H. Note that since the first and second source of H are equal, the vertices a and b become identified in $G[H/e]$. □

Context free hypergraph edge replacement grammars (simply hypergraph grammars in the sequel) are tuples $\Gamma = \langle \mathcal{A}, \mathcal{U}, P, S \rangle$. Here $\mathcal{A}$ and $\mathcal{U}$ are disjoint finite ranked alphabets with rank function $\tau : \mathcal{A} \cup \mathcal{U} \rightarrow \mathbf{N}$. The elements of $\mathcal{A}$ are called the *terminals* of the grammar and the elements of $\mathcal{U}$ are called the *nonterminals*. The set P of *productions* of the grammar contains pairs of the form (u, G), where u is a nonterminal and G is a $\tau(u)$-graph. Finally, S is an n-graph, called the *axiom* of the grammar. The number n is the type of the grammar.

Associated with each hypergraph grammar is a *derivation* relation on hypergraphs, defined by $H \longrightarrow H'$ if there exists an edge e of H labeled by a nonterminal u such

that $H' = H[G/e]$ for some production $(u, G) \in P$. The relation $\longrightarrow^*$ is the transitive reflexive closure of this relation. A sequence of hypergraphs $S_0 \longrightarrow S_1 \longrightarrow \ldots \longrightarrow S_n$ is called a *derivation* of S_n from S_0. The *language generated* by the grammar is the set $L(\Gamma) = \{H \in Graph(\mathcal{A}) \mid S \longrightarrow^* H\}$. That is, the language generated by the grammar consists of the derivable hypergraphs all of whose edges are labelled by terminals.

We now demonstrate a connection between recursively indefinite databases and hypergraph grammars. We assume that the program Σ contains no constants. (It will be explained below how to remove this assumption.) Suppose we are given a Datalog rule

$$r: \quad \mathbf{R}(x_1 \ldots x_n) :- A_1(u_{1,1} \ldots u_{1,l_1}), \ldots, A_k(u_{k,1} \ldots u_{k,l_k})$$

where the x_i are (not necessarily distinct) variables, the A_i are either defined or basic predicates and the $u_{i,j}$ are variables from the set $Var = \{x_1 \ldots x_n\} \cup \{y_1 \ldots y_m\}$. We define the production $p(r) = (\mathbf{R}, G)$ where G is the n-graph with vertices Var and edges $e_1 \ldots e_k$. We put $lab(e_i) = A_i$. For each $i = 1 \ldots k$, we let $vert(e_i) = u_{i,1} \ldots u_{i,l_i}$. Finally, the sources of the hypergraph G are $src = x_1 \ldots x_n$. Thus, every program Σ has an associated set of productions $P(\Sigma) = \{p(r) \mid r \in \Sigma\}$. We interpret the defined predicates as non-terminals and the basic predicates as terminals.

If $\mathbf{R}$ is a defined predicate of arity n we define the set $L(\Sigma, \mathbf{R})$ to be the language generated by the grammar with productions $P(\Sigma)$ and axiom the hypergraph with n distinct vertices $\{x_1 \ldots x_n\}$, sources $src = x_1 \ldots x_n$ and a single edge e with $lab(e) = \mathbf{R}$ and $vert(e) = x_1 \ldots x_n$. We call elements of $L(\Sigma, \mathbf{R})$ *expansions* of $\mathbf{R}$.

Example 3.3.2: Let Σ be the program of Example 3.1.3. In examples we will write hypergraphs in the format $S = \{xy : \mathbf{rgpath}(x, y)\}$. Here the prefix "$xy$:" indicates the hypergraph has sources $\langle xy \rangle$. There is one edge with label **rgpath** and vertices $\langle xy \rangle$. The following is a derivation of the grammar associated with

the program Σ from the axiom S:

$$\begin{aligned}
S &\rightarrow \{xy : \mathbf{rg}(x), edge(x,z), \mathbf{rgpath}(z,y)\} \\
&\rightarrow \{xy : red(x), edge(x,z), \mathbf{rgpath}(z,y)\} \\
&\rightarrow \{xy : red(x), edge(x,z), \mathbf{rg}(z), edge(z,y), \mathbf{rg}(y)\} \\
&\rightarrow \{xy : red(x), edge(x,z), green(z), edge(z,y), \mathbf{rg}(y)\} \\
&\rightarrow \{xy : red(x), edge(x,z), green(z), edge(z,y), red(y)\}
\end{aligned}$$

Since the final hypergraph contains no defined edges, it is an element of the set $L(\Sigma, \mathbf{rgpath})$. $\square$

Hypergraphs and structures are closely related. A n-graph $G = \langle V, E, lab, vert, src \rangle$ may be interpreted as a structure via the definition $mod(G) = \langle i, I, J \rangle$ as follows. We introduce n constants c_j and interpret these by $i(c_j) = src[j]$ for $j = 1 \ldots n$. If A is a basic predicate the atom $A(u_1 \ldots u_l) \in I$ just when there exists an edge e with $lab(e) = A$ and $vert(e) = u_1 \ldots u_l$. If $\mathbf{R}$ is a defined predicate the atom $\mathbf{R}(u_1 \ldots u_l) \in J$ when there exists an edge e with $lab(e) = \mathbf{R}$ and $vert(e) = u_1 \ldots u_l$. Note that there may be fewer atoms than edges, since this operation removes 'duplicate edges'.

Similarly, any structure $M = \langle i, I, J \rangle$ in which i interprets the constants c_j for $j = 1 \ldots n$ can be straightforwardly interpreted as an n-graph $graph(M)$. This hypergraph has vertices $dom(M)$ and an edge e with $lab(e) = A$ and $vert(e) = u_1 \ldots u_k$ for each basic or defined atom $A(u_1 \ldots u_k)$ in $I \cup J$. The sources are given by $src = i(c_1) \ldots i(c_n)$. A pair $[B, D]$ of sets of basic and defined facts may also be interpreted as a structure, and hence as a graph, by $[B, D] = \langle id, B, D \rangle$ where id is the identity function on constants: $id(c) = c$ for all c.

The correspondence between hypergraphs and structures allows us to apply operations on hypergraphs to recursively indefinite databases. Under this correspondence, atoms correspond to edges, so we will sometimes refer to an edge e with $lab(e) = A$ and $vert(e) = u_1 \ldots u_n$ as the atom $A(u_1 \ldots u_n)$. Conversely, we may refer to atoms as edges when we wish to focus on hypergraph operations. The principle difference between hypergraphs and structures is the possibility of duplicate atoms in hypergraphs. This is necessary to maintain the context freeness of the hypergraph derivation relation.

We now show how to eliminate the assumption that rules do not contain constants.

Suppose the program Σ contains m constants $c_1 \ldots c_m$. For each n-ary defined predicate $\mathbf{R}$ introduce a new $n+m$-ary defined predicate $\mathbf{R}'$. Given a rule r, construct the rule r' as follows. Introduce m new variables $z_1 \ldots z_m$. First, replace each occurrence of a constant c_j in r by the variable z_j. Then, replace each occurrence of a defined atom $\mathbf{R}(u_1 \ldots u_n)$, in either the head or the body of the rule, by the atom $\mathbf{R}'(u_1 \ldots u_n, z_1 \ldots z_m)$. Let Σ' be the program obtained by replacing each rule r in Σ by the rule r'. Similarly, if φ is a query then define φ' to be the query obtained by replacing each occurrence of a defined atom $\mathbf{R}(u_1 \ldots u_n)$ by $\mathbf{R}'(u_1 \ldots u_n, c_1 \ldots c_m)$. (Note that φ is unchanged if it is a basic query.) Finally, given a set D of defined atoms, let D' be the set which contains a defined atom $\mathbf{R}'(u_1 \ldots u_n, c_1 \ldots c_m)$ for each defined atom $\mathbf{R}'(u_1 \ldots u_n)$ in D. Then it is straightforward to show the following:

Lemma 3.3.3: $\Sigma, [B, D] \models \varphi$ if and only if $\Sigma', [B, D'] \models \varphi'$

Note that the program Σ' contains no constant symbols. Thus, this result enables us to apply the techniques we develop for programs without constants to programs which do contain constants. Note, however, that it results in an increase in the arity of defined predicates. Some of our results concern the case of defined predicates of bounded arity, and these need to be carefully interpreted when dealing with programs containing constants: they apply only when the number of constants occuring in the rules is also bounded.

Example 3.3.4: Consider the program of Example 3.2.2. This contains one constant b. The transformed program Σ' is

$$\mathbf{R}'(x, z_1) :- \mathbf{S}'(x, z_1, z_1)$$
$$\mathbf{S}'(x, x, z_1) :- B(x).$$

There is just one expansion G of $\mathbf{R}'$ by Σ':

$$\{x z_1 : \mathbf{R}'(x, z_1)\} \longrightarrow \{x z_1 : \mathbf{S}'(x, z_1, z_1)\}$$
$$\longrightarrow \{z_1 z_1 : B(z_1)\} = G$$

If $D = \{\mathbf{R}(a)\}$ then $D' = \{\mathbf{R}'(a, b)\}$. Substituting the hypergraph G for the defined edge of the hypergraph corresponding to $[\{A(b)\}, \{\mathbf{R}'(a, b)\}]$ results in a

hypergraph corresponding to the structure $\langle i, \{A(b), B(b)\}, \emptyset \rangle$ where $i(a) = i(b) = b$. □

We now show that the structures obtained by substituting for each defined atom in a database an expansion of the corresponding defined predicate form a complete set of models for query answering. We call these structures the *expansions* of the database.

Lemma 3.3.5: Let $\mathbf{D} : \Sigma, [B, D]$ be a recursively indefinite database. For each defined edge $e = \mathbf{R}(a_1, \ldots, a_n)$ of D let $\theta(e)$ be an element of $L(\Sigma, \mathbf{R})$. Then $\Sigma^\infty([B, D]\theta) \in Mod(\mathbf{D})$.

Proof: A straightforward induction shows that the derivation of $\theta(e)$ provides a sequence of inferences showing $e \in \Sigma^\infty([B, D]\theta)$. □

Lemma 3.3.6: Let $\mathbf{D} : \Sigma, [B, D]$ be a recursively indefinite database. For every model $M \in Mod(\mathbf{D})$ there exists a substitution θ mapping each defined edge $e = \mathbf{R}(a_1, \ldots, a_k) \in D$ to an element of $L(\Sigma, \mathbf{R})$, such that $\Sigma^\infty([B, D]\theta) \leq M$.

Proof: (sketch) We show by induction on n that for every defined atom $e \in \Sigma^n(M)$ there exists an expansion G and a homomorphism $h : G \rightarrow M$. If $e \in \Sigma^{n+1}(M)$ then there exists a rule r and a substitution α for the variables of r such that the head of $r\alpha$ is e and all elements of the body are in $\Sigma^n(M)$. By the induction hypothesis this means that there exist expansions and homomorphisms for the defined atoms of the body. The homomorphism for e is constructed from these and the substitution α. □

The following is direct from Proposition 3.2.3 and Lemmas 3.3.5 and 3.3.6. This result shows that we may restrict our attention to the countable set of expansions for query answering. Thus the problem of query answering for arbitrary intensional queries is in Π_1^0.

Lemma 3.3.7: Let $\mathbf{D} : \Sigma, [B, D]$ be a recursively indefinite database. Then $D \models \varphi$ if and only if $\Sigma^\infty([B, D]\theta) \models \varphi$ for all substitutions θ mapping each defined edge $e = \mathbf{R}(a_1, \ldots, a_k) \in D$ to an element of $L(\Sigma, \mathbf{R})$.

Unfortunately, the problem of answering general intensional queries is undecidable even under somewhat restrictive conditions. Undecidability of binary intensional queries is closely related to a result of Shmueli [116] stating that containment of binary Datalog

queries is undecidable. Gaifman et al. [39] developed techniques to prove undecidability of boundedness of Datalog programs which may be used to show that containment remains undecidable for linear programs. Vardi [124] has shown that undecidability of boundedness holds even for rules containing binary defined predicates, the sharpest possible version of this result, since boundedness of unary programs is known to be decidable. The following proof is a straightforward adaptation of Vardi's ideas to show undecidability of binary linear queries.

Proposition 3.3.8: There exists a program Σ containing only monadic and dyadic defined predicates and there exists an intensional query φ such that the problem of deciding $\Sigma, [B, D] \models \varphi$ as the sets of facts B and D vary is undecidable.

Proof: Let M be a universal Turing machine. We describe a program Σ, a query φ and an encoding of inputs to the Turing machine M as sets $[B, D]$ such that for every for every input, $\Sigma, [B, D] \models \varphi$ if and only if M does not halt on the input. Expansions of the database will correspond to computations of the Turing machine.

We represent computations of M as a linear sequence of cells. Portions of this sequence between cells containing the 'end-of-configuration marker' # represent successive configurations of the computation. We let S be a set of symbols which are either a tape symbol of the Turing machine, or a 'hybrid' symbol which indicates both a tape symbol and that the head of the machine is currently at that location, and also provides the state the machine is in. It is well known that with these conventions the contents of any cell in a configuration can be determined from the contents of the three surrounding cells of the preceding configuration. Thus, there exists a relation $Error \subseteq (S \cup \{\#\})^6$ such that a sequence of cells does not represent a valid partial computation of M if and only if there exists a pair of triples abc and def of successive cells in corresponding locations of successive configurations with $\langle abcdef \rangle \in Error$.

For each symbol $a \in S \cup \{\#\}$ we introduce a basic predicate $cell_a(x)$ which intuitively asserts that x is a cell containing the symbol a. Suppose the input configuration for a computation is $a_1 \ldots a_n$. Then we introduce constants $c_0 \ldots c_{n+1}$ to represent the first $n + 2$ cells of the computation, and represent the initial configuration by including $cell_{\#}(c_0), cell_{a_1}(c_1), \ldots, cell_{a_n}(c_n), cell_{\#}(c_{n+1})$ in the set B of basic facts of the

database, together with the basic facts $next(c_i, c_{i+1})$ for $i = 0 \ldots n$ to indicate the order of these cells. The set of defined atoms D of the database consists of the single atom $\mathbf{cellseq}(c_{n+1})$. The expansions of this atom will generate the remainder of the computation. Thus, the program Σ contains for each symbol $a \in S \cup \{\#\}$ which does not indicate that the machine is in the final state a rule

$$\mathbf{cellseq}(x) : -next(x, y), cell_a(y), \mathbf{cellseq}(y).$$

We assume, without loss of generality, that any computation halts with the head at the rightmost tape position. Thus, if h is the symbol indicating the final state then we also have the rule

$$\mathbf{cellseq}(x) : -next(x, y), next(y, z), cell_h(y), cell_{\#}(z)$$

which terminates the expansion. Notice that any expansion of the atom $\mathbf{cellseq}(c_{n+1})$ consists of a linear sequence of cells containing the symbols $S \cup \{\#\}$, such that the last two cells contain h and $\#$ respectively.

The query φ will express 'the sequence of cells is not a valid computation'. Since every expansion of the database described above corresponds to a finite sequence of configurations terminating in a halting configuration, a query expressing this will hold in all expansions just in case the computation starting with the input configuration does not halt. Undecidability of the query is then immediate. Thus, it remains only to show how to detect errors in the expansions. This is done using a binary defined predicate $\mathbf{fing}(x, y)$ which provides a pair of 'fingers' identifying corresponding locations x and y in successive configurations. The following rules define this predicate by showing how to move the fingers to the next pair of corresponding positions. We have an instance

of these rules for all symbols $a, b, c, d \in S$:

$$\begin{aligned}
\mathbf{fing}(x_1, y_1) :- & \mathbf{fing}(x_0, y_0), q_\#(x_0), q_\#(y_0), next(x_0, x_1), next(y_o, y_1) \\
\mathbf{fing}(x_1, y_1) :- & \mathbf{fing}(x_0, y_0), q_a(x_0), q_b(x_1), q_c(y_0), q_d(y_1), \\
& next(x_0, x_1), next(y_o, y_1) \\
\mathbf{fing}(x_1, y_2) :- & \mathbf{fing}(x_0, y_0), q_a(x_0), q_\#(x_1), q_b(y_0), q_\#(y_1), \\
& next(x_0, x_1), next(y_o, y_1), \\
\mathbf{fing}(x_1, y_2) :- & \mathbf{fing}(x_0, y_0), q_a(x_0), q_\#(x_1), q_b(y_0), q_c(y_1), q_\#(y_2), \\
& next(x_0, x_1), next(y_o, y_1), next(y_1, y_2).
\end{aligned}$$

The last two of these rules state the way the fingers move at the end of configurations. Note that the length of a configuration can increase by at most one, since the head can move at most one position in any step. To start the fingers off we have the initialization rule

$$\mathbf{fing}(x_1, y_1) :- start(x_1, y_1)$$

and we include the basic fact $start(c_0, c_{n+1})$ in the set B of basic facts.

The query will detect errors in two stages. First, we need to eliminate the possibility that the successor of some configuration consisting of n tape positions has more than $n+1$ tape positions. This is done by means of the query φ_1 consisting of the disjunction of the queries

$$\begin{aligned}
\exists x_1 x_2 y_1 y_2 y_3 \{ & q_a(x_1) \wedge q_\#(x_2) \wedge \\
& q_b(y_1) \wedge q_c(y_2) \wedge q_d(y_3) \wedge next(x_1, x_2) \wedge \\
& next(y_1, y_2) \wedge next(y_2, y_3) \wedge \mathbf{fing}(x_1, y_1) \}
\end{aligned}$$

for all symbols $a, b, c, d \in S$. Next, we check the corresponding consecutive triples in successive configurations by means of the query φ_2 consisting of the disjunction of the queries

$$\begin{aligned}
\exists x_1 x_2 x_3 y_1 y_2 y_3 \{ & q_a(x_1) \wedge q_b(x_2) \wedge q_c(x_3) \wedge \\
& q_d(y_1) \wedge q_e(y_2) \wedge q_f(y_3) \wedge next(x_1, x_2) \wedge \\
& next(x_2, x_3) \wedge next(y_1, y_2) \wedge next(y_2, y_3) \wedge \mathbf{fing}(x_2, y_2) \}
\end{aligned}$$

for all tuples $\langle abcdef \rangle \in Error$. These queries will also ensure that no configuration consisting of n tape positions has a successor consisting of fewer than n tape positions. Now define the query φ to be the disjunction $\varphi_1 \vee \varphi_2$.

Note that $\mathbf{cellseq}(c_{n+1})$ must generate at least two successor cells c_{n+2}, c_{n+3} to c_{n+1}. Using the first finger rule, we obtain that $\mathbf{fing}(c_1, c_{n+2})$ must hold. The query φ_2 now verifies that the symbol in cell c_{n+2} is that computed from the triple consisting of the contents of the cells c_0, c_1, c_2. If the symbol selected for this cell is not the correct symbol of the next configuration of the machine M on the input, then one of these disjuncts will be satisfied with $x_1 = c_0$, $x_2 = c_1$, $x_3 = c_2$ and $y_1 = c_{n+1}$, $y_2 = c_{n+2}$, $y_3 = c_{n+3}$. The query φ_2 simultaneously verifies that the symbol in c_{n+3} is not # if that in c_2 is not #. The fingers may now be moved one position to the right, i.e., the query $\mathbf{fing}(c_2, c_{n+3})$ is satisfied, so we now verify that the symbol generated for c_{n+3} is correct. This observation is readily generalized to an inductive proof that each step of expansion of **cellseq** must select the next symbol of the computation of M, or else the query will be satisfied. The query φ_1 is used to verify that the ends of configurations come at the right places. Eventually we have either detected that the expansion does not correspond to a computation of M, or the left finger points to a cell containing # and the right finger points to the final cell in the sequence, which must also contain #. In that case, we have just verified that the next to last cell correctly indicates a halting state, so the computation halts, and the query is not satisfied in the expansion. □

Note that since the complement of the query problem is recursively enumerable (to show that the query fails it suffices to find a single expansion of the database in which it fails), this result shows that there can be no recursively enumerable proof theory for answering intensional queries containing binary defined predicates.

3.4 Adorned Glue Types

This section is devoted to setting up the formal machinery we will use to establish the decidability of monadic queries. We define an equivalence relation on expansions, and describe *adorned glue types,* the equivalence classes of this relation. The next section will use the tools developed here to obtain upper bounds on the complexity of monadic

queries.

Lemma 3.3.7 may be interpreted as stating that the defined atoms $\mathbf{R}(a_1, \ldots, a_n) \in D$ of a database express the (possibly) infinite disjunction

$$H_1(a_1, \ldots, a_n) \vee H_2(a_1, \ldots, a_n) \vee \ldots$$

where the H_i are all the expansions of $\mathbf{R}(x_1, \ldots, x_n)$ by the program Σ. For example, the defined atom $\mathbf{R}(x, y)$ of Example 3.2.1 expresses the disjunction

$$\begin{aligned} & edge(x, y) \\ \vee & \exists z_1[edge(x, z_1) \wedge edge(z_1, y)] \\ \vee & \exists z_1 z_2[edge(x, z_1) \wedge edge(z_1, z_2) \wedge edge(z_2, y)] \\ & \vdots \end{aligned}$$

The idea underlying the decidability of monadic queries is that instead of this infinite set of possibilities, it suffices to consider only a finite subset when determining entailment of a query. This will be shown by introducing an equivalence relation on the expansions, such that two expansions are equivalent if they behave identically with respect to satisfaction of the query. We will show that this equivalence relation has finite degree. Our decision procedures for monadic queries will work with representatives of the equivalence classes instead of the infinite set of expansions.

The expansions of a defined atom are hypergraphs in which all edges are labelled by basic predicates. When dealing with monadic queries it is convenient to work with a slightly larger class of hypergraphs, which contain edges for the monadic defined predicates as well. Suppose that $\mathcal{P}$ is a set of monadic defined predicates. A $\mathcal{P}$-adorned hypergraph is a hypergraph G on the base predicates and $\mathcal{P}$. If v is a vertex of an adorned hypergraph G then the set of predicates $\mathbf{A} \in \mathcal{P}$ such that the hypergraph G contains an edge $\mathbf{A}(v)$ will be called the *adornment* of v. Under the correspondence between hypergraphs and structures, a $\mathcal{P}$-adorned hypergraph G corresponds to a structure for the basic predicates and the predicates $\mathcal{P}$. Thus, we may speak of satisfaction of a formula φ containing basic and monadic defined predicates in G, and write $G \models \varphi$. Note that we do not invoke any definitions Π of the predicates $\mathcal{P}$ when determining $G \models \varphi$ for a $\mathcal{P}$-adorned hypergraph G: we are simply testing satisfaction of a first-order

formula in a first-order structure. To distinguish this relation from satisfaction of the monadic query (Π, φ), in which we do invoke the definitions, we will refer to it as *flat* satisfaction.

If Π is a program defining the predicates $\mathcal{P}$ and $\Pi^\infty(G) = G$ then we say that G is *legal* for Π. Equivalently, G is legal for Π if, as a first order structure, it satisfies the rules Π. Now define the relation $\sim_\varphi^\Pi$ on $\mathcal{P}$-adorned k-graphs by $G_1 \sim_\varphi^\Pi G_2$ if and only if both of the following hold:

1. For all $\mathcal{P}$-adorned k-graphs H, $G_1 \circ H$ is legal for Π if and only if $G_2 \circ H$ is legal for Π.

2. For all $\mathcal{P}$-adorned k-graphs H, $G_1 \circ H \models \varphi$ if and only if $G_2 \circ H \models \varphi$.

The second condition states that, with respect to flat satisfaction of the query φ, gluing on the adorned k-graph G_1 is equivalent to gluing on the adorned k-graph G_2. The first condition may be interpreted to state that the operations of gluing on the two adorned hypergraphs are equivalent with respect to satisfaction of the rules Π. (If the query φ is basic, it is possible to drop the first condition, and work with the relation defined by the second condition only.) Note that if G is a k-graph containing defined atoms and θ is a substitution such that for each defined edge e of G, $\theta(e)$ is a $\mathcal{P}$-adorned hypergraph, then $G\theta$ is also a $\mathcal{P}$-adorned hypergraph. The proof of the following is straightforward.

Lemma 3.4.1: 1. The relation $\sim_\varphi^\Pi$ is an equivalence relation.

2. Suppose θ and θ' are substitutions for the defined atoms of a hypergraph G. If $\theta(e) \sim_\varphi^\Pi \theta'(e)$ for each defined atom e then

   (a) $G\theta \sim_\varphi^\Pi G\theta'$,

   (b) $G\theta \models \varphi$ if and only if $G\theta' \models \varphi$, and

   (c) $G\theta$ is legal if and only if $G\theta'$ is legal.

The equivalence class of an adorned hypergraph G will be called the *adorned glue type* of G. A set S of adorned k-graphs is a *complete set of adorned representatives* if for each adorned k-graph G there exists a hypergraph $H \in S$ such that $H \sim_\varphi^\Pi G$.

The following shows that to determine the equivalence of two adorned hypergraphs it is sufficient to check the conditions of the definition on a complete set of representatives.

Lemma 3.4.2: Suppose that S is a complete set of adorned representatives of adorned k-graphs. If G_1, G_2 are adorned k-graphs then $G_1 \sim_\varphi^\Pi G_2$ exactly when both

1. $G_1 \circ H$ is legal if and only if $G_2 \circ H$ is legal, and
2. $G_1 \circ H \models \varphi$ if and only if $G_2 \circ H \models \varphi$

hold for all $H \in S$.

Proof: The implication from left to right is immediate. For the converse, assume that conditions 1 and 2 hold for all $H \in S$. Suppose that H' is an adorned k-graph, and let $H \in S$ satisfy $H' \sim_\varphi^\Pi H$. Then $G_1 \circ H'$ is legal if and only if $G_1 \circ H$ is legal, which holds if and only if $G_2 \circ H$ is legal. But this holds just when $G_2 \circ H'$ is legal. Similarly, $G_1 \circ H' \models \varphi$ if and only if $G_2 \circ H' \models \varphi$. □

We now set out to show that for each monadic query (Π, φ) and for each k the relation $\sim_\varphi^\Pi$ on the set of adorned k-graphs has a finite number of equivalence classes. We need the following definitions. If $G = \langle V, E, lab, vert, src \rangle$ is a hypergraph then a *subgraph* will be any hypergraph of the form $G' = \langle V', E', lab, vert, src \rangle$ where $\{src[i] \mid 1 \leq i \leq k\} \subseteq V' \subseteq V$ and $E' \subseteq E$. A subgraph G' of an adorned hypergraph G will be called an *adorned subgraph* if every vertex v of G' has the same adornment it had in G.

A notion of homomorphism of hypergraphs may be defined in a fashion analogous to the definition of homomorphism of interpretations. That is, a homomorphism from G_1 to G_2 is a mapping of vertices of G_1 to vertices of G_2 and edges of G_1 to edges of G_2 which preserves labels, the vertices associated to each edge and the sequence of sources. We write $G_1 \leq G_2$ when such a mapping exists. Two hypergraphs G_1, G_2 are *isomorphic* if there exists a homomorphism from G_1 to G_2 which is bijective on both the vertex and the edge sets. A homomorphism $h : G_1 \leq G_2$ is *adornment preserving* if for each vertex v of G_1, the adornment of v in G_1 is exactly the same as the adornment of $h(v)$ in G_2. The proof of the following lemma is straightforward.

Lemma 3.4.3: 1. If G_1, G_2 and H are k-graphs with (adornment preserving) homomorphisms $h_1 : G_1 \leq H$ and $h_2 : G_2 \leq H$ then there exists an (adornment preserving) homomorphism $h : G_1 \circ G_2 \leq H$.

2. If G' is an adorned subgraph of the adorned hypergraph G then there exists an adornment preserving homomorphism from G' to G.

3. If there exists an adornment preserving homomorphism from G_1 to G_2, then for any adorned hypergraph H there exists an adornment preserving homomorphism from $G_1 \circ H$ to $G_2 \circ H$.

A k-graph P will be called a *source identifier* if all its vertices are sources and it has an empty set of edges. Thus, if P is a *source identifier* and G is any k-graph, the hypergraph $G \circ P$ is just the hypergraph G with some of its sources identified. A *partition* of a conjunctive query δ is a decomposition of the query as

$$\delta = \exists \mathbf{xyz}[\delta_1(\mathbf{x}, \mathbf{y}) \wedge \delta_2(\mathbf{y}, \mathbf{z})] \tag{3.3}$$

where only the variables $\mathbf{x}, \mathbf{y}$ occur in the subquery δ_1 and only the variables $\mathbf{y}, \mathbf{z}$ occur in the subquery δ_2.

Suppose that G' is an adorned subgraph of the adorned hypergraph G and let ψ be a query. We will say that G' is an *adorned ψ-contribution* of G if there exists a source identifier P, a partition $\delta = \exists \mathbf{xyz}[\delta_1(\mathbf{x}, \mathbf{y}) \wedge \delta_2(\mathbf{y}, \mathbf{z})]$ of some disjunct δ of the disjunctive normal form of ψ and an assignment α mapping the variables $\mathbf{y}$ to sources of $G \circ P$ such that $G' \circ P \models \exists \mathbf{x}[\delta_1(\mathbf{x}, \alpha(\mathbf{y}))]$, but for every proper adorned subgraph G'' of G' we have $G'' \circ P \not\models \exists \mathbf{x}[\delta_1(\mathbf{x}, \alpha(\mathbf{y}))]$. Intuitively, an adorned ψ-contribution of G is a portion of G that may result in the satisfaction of ψ after G has been glued to some graph.

Example 3.4.4: Let ψ be the query $\exists xy[R(x, x, y) \wedge \mathbf{A}(y)]$, where R is a basic predicate and Π is a program defining the monadic predicates $\mathbf{A}$ and $\mathbf{B}$. Suppose that G is the 3-graph with sources s_1, s_2, s_3, internal vertices u, v, edges $R(s_1, s_2, s_3), R(u, v, s_3)$ and adornment $\mathbf{A}(u), \mathbf{B}(u), \mathbf{B}(s_1)$. Then the subgraph G' with vertices s_1, s_2, s_3, edges $R(s_1, s_2, s_3)$ and adornment $\mathbf{B}(s_1)$ is an adorned ψ-contribution. For, suppose P identifies s_1 and s_2. If $\alpha(y) = s_3$ then then

$G' \circ P \models \exists x[R(x, x, \alpha(y))]$. No proper adorned subgraph of G' has this property. Note that the subgraph of G' obtained by removing the adornment $\mathbf{B}(s_1)$ is not a proper adorned subgraph of G because the vertex s_1 does not retain the adornment it had in G. Thus, this hypergraph is not a ψ-contribution. The subgraph G'' with vertices u, v, s_1, s_2, s_3, edges $R(u, v, s_3)$ and adornment $\mathbf{A}(u), \mathbf{B}(u), \mathbf{B}(s_1)$ is an adorned subgraph. However, it is not a ψ-contribution: the vertices u, v are internal vertices, so they cannot be equated by any source-identifier. □

The following is immediate from the fact that a ψ-contribution is an adorned subgraph:

Lemma 3.4.5: If G' is an adorned ψ-contribution of G then there exists an adornment preserving homomorphism $h : G' \leq G$.

We now show that the number of glue types of adorned k-graphs is finite for each k. For each adorned k-graph G we construct an adorned k-graph $rep(G)$ such that $rep(G) \sim_{\varphi}^{\Pi} G$. Given a query ψ, consider the set of ψ-contributions of G. Some of these may be isomorphic. Let S be a set containing one representative of all the isomorphism classes of ψ-contributions. Now let $R(\psi, G)$ be the result of gluing together the adorned k-graphs in S. That is, $R(\psi, G)$ is the k-graph obtained by first renaming the vertices of the hypergraphs in S so that no two share a vertex, and then identifying the corresponding sources of all these hypergraphs. Note that for every adorned ψ-contribution G' of G there exists a hypergraph $G'' \in S$ such that G'' is isomorphic to G', so it follows that $G' \leq G$. We now define $rep(G)$ to be the result of gluing together all the hypergraphs $R(\psi, G)$, where the query ψ is either φ or the body $B(x)$ of some rule $\mathbf{A}(x) :- B(x)$ of Π.

Proposition 3.4.6: Let (Π, φ) be a monadic query. For each k the set of adorned glue types of the relation $\sim_{\varphi}^{\Pi}$ is finite, and there exists a set of adorned representatives, each element of which can be represented in space $k^{|\varphi| \cdot |\Pi|^2}$.

Proof: We show that $rep(G) \sim_{\varphi}^{\Pi} G$ for each adorned k-graph G, and compute a bound on the size of $rep(G)$. Since there can only be a finite number of adorned k-graphs of any size, it will follow that the number of adorned glue types is finite.

Observe that there exists an adornment preserving homomorphism $h : rep(G) \leq G$. This follows from the fact that for each ψ-contribution G' we have $G' \leq G$, using Lemma 3.4.3(1).

First, we show that for all adorned k-graphs H, $G \circ H \models \varphi$ if and only if $rep(G) \circ H \models \varphi$. That $rep(G) \circ H \models \varphi$ implies $G \circ H \models \varphi$ is immediate from the fact that $rep(G) \leq G$ using Lemma 3.2.3. For the converse, suppose that $G \circ H \models \varphi$. Let G' be a minimal adorned subgraph of G such that $G' \circ H \models \varphi$. Let P be the source identifier that identifies two sources just when they are identified in $G \circ H$. Then there exists a partition $\delta = \exists \mathbf{xyz}[\delta_1(\mathbf{x}, \mathbf{y}) \wedge \delta_2(\mathbf{y}, \mathbf{z})]$ of some disjunct δ of the disjunctive normal form of φ and an assignment α mapping the variables $\mathbf{y}$ to the sources of $G \circ H$ such that $G' \circ P \models \exists \mathbf{x}[\delta_1(\mathbf{x}, \alpha(\mathbf{y}))]$ and $P \circ H \models \exists \mathbf{z}[\delta_2(\alpha(\mathbf{y}), \mathbf{z})]$. Now let G'' be a proper adorned subgraph of G'. Then we have $G'' \circ P \not\models \exists \mathbf{x}[\delta_1(\mathbf{x}, \alpha(\mathbf{y}))]$. For otherwise we would have $G'' \circ P \models \exists \mathbf{x}[\delta_1(\mathbf{x}, \alpha(\mathbf{y}))]$ and $P \circ H \models \exists \mathbf{z}[\delta_2(\alpha(\mathbf{y}), \mathbf{z})]$, which implies that $G'' \circ H \models \varphi$, contrary to the minimality of G'. Thus G' is an adorned φ-contribution of G. This implies that $G' \leq rep(G)$, so $rep(G) \circ H \models \varphi$ because $G' \circ H \models \varphi$. This proves that $G \circ H \models \varphi$ if and only if $rep(G) \circ H \models \varphi$. (In fact, we have shown slightly more than this, namely, that $G \circ H \models \varphi$ if and only if there exists an adorned ψ-contribution G' such that $G' \circ H \models \varphi$.)

Next we show that $G \circ H$ is legal if and only if $rep(G) \circ H$ is legal. Suppose that $G \circ H$ is legal. Because there is an adornment preserving homomorphism from $rep(G)$ to G, there exists an adornment preserving homomorphism h from $rep(G) \circ H$ to $G \circ H$, by Lemma 3.4.3(1). Let $\mathbf{A}(x) : -B(x)$ be a rule of Π and suppose that $rep(G) \circ H \models B(v)$ for some vertex v. Since h is a homomorphism, we have $G \circ H \models B(h(v))$. The legality of $G \circ H$ now implies that $h(v)$ is adorned by $\mathbf{A}$. Since the homomorphism is adornment preserving, we also have that v is adorned by $\mathbf{A}$. This shows that if $G \circ H$ is legal then $rep(G) \circ H$ is legal.

Conversely, assume that $rep(G) \circ H$ is legal. Let $\mathbf{A}(x) : -B(x)$ be a rule of Π, and suppose that $G \circ H \models B(v)$ for some vertex v. We consider two cases. First, if v arises from some node of H, then by an argument similar to that above we may show $rep(G) \circ H \models B(v)$. Since $rep(G) \circ H$ is legal this means that v is adorned by $\mathbf{A}$ in

$rep(G) \circ H$, and consequently also in $G \circ H$.

Second, if v arises from some internal node of G, let G' be a minimal adorned subgraph of G containing v such that $G' \circ H \models B(v)$. Then it may be argued as above that G' is an adorned $B(x)$-contribution. Thus, there exists an adornment preserving homomorphism h' from G' to $rep(G)$. This implies that $rep(G) \circ H \models B(h'(v))$, so $h'(v)$ is adorned by $\mathbf{A}$ in $rep(G)$. This means that v is adorned by $\mathbf{A}$ in G', and consequently in G also. This proves that $G \circ H$ is legal. Thus, we have shown that if $rep(G) \circ H$ is legal then $G \circ H$ is legal.

It remains only to determine the size of the hypergraph $rep(G)$. To do this, note first that each adorned φ-contribution G' of G is isomorphic to a hypergraph G'' of size $|\varphi|(|\Pi| + log\, k)$. This is because, by minimality of G', each edge e of G' corresponds to an atom of φ. The vertices of this edge are either one of the k sources or an internal vertex, so we may require an additional $log\ k$ bits to describe each vertex associated with an edge. Finally, each vertex is adorned by some of the monadic predicates defined by the program Π, accounting for the contribution $|\Pi|$. Similarly, each adorned $B(x)$-contribution of G is of size $|\Pi|(|\Pi| + log\, k)$. Since $R(\varphi, G)$ is the result of gluing together some set of non-isomorphic φ-contributions, it can be represented in space $2^{|\varphi|(|\Pi|+log\ k)}$. Similarly, $R(B(x), G)$ can be represented in space $2^{|\Pi|(|\Pi|+log\ k)}$. Thus $rep(G)$ can be represented in space $2^{|\varphi|(|\Pi|+log\ k)} + |\Pi| \cdot 2^{|\Pi|(|\Pi|+log\ k)} \leq k^{|\varphi| \cdot |\Pi|^2}$. $\square$

The following is an immediate consequence of Lemma 3.4.2 and Proposition 3.4.6

Corollary 3.4.7: It is possible to determine if $G \sim_{\varphi}^{\Pi} H$ for k-graphs G, H by a deterministic computation using additional space $O\left(k^{|\varphi| \cdot |\Pi|^2}\right)$.

3.5 Complexity: Upper Bounds

We now characterize the exact complexity of various decision problems related to answering monadic queries. We will work primarily with the standard semantics. However, all our results apply also to the unique names semantics, and we indicate at the end of this section why this is the case.

We use the following measures of the complexity of the querying recursively indefinite databases. Following Vardi [123] we define the *answer set* of a query (Π, φ) with respect to a class C of programs to be the set

$$AS_C(\Pi, \varphi) = \{\Sigma, [B, D] \mid \Sigma, [B, D] \models (\Pi, \varphi) \text{ and } \Sigma \in C\}.$$

The *data complexity* of a query (Π, φ) with respect to a class C of programs is the complexity of the set $AS_C(\Pi, \varphi)$. That is, the data complexity describes the complexity of answering a fixed query as the size of the database grows. Data complexity is intended to capture the intuition that queries are generally short, whereas databases may grow to very large sizes, so that the dominant contribution to the complexity of query answering will come from the database size.

One slightly unusual feature of our definition of data complexity is that we vary the program Σ. This is because in recursively indefinite databases the program Σ is considered part of the data, since it provides the definitions of the defined facts D. By varying the class of programs C we measure the complexity of querying defined facts stated in terms of various types of definitions. However, by taking the class C to be a singleton set $\{\Sigma\}$ we may measure the complexity of querying data expressed in terms of a fixed set of defined predicates, i.e., we consider only databases containing an *a priori* determined set of types of recursively indefinite information. We write $AS_\Sigma(\Pi, \varphi)$ for the answer set in this case.

Besides the singleton sets we consider the following classes C of programs. The class $Prog$ contains all Datalog programs. The class $Arity(k)$ contains all programs whose defined predicates have arity no greater than k and whose rules contain only a fixed set of k constants. The class $Linear$ consists of all linear programs. We will also say that a recursively indefinite database $\mathbf{D}$ is linear if its program is linear. For basic queries φ we will write simply $AS_C(\varphi)$ for the answer set.

For completeness we also consider the contribution to complexity due to the size of the query. The answer set of a *database* $\mathbf{D} : \Sigma, [B, D]$ with respect to a class Q of queries is the set

$$AS_Q(\mathbf{D}) = \{\psi \mid \mathbf{D} \models \psi \text{ and } \psi \in Q\}$$

of queries entailed by the database. The *expression complexity* of a database $\mathbf{D}$ is the complexity of the set $AS_Q(\mathbf{D})$. This is a measure of the complexity of query answering as a function of the size of the query. We will consider the classes $Basic$ of basic queries and $Monadic$ of monadic queries.

Finally, *combined complexity* is a measure in which both the query and the database are allowed to vary. If C is a class of programs and Q is a class of queries then we define the set

$$AS_{C,Q} = \{(\Sigma, [B, D], \psi) \mid \Sigma, [B, D] \models \psi \text{ and } \Sigma \in C \text{ and } \psi \in Q\}.$$

The combined complexity with respect to C and Q is the complexity of this set.

Let Σ be a Datalog program. We assume that Σ contains no constants: see the discussion in Section 3.3 on how this assumption may be eliminated. Let $\mathbf{R}$ be a defined predicate of arity k. Recall that $L(\Sigma, \mathbf{R})$ is the set of k-graphs generated by the productions $P(\Sigma)$ from the axiom $\{x_1, \ldots x_k : \mathbf{R}(x_1, \ldots, x_k)\}$. An adorned representative G will be called an *adorned representative of* $L(\Sigma, \mathbf{R})$ if there exists a hypergraph $H \in L(\Sigma, \mathbf{R})$ with an adornment H^a such that $G \sim_{\varphi}^{\Pi} H^a$. We write $Types(\Sigma, \mathbf{R})$ for a complete set of adorned representatives of $L(\Sigma, \mathbf{R})$.

Example 3.5.1: Consider the program of Example 3.1.3. The expansions depicted in Figure 3.2 are a complete set of representatives of the expansions of **rgpath** with respect to $\varphi = \exists xy[red(x) \wedge edge(x, y) \wedge green(y)]$. This can be shown as follows. The expansions of $\{x_1x_2 : \mathbf{rgpath}(x_1, x_2)\}$ consist of a sequence of edges from the first source x_1 (indicated by 1 in the diagram), to the second source x_2 (indicated by 2), such that each node is "labelled" either *red* (r) or *green* (g). The expansions consisting of only two nodes are depicted on the top row. Consider the expansions which have three or more nodes. If there exists an edge from a red node to a green node, then the expansion satisfies the query on its own, so is equivalent to the second expansion on the top row. If all nodes are green then the expansion is equivalent to the rightmost expansion on the first row. This leaves the expansions consisting of a series of green nodes followed by a series of red nodes. We may represent these expansions as a sequence of colours

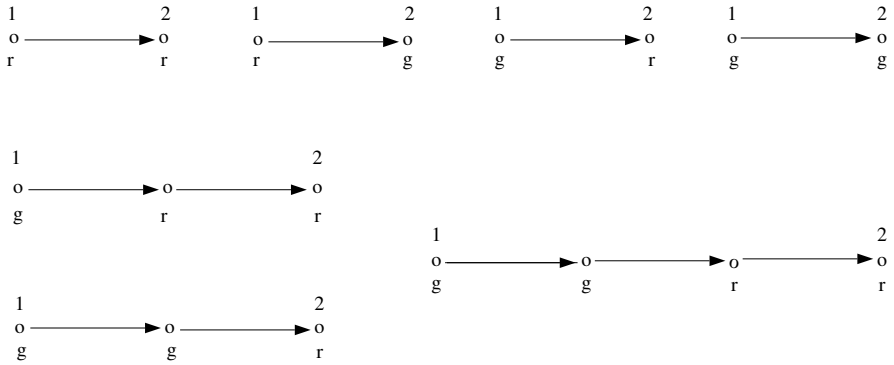

Figure 3.2: A complete set of expansions

$G = \langle g, u_1, \ldots, u_n, r \rangle$, where possibly $n = 1$. Since we assume that G does not satisfy φ on its own, it is apparent that the query φ can never be satisfied using nodes from $\{u_2, \ldots, u_{n-1}\}$ for the variables x, y. Hence the glue type of G is determined by the colours of u_1 and u_n. We cannot have u_1 red and u_n green, else the query is satisfied. This leaves three cases, represented by the bottom three expansions in the diagram. We leave it for the reader to verify that all the expansions shown are of distinct glue types, i.e., that for any distinct pair G_1,G_2 there exists a 2-graph H such that exactly one of $G_1 \circ H$, $G_2 \circ H$ satisfies φ. □

The following result, together with the fact that monadic queries generate a finite set of glue types, establishes that monadic queries are decidable, and forms the basis for all our decision procedures:

Lemma 3.5.2: $\Sigma, [B, D] \not\models (\Pi, \varphi)$ if and only if there exists an adornment B^a of B and there exists for each atom $e = \mathbf{R}(a_1, \ldots, a_n) \in D$ an adorned representative G_e of $L(\Sigma, \mathbf{R})$ such that $M = [B^a, D]\theta$ is legal and $M \not\models \varphi$, where θ is the substitution that replaces each atom $e \in D$ by G_e.

Proof: First we show the implication from left to right. Suppose that $\Pi \cup \Sigma, [B, D] \not\models \varphi$. Then by Lemma 3.3.6 there exists an expansion H_e for each atom $e \in D$ such that $M' \not\models \varphi$, where $M' = \Pi^\infty([B, D]\theta')$. Here θ' is the substitution that replaces each $e \in D$ by H_e. For each predicate $\mathbf{A} \in \mathcal{P}$, adorn H_e by adding $\mathbf{A}(c)$ for each node c of H_e such that $M' \models \mathbf{A}(c)$. Call the resulting adorned hypergraph H_e^a, and let θ^a be

the substitution that replaces each atom $e \in D$ by H_e^a. Let B^a be the result of similarly adorning B. Then $[B^a, D]\theta^a = M'$ is legal and $M' \not\models \varphi$. Now let $G_e \sim_\varphi^\Pi H_e^a$ be representatives of the adorned hypergraphs H_e^a and let θ be the substitution mapping each defined atom e in D to the adorned hypergraph G_e. Define M to be the structure $[B^a, D]\theta$. By Lemma 3.4.1 we have that M is legal and $M \not\models \varphi$.

Conversely, suppose $M = [B^a, D]\theta$ is legal and $M \not\models \varphi$, where the substitution θ replaces each atom $e \in D$ by the adorned representative G_e of $L(\Sigma, \mathbf{R})$. We show that $\Pi \cup \Sigma, [B, D] \not\models \varphi$. For each e, let $H_e \in L(\Sigma, \mathbf{R})$ be a hypergraph which has an adornment $H_e^a \sim_\varphi^\Pi G_e$. Let the substitution θ' replace each atom $e \in D$ by the hypergraph H_e and let the substitution θ^a replace each atom $e \in D$ by the adorned hypergraph H_e^a. Define the structure M^a by $M^a = [B^a, D]\theta^a$. Because $\theta(e) \sim_\varphi^\Pi \theta^a(e)$ for each defined atom e, it follows using Lemma 3.4.1 that M^a is legal and $M^a \not\models \varphi$. Now let $M' = [B, D]\theta'$. Because, by construction, we have $M' \leq M^a$, it follows that $\Pi^\infty(M') \leq \Pi^\infty(M^a)$. But the legality of M^a means that $\Pi^\infty(M^a) = M^a$. Consequently, $\Pi^\infty(M') \not\models \varphi$ by Lemma 3.2.3. Lemma 3.3.5 now yields that $\Sigma, [B, D] \not\models (\Pi, \varphi)$. □

Lemma 3.5.2 immediately suggests the following non-deterministic procedure to determine the complement of query satisfaction, i.e., to determine if $\Sigma, [B, D] \not\models (\Pi, \varphi)$.

$FALSE(\Sigma, [B, D], \Pi, \varphi)$:

1. Guess an adornment B^a of B.
2. For each atom $e = \mathbf{R}_e(a_1, \ldots, a_k) \in D$ guess an adorned representative G_e. Let θ be the substitution that replaces each e by G_e. Put $M = [B^a, D]\theta$.
3. Check if M is legal and $M \models (\Pi, \varphi)$: if false, reject.
4. Verify that $G_e \in \mathit{Types}(\Sigma, \mathbf{R}_e)$ for each e: if true, accept, else reject.

Suppose now that we fix the program Σ and the monadic query (Π, φ). Then the sets $\mathit{Types}(\Sigma, \mathbf{R})$ are also fixed and we can restrict the procedure $FALSE$ to guess adorned representatives from a fixed set. (We will see shortly how the sets $\mathit{Types}(\Sigma, \mathbf{R})$ may be computed.) Thus, the test of line 4 of the procedure $FALSE$ may be performed in

constant time by a simple table look-up. This establishes the following upper bound on the data complexity of databases with fixed program.

Theorem 3.5.3: For every program Σ and monadic query (Π, φ) the set $AS_\Sigma(\Pi, \varphi)$ is in co-NP.

The following fixpoint computation suffices to associate to each defined predicate its set of representatives. We inductively construct a sequence of functions T_i mapping the defined predicates of Σ to sets of adorned representatives. If $\mathbf{R}$ is a k-ary defined predicate of Σ then $T_i(\Sigma, \mathbf{R})$ will be a set of k-graphs. For the basis we put $T_0(\Sigma, \mathbf{R}) = \emptyset$ for all defined predicates $\mathbf{R}$. The function T_{i+1} is obtained from T_i as follows. Let r be a rule of Σ of the form $\mathbf{R}(x_1, \ldots x_k) : -\exists y_1 \ldots y_n B$. (The variables $x_1, \ldots, x_k$ need not be distinct.) We think of the body B as a k-graph with vertices $\{x_1, \ldots, x_k, y_1, \ldots y_n\}$. The sources of B are $x_1, \ldots, x_k$, and there is an edge for each basic or defined atom. Choose an adornment B^a of this hypergraph. Let the substitution θ map each defined edge $e = \mathbf{R}_e(u_1 \ldots u_k)$ to an element of $T_i(\Sigma, \mathbf{R}_e)$. Now let G be the adorned representative with $G \sim_\varphi^\Pi B^a\theta$. We define $T_{i+1}(\Sigma, \mathbf{R})$ be the set of all adorned representatives G obtained in this fashion. Since for each k the set of adorned glue types of k-graphs is finite, for some number N we will have $T_N = T_{N+1}$. A straightforward induction then shows that for all defined predicates $\mathbf{R}$ we have $T_N(\Sigma, \mathbf{R}) = Types(\Sigma, \mathbf{R})$.

Suppose now that the query (Π, φ) is fixed and that the programs Σ are constrained to have arity bounded by k. Then by Proposition 3.4.6 there exists a finite set S of size

$$M = O(2^{k^{|\varphi|\,|\Pi|^2}})$$

of adorned glue types such that for every defined predicate $\mathbf{R}$ we have $T_i(\Sigma, \mathbf{R}) \subseteq S$. It follows that the fixpoint computation converges in $N \leq |\Sigma| \cdot M$ steps. However, it is not necessarily the case that we can compute the mapping T_{i+1} from the mapping T_i in time polynomial in the size of the program Σ. This is because the program may contain a rule of length $c \cdot |\Sigma|$ for some constant c. In order to determine all the adorned hypergraphs $G \in T_{i+1}(\Sigma, \mathbf{R})$ generated by this rule, we need to search through a set of order $2^{c \cdot |\Sigma|}$ substitutions and adornments, even though these substitutions generate

at most M new representatives. However, since we know that the T_i converge in a polynomial number of steps, what we can do is verify a *proof* that $G \in T_N(\Sigma, \mathbf{R})$ in polynomial time.

A *derivation* Δ will be a sequence of steps, each of which is a tuple consisting of a rule r of Σ, an adornment B^a of the body B of r, a substitution θ for the defined atoms of the body and a *conclusion* of the form $(G, \mathbf{R})$ where G is an adorned representative and $\mathbf{R}$ is the predicate in the head of r. We will say that a derivation is *valid* if for each step we have $G \sim_{\varphi}^{\Pi} B^a\theta$ and for every defined atom $e = \mathbf{R}_e(u_1, \ldots, u_k)$ of B there exists an earlier step with conclusion $(\theta(e), \mathbf{R}_e)$. Finally, a derivation *proves* $G \in T_N(\Sigma, \mathbf{R})$ if it is valid and contains a step with conclusion $(G, \mathbf{R})$. It is straightforward to verify that $G \in Types(\Sigma, \mathbf{R})$ if and only if there exists a derivation of length $|\Sigma| \cdot M$ which proves $G \in T_N(\Sigma, \mathbf{R})$. This yields the following nondeterministic algorithm for performing the verification of $G \in Types(\Sigma, \mathbf{R})$ required in line 4 of the algorithm $FALSE$:

1. Guess a derivation Δ of length $|\Sigma| \cdot M$.

2. Verify that Δ proves $G \in T_N(\Sigma, \mathbf{R})$: if true, accept, else reject.

This establishes the following upper bound for data complexity on databases with arity k programs.

Theorem 3.5.4: The set $AS_{Arity(k)}(\Pi, \varphi)$ is in co-NP for every monadic query (Π, φ) and number k, .

We will now analyze the data complexity in the case of programs of unbounded arity, and simultaneously analyze the expression complexity and combined complexity for linear and non-linear databases. It turns out a uniform algorithm provides upper bounds for all of these problems. The algorithm makes use of alternating Turing machines. We refer the reader to [13] for details on this generalization of nondeterministic computation. We reuse the procedure $FALSE$. In order to perform the test of $G \in Types(\Sigma, \mathbf{R})$ at line 4 of this procedure, we use the recursive alternating procedure shown in Figure 3.3. The following lemma establishes correctness of this routine.

Lemma 3.5.5: The procedure call $ELT(G, \mathbf{R})$ accepts if and only if G is an adorned representative of $L(\Sigma, \mathbf{R})$.

$ELT(G, \mathbf{R})$:

1. Existentially guess a rule $\mathbf{R}(x_1, \ldots, x_n) : -B$ of Σ and an adornment B^a of the body B.
2. Existentially guess an adorned representative G_e for each defined atom e of B. Let θ be the substitution mapping each e to G_e.
3. Check that $G \sim_{\varphi}^{\Pi} B^a\theta$: if false, reject
4. If B has no defined atoms, accept.
5. Universally call $ELT(G_e, \mathbf{R}_e)$ for each defined atom e, where $\mathbf{R}_e$ is the predicate of e.

Figure 3.3: A routine to test $G \in L(\Sigma, \mathbf{R})$.

Proof: First we show that if $ELT(G, \mathbf{R})$ accepts then G is an adorned representative of $L(\Sigma, \mathbf{R})$. We may view an accepting computation as a tree T in which the nodes correspond to calls of the procedure ELT. Each such node n has associated with it an adorned hypergraph $G(n)$ and a predicate $\mathbf{R}(n)$ when it corresponds to the call $ELT(G(n), \mathbf{R}(n))$. In addition, for each node n we have a rule with predicate $\mathbf{R}(n)$ in the head, and an adornment $B^a(n)$ of the body $B(n)$. The successors of the node n correspond to the calls $ELT(G_e, \mathbf{R}_e)$ for defined atoms e in the body $B(n)$. All of these calls accept. If r is the root of the tree then we have $G(r) = G$ and $\mathbf{R}(r) = \mathbf{R}$.

We associate an adorned hypergraph $H^a(n)$ with each node n of this tree as follows. If the node n is a leaf, then $H^a(n) = B^a(n)$. Otherwise, for each defined atom e of $B(n)$ let n_e be the corresponding successor of n. Let θ^a be the substitution mapping the edge e to the adorned hypergraph $H^a(n_e)$. We let $H^a(n)$ be the adorned hypergraph $B^a\theta^a$. It is clear that that the hypergraph $H^a(r)$ is an adorned version of a hypergraph in $L(\Sigma, \mathbf{R})$. We show by induction on T that $G \sim_{\varphi}^{\Pi} H^a(r)$.

For the base case, suppose that n is a leaf. Since the call $ELT(G(n), \mathbf{R}(n))$ accepts, the test at step 3 of the routine must succeed, i.e., $G(n) \sim_{\varphi}^{\Pi} B^a(n)$. Since $H^a(n) = B^a(n)$, we have $G(n) \sim_{\varphi}^{\Pi} H^a(n)$. For the inductive step, suppose that $G(n_e) \sim_{\varphi}^{\Pi} H^a(n_e)$ for all the successors n_e of the node n. It follows from this by Lemma 3.4.1 that

$$B^a(n)\theta \sim_{\varphi}^{\Pi} B^a(n)\theta^a$$

where θ^a substitutes $H^a(n_e)$ and θ substitutes $G(n_e)$ for each defined atom e of $B(n)$. Because the test at step 3 of the routine succeeds, the left hand side of this equivalence is equivalent to $G(n)$. The right hand side is $H^a(n)$, so we have $G(n) \sim_\varphi^\Pi H^a(n)$.

For the converse, suppose that $G \sim_\varphi^\Pi H^a$ where H^a is obtained by adornment of a hypergraph $H \in L(\Sigma, \mathbf{R})$. Consider the derivation $K_0 \longrightarrow K_1 \longrightarrow \ldots \longrightarrow K_m = H$ of H from the axiom $K_0 = \{x_1 \ldots x_k | \mathbf{R}(x_1, \ldots, x_k)\}$ using the hypergraph grammar associated to the program Σ. That is, for each $n = 1 \ldots m$ there exists an edge $e_n = \mathbf{R}(n)(a_1, \ldots, a_k)$ of K_{n-1} and a hypergraph $B(n)$ constructed from the body of a rule with predicate $\mathbf{R}(n)$ in the head, such that $K_n = K_{n-1}[B(n)/e_n]$. This derivation may be associated with a tree T as follows. For each $n = 1 \ldots m$ there is a node of this tree. The root corresponds to $n = 1$. If the defined edge e_n first occurs in the hypergraph K_j, then the node n is a successor to the node j. In other words, a node j has a successor for each of the defined edges in the hypergraph $B(j)$, which describe the hypergraphs eventually substituted for these edges by the derivation. Using the tree T we obtain a "bottom up" derivation of the hypergraph H, as follows. If n is a leaf we define $H(n) = B(n)$. Otherwise, if n is a node of T with successors n_e corresponding to the defined edges e of $B(n)$, define

$$H(n) = B(n)\theta' \tag{3.4}$$

where θ' is the substitution mapping each defined edge e to $H(n_e)$. A straightforward induction shows $H(1) = H$.

We adorn these hypergraphs as follows. Each vertex c of $B(n)$ or $H(n)$ maps homomorphically to a vertex c' of H. Let $B^a(n)$ and $H^a(n)$ be the adorned hypergraphs obtained from $B(n)$ and $H(n)$ respectively by adding the atom $\mathbf{A}(c)$ whenever $\mathbf{A}(c')$ holds in H^a. For each node n let $G(n)$ be an adorned representative with $G(n) \sim_\varphi^\Pi H^a(n)$. We claim that the tree T together with the mappings G, B^a, B and $\mathbf{R}$ describes an accepting computation of the procedure call $ELT(G, \mathbf{R})$, where each node n corresponds to the call $ELT(G(n), \mathbf{R}(n))$.

The proof is by induction on the tree T. If n is a leaf we have that $H^a(n) = B^a(n)$. This implies that the test at line 3 of the procedure succeeds, so the call $ELT(G(n), \mathbf{R}(n))$ accepts. This establishes the basis. Suppose that the subroutine

calls $ELT(G(n_e), \mathbf{R}(n_e))$ accept for each of the successors n_e of a node n. Note that the identity (3.4) implies that

$$H^a(n) = B^a(n)\theta^a \tag{3.5}$$

where the substitution θ^a maps the defined atom e to the adorned hypergraph $H^a(n_e)$. Let θ be the substitution mapping e to the adorned representative $G(n_e)$. Because $\theta^a(e) = H^a(n_e) \sim_\varphi^\Pi G(n_e) = \theta(e)$ it follows from equation (3.5) by Lemma 3.4.1 that $H^a(n) \sim_\varphi^\Pi B^a(n)\theta$. Since $G(n)$ is defined to be a representative equivalent to $H^a(n)$ it follows that the test at line 3 succeeds during the procedure call $ELT(G(n), \mathbf{R}(n))$. Since all the calls $ELT(G(n_e), \mathbf{R}(n_e))$ accept, so does the call $ELT(G(n), \mathbf{R}(n))$. □

We now consider the procedure for query answering resulting from the combination of the procedure $FALSE$ with the subroutine ELT. We analyze the complexity of this procedure under various assumptions on the type of query and database. We begin with data complexity. Suppose that the query (Π, φ) is fixed. Then by Proposition 3.4.6 we can restrict attention to representatives of size $k^{|\varphi|\cdot|\Pi|^2}$ where k is the maximum arity of the defined predicates of Σ. Similarly, by Corollary 3.4.7, the computation to determine if $G_1 \sim_\varphi^\Pi G_2$ can be done at a cost of space $k^{|\varphi|\cdot|\Pi|^2}$. Since $k < |\Sigma|$ the algorithm may be made to run in space polynomial in the size of the database. We note that by the results of [13], the class

$$\text{ASPACE}(f(n)) = \bigcup_{c>0} \text{DTIME}(c^{f(n)})$$

for $f(n) \geq log(n)$. Thus, for arbitrary programs Σ the algorithm runs in APSPACE = EXPTIME. If the program is linear, the body of each rule contains at most one defined atom, so we need only guess a single representative at step 2 of the routine ELT. Similarly, in step 5 we need only make one recursive call, so we can dispense with the universal quantifier. This leaves us with a nondeterministic polynomial space bounded algorithm. We can then eliminate the nondeterminism with only a polynomial blow-up in space requirements using Savitch's theorem [114]. This establishes the following upper bounds for data complexity when programs have unbounded arity:

Theorem 3.5.6: For all monadic queries (Π, φ) the set $AS_{Prog}(\Pi, \varphi)$ is in EXPTIME and the set $AS_{Linear}(\Pi, \varphi)$ is in PSPACE.

We now analyze the above algorithm from the point of view of combined complexity. By Lemma 3.4.6 and Corollary 3.4.7, all the representatives required can be represented in space 2^{n^4} where $n = |\Sigma, [B, D], \Pi, \varphi|$. Thus, the algorithm runs in alternating space $2^{poly(n)}$. If the database is linear there is again no need for the universal branching and the algorithm may be made deterministic. Thus, we have

Theorem 3.5.7: The set $AS_{Prog,Monadic}$ is in 2-EXPTIME and the set $AS_{Linear,Monadic}$ is in EXPSPACE.

Finally, we obtain identical bounds on expression complexity as an immediate corollary of this. We will show in the next section that these bounds are tight. Thus, the length of the query is the dominant determinant of the combined complexity.

Theorem 3.5.8: The sets $AS_{Monadic}(\mathbf{D})$ are in 2-EXPTIME for arbitrary databases $\mathbf{D}$ and in EXPSPACE for linear databases $\mathbf{D}$.

Let us now consider the problem of answering queries with respect to the unique names semantics. It is possible to establish results for this semantics which adds to Lemma 3.3.5 and Lemma 3.3.6 the condition that substitutions not identify vertices which the unique names condition requires are distinct. This results in the following equivalent of Lemma 3.5.2. Say that an n-graph with sequence of sources σ *conforms* with an atom $\mathbf{R}(a_1, \ldots, a_n)$ if $\sigma[i] = \sigma[j]$ implies $a_i = a_j$.

Lemma 3.5.9: $\Sigma, [B, D] \not\models^{un} (\Pi, \varphi)$ if and only if there exists an adornment B^a of B and there exists for each atom $e = \mathbf{R}(a_1, \ldots, a_n) \in D$ an adorned representative G_e of $L(\Sigma, \mathbf{R})$ which conforms with e such that $M = [B^a, D]\theta$ is legal and $M \not\models \varphi$, where θ is the substitution that replaces each atom $e \in D$ by G_e.

It is now straightforward to modify the algorithm $FALSE$ to check for conformity of the representatives G_e guessed in step 2. This suffices to turn all the procedures for query answering of this section into procedures valid for the unique names semantics. It follows that all the upper bounds also hold with respect to the unique names semantics.

3.6 Lower Bounds

In this section we show that the upper bounds of the previous section are optimal. The upper bounds were all applicable to monadic queries as well as basic queries. It turns out that each of these bounds can be met with a lower bound result that requires only basic queries. Thus, monadic queries are no more complex than basic queries. All of the reductions of this section will be valid for both the unique names semantics and the standard semantics.

It is known that the data complexity of databases with disjunction is co-NP complete [57]. We note that recursively indefinite databases with fixed program generalize disjunctive databases in one sense, but are more restrictive in another. They are more general because the recursion allows more expressive types of incomplete information. On the other hand, if we fix the program Σ we cannot express all disjunctions. Nevertheless, the data complexity of recursively indefinite databases with fixed program remains co-NP complete.

Theorem 3.6.1: There exists a program Σ and a basic query φ such that the set $AS_\Sigma(\varphi)$ is co-NP complete.

Proof: We show that there exists a reduction from the complement of graph 3–colourability. Let Σ be the program

$$\begin{aligned}
&\mathbf{coloured}(x) : -colour(x, red)\\
&\mathbf{coloured}(x) : -colour(x, blue)\\
&\mathbf{coloured}(x) : -colour(x, green).
\end{aligned}$$

where *red*, *blue* and *green* are constants. Let φ be the query

$$\exists x_1 x_2 y[colour(x_1, y) \wedge colour(x_2, y) \wedge edge(x_1, x_2)].$$

Suppose that $G = (V, E)$ is a (binary) undirected graph with vertices V and edges E. Define the database $\mathbf{D} : \Sigma, [B, D]$ by

$$\begin{aligned}
B &= \{edge(u, v) \mid (u, v) \in E\}\\
D &= \{\mathbf{coloured}(v) \mid v \in V\}.
\end{aligned}$$

Note that each expansion of the database assigns one of the colours to each vertex of the graph, and that the query holds just when some pair of adjacent nodes have the same colour. Thus, $\mathbf{D} \models \varphi$ if and only if every colouring of the graph contains a pair of adjacent nodes with the same colour, i.e., the graph is not three colourable. □

It follows from Theorem 3.6.1 and Theorem 3.5.4 that the data complexity of monadic queries for databases with bounded arity program is also co-NP complete. In order to obtain lower bounds when the arity of programs is not bounded, we will show how to simulate space bounded alternating computations. Given an alternating Turing machine M and an input w, we will construct queries φ and recursively indefinite databases $\mathbf{D}$ such that $\mathbf{D} \models \varphi$ if and only if M accepts w.

Alternating computations may be thought of as binary trees, with each node labeled with a bit 0 or 1, a universal or an existential quantifier, and a Turing machine configuration. We will present databases whose expansions correspond to such trees. The following rules take care of the tree structure and are common to all our simulations.

$$
\begin{aligned}
&\mathbf{bit}(x) : -zero(x)\\
&\mathbf{bit}(x) : -one(x)\\
&\mathbf{quant}(x) : -all(x)\\
&\mathbf{quant}(x) : -some(x)\\
&\mathbf{node}(x) : -leaf(x), \mathbf{bit}(x), \mathbf{quant}(x), \mathbf{id}(x)\\
&\mathbf{node}(x) : -\mathbf{bit}(x), \mathbf{quant}(x), \mathbf{id}(x),\\
&\qquad\qquad left(x, y_1), right(x, y_2), \mathbf{node}(y_1), \mathbf{node}(y_2).
\end{aligned}
\tag{3.6}
$$

Expansion of $\mathbf{node}(x)$ generates a labelled binary tree, with $left(x, y)$ indicating that y is the left successor of x, and similarly for $right(x, y)$. Expanding $\mathbf{id}(x)$ will generate a Turing machine configuration for the node x. The rules for this predicate will be given below.

Not all trees generated by these rules correspond to alternating computations, because the attributes of each node are randomly assigned. We will handle this overgeneration by using a basic query which may be thought of as expressing 'either the tree is not a valid computation or it accepts'. We show that the various error conditions can be represented by basic queries. For example, the bits labeling each node

must be correctly computed from the quantifier labeling the node and the bits labeling the successors of the node. The query that detects errors in the labeling of nodes by bits is

$$\exists xyz\{left(x,y) \wedge right(x,z) \wedge \\ ([all(x) \wedge A(x,y,z)] \vee [some(x) \wedge E(x,y,z)])\}$$

where

$$A(x,y,z) \equiv \{one(x) \wedge (zero(y) \vee zero(z))\} \vee \\ \{zero(x) \wedge one(y) \wedge one(z)\}$$

tests for an error in a universal node computation and $E(x,y,z)$ is a similar query testing for errors in existential nodes.

The predicate **id** will generate the Turing machine configuration. Configurations consist of a linearly ordered set of *cells*. Each cell contains an object which represents the tape symbol written in the cell, but in addition indicates whether the head of the Turing machine is in the cell, and the state of the machine when this is the case. These objects will be called the *symbols* of the machine. It is well known that instead of a transition table we may describe a Turing machine by a function computing the contents of a cell from the previous contents of that cell and the two adjacent cells.

We first establish the lower bound for data complexity in the case of arbitrary programs. Suppose we are given an alternating Turing machine which runs using polynomial space. We assume without loss of generality that all computation paths halt. Let $a_1, \ldots, a_k$ be a collection of constants representing the symbols of the machine. We include in the basic facts B of the database the atom $nonfinal(a)$ for each symbol a which indicates that the machine is in a state which is not final. For an input on which the machine runs using no more than m cells, we use constants $c_1, \ldots, c_m$ to represent these cells and include the atoms

$$first(c_1), next(c_1, c_2), \ldots, next(c_{m-1}, c_m), last(c_m)$$

in the basic facts B to indicate the order. The program Σ associated to this input

consists of the rules (3.6), plus the rules

$$\begin{aligned} \mathbf{id}(x) &: -\mathbf{cell}(x, c_1), \ldots, \mathbf{cell}(x, c_m) \\ \mathbf{cell}(x, y) &: -contains(x, y, a_1) \\ &\vdots \\ \mathbf{cell}(x, y) &: -contains(x, y, a_k). \end{aligned} \tag{3.7}$$

Here the predicate $contains(x, y, z)$ is intended to express that cell y of the configuration of node x contains the symbol z. The root of the computation tree is represented by a constant r and the initial configuration $a_{i_1} \ldots a_{i_m}$ of the computation is represented by including the atoms $contains(r, c_1, a_{i_1}), \ldots, contains(r, c_m, a_{i_m})$ in the basic facts B. The set B also contains the atoms $left(r, b_1)$,$right(r, b_2)$, and the set of defined facts D of the database consists of the atoms $\mathbf{node}(b_1), \mathbf{node}(b_2)$, from which the remainder of the tree is generated.

Three sorts of errors need to be eliminated from the computation trees generated by these rules. First, the computation may halt too soon, i.e., some leaf of the tree may have a nonhalting configuration. This is detected by the query

$$\exists xyz(leaf(x) \wedge contains(x, y, z) \wedge nonfinal(z)).$$

Next, the quantifier labeling a node must be correctly determined by the state of the machine at the node. This is ensured by including in B the atom $universal(a)$ for each symbol indicating a universal state, and $existential(a)$ for each symbol indicating an existential state. Then the query

$$\begin{aligned} \exists xyz(contains(x, y, z) \wedge \{&[all(x) \wedge existential(z)] \\ &\vee [some(x) \wedge universal(z)]\}) \end{aligned}$$

detects a labeling error. Finally, we need to ensure that the contents of the cells have been correctly computed. This is done by including in B the atom $lerror(a, b, c, d)$ for each tuple (a, b, c, d) of symbols such that d is *not* the symbol in the left successor calculated from the symbols a, b, c. One form of transition error is then detected by the

query

$$\exists x\, y z_1 t_1 z_2 t_2 z_3 t_3 t_2' \{$$
$$left(x, y) \wedge next(z_1, z_2) \wedge next(z_2, z_3) \wedge$$
$$\bigwedge_{i=1}^{3} contains(x, z_i, t_i) \wedge$$
$$contains(y, z_2, t_2') \wedge lerror(t_1, t_2, t_3, t_2')\}.$$

A similar technique handles errors at the boundaries of configurations and right transitions.

Let φ be the query formed by taking the disjunction of the queries described above with the atom $one(r)$. Then the resulting query follows from the database described above just when the alternating Turing machine accepts the input. Note that the query is independent of both the input and the alternating Turing machine. If the alternating Turing machine runs in space polynomial in the size of the input (i.e., m is polynomial in the size of the input) then the total size of the database is polynomial in the size of the input. Since APSPACE=EXPTIME we have shown:

Theorem 3.6.2: There exists a basic query φ such that the set $AS_{Prog}(\varphi)$ is complete for EXPTIME under logspace reductions.

In order to prove the lower bound for linear programs it is convenient to introduce the following notion. Let Σ be a program whose defined predicates are ranked according to $P_1, P_2, \ldots$. We say that the program is *weakly linear* if each rule with predicate P_i in the head contains in the body no occurrences of the predicates P_j for $j > i$, and at most one occurrence of the predicate P_i. The following lemma demonstrates how weakly linear programs may be translated into linear programs.

Lemma 3.6.3: Let Σ be a weakly linear program with N defined predicates. Then there exists a linear program Σ' of size $|\Sigma|^{cN}$ such that for every defined predicate $\mathbf{P}$ the expansions of $\mathbf{P}(x_1, \ldots, x_n)$ by Σ' are exactly the same as the expansions by Σ.

Proof: We proceed by induction on N. If $N = 1$ there is nothing to prove. Assume therefore that we have a linear program Σ_1 of size $|\Sigma|^{c(N-1)}$ which is expansion-equivalent to Σ with respect to the predicates $\mathbf{P}_1 \ldots, \mathbf{P}_{N-1}$. Let $\mathcal{Q}$ be the set of

predicates of this program. We suppose that the rules of Σ with predicate $\mathbf{P}_N$ in the head are written as

$$r: \qquad \mathbf{P}_N(\mathbf{x}_r) : -\exists \mathbf{y}_r [B_r, \mathbf{Q}_{r,1}(V_{r,1}), \ldots, \mathbf{Q}_{r,n_r}(V_{r,n_r}), \mathbf{P}_N(V_{r,n_r+1})]$$

Here B_r is a conjunction of the basic atoms of the rule. The notation $\mathbf{Q}_{r,j}(V_{r,j})$ indicates a literal in the body of the rule r which has predicate $\mathbf{Q}_{r,j} \in \mathcal{Q}$ and sequence of variables and constants given by $V_{r,j}$. For example, a literal $\mathbf{Q}(x, y, x, c)$ is decomposed as $\mathbf{Q}_{r,j} = \mathbf{Q}$ and $V_{r,j} = (x, y, x, c)$. We suppose that the variables of the rules of Σ and Σ_1 have been standardized apart.

Intuitively, the transformed linear program will simulate the expansions of such rules by expanding the predicates $\mathbf{Q}_{r,j}$ one at a time. We use additional "indexed" predicates to keep track of how much of the rule has been expanded, i.e., where to continue once $\mathbf{Q}_{r,j}$ has been fully expanded. It is also necessary to "stack" the arguments $\mathbf{x}_r, \mathbf{y}_r$ while expanding $\mathbf{Q}_{r,j}$. Formally, we proceed as follows.

For each rule r of Σ with $\mathbf{P}_N$ in the head and for each predicate $\mathbf{Q} \in \mathcal{Q}$ of arity a the program Σ' will have a predicate $\mathbf{P}_N^{r,j}_\mathbf{Q}$ of arity $|\mathbf{x}_r|+|\mathbf{y}_r|+a$ where $1 \leq j \leq n_r$. Intuitively, this predicate states that we are in the process of expanding rule r at position j. This tells us where to continue once the predicate $\mathbf{Q}$ has been fully expanded. The expansion of each rule r of Σ with $\mathbf{P}_N$ in the head is started off by the rule

$$\mathbf{P}_N(\mathbf{x}_r) : -B_r, \mathbf{P}_N^{r,1}_\mathbf{Q}_{r,1}(\mathbf{x}_r, \mathbf{y}_r, V_{r,1})$$

of Σ'.

For each recursive rule $\mathbf{Q}_1(U_1) : -C, \mathbf{Q}_2(U_2)$ of Σ_1, where C is a conjunction of basic atoms, and for each rule r of Σ with $\mathbf{P}_N$ in the head, the program Σ' will contain a rule

$$\mathbf{P}_N^{r,j}_\mathbf{Q}_1(\mathbf{x}_r, \mathbf{y}_r, U_1) : -C, \mathbf{P}_N^{r,j}_\mathbf{Q}_2(\mathbf{x}_r, \mathbf{y}_r, U_2).$$

This rule does one step of the expansion of $\mathbf{Q}_{r,j}$ and continues to store the return point. Similarly, for each "basis rule" $\mathbf{Q}(U) : -C$ of Σ_1, the program Σ' will contain a rule

$$\mathbf{P}_N^{r,j}_\mathbf{Q}(\mathbf{x}_r, \mathbf{y}_r, U) : -C, \mathbf{P}_N^{r,j+1}_\mathbf{Q}_{r,j+1}(\mathbf{x}_r, \mathbf{y}_r, V_{r,j+1})$$

if $j<n_r$ (continue with the next step in rule r) or a rule

$$\mathbf{P}_N^{r,j}_\mathbf{Q}(\mathbf{x}_r, \mathbf{y}_r, U) : -C, \mathbf{P}_N(V_{r,n_r+1})$$

if $j = n_r$ (do the recursive call in rule r.) If the rule r does not have an occurrence of the predicate $\mathbf{P}_N$ then neither does this last rule. Finally, we add to Σ' all the rules of Σ_1. A straightforward induction shows that Σ' and Σ are expansion-equivalent for the predicates $\mathbf{P}_1, \ldots, \mathbf{P}_N$.

To calculate the size of the program Σ', note that the dominant contribution to this program is obtained from Σ_1 by taking $|\Sigma|$ copies of each rule and expanding each copy by an amount $|\Sigma|$. That is, $|\Sigma'| \leq |\Sigma|^c.|\Sigma_1| = |\Sigma|^{cN}$. □

To establish the lower bound on data complexity for recursively indefinite databases containing only linear programs, we modify the rules used above to generate computations of a deterministic space bounded Turing machine. Instead of the rules (3.6) we use

$$\begin{aligned} &\mathbf{node}(x) : -leaf(x), \mathbf{id}(x) \\ &\mathbf{node}(x) : -\mathbf{id}(x), nextnode(x, y), \mathbf{node}(y). \end{aligned} \tag{3.8}$$

to generate a sequence of Turing machine configurations representing the computation. We reuse the rules (3.7) to generate the configurations. Note that this program is not linear. However, it is weakly linear, the ranking of defined predicates being $\mathbf{cell}, \mathbf{id}, \mathbf{node}$. Thus, we may use Lemma 3.6.3 to translate it to a linear program which is expansion-equivalent. This leads to a blow-up in the size of the program, but it is only a polynomial blow-up since we have a fixed number of defined predicates. The remainder of the database is as before, and the query is readily modified to express that either the expansion is not a valid computation or else it accepts. This proves

Theorem 3.6.4: There exists a basic query φ such that the set $AS_{Linear}(\varphi)$ is complete for PSPACE under logspace reductions.

We now turn to lower bounds for expression complexity. The idea of the proof is similar to the proof just discussed, and we reuse the rules (3.6) which generate the computation tree. However, the Turing machine configurations will be handled somewhat differently. With each cell we will associate two linear sequences of zeros and

ones. One of these sequences represents the contents of the cell. The other will be the *sequence number* of the cell, and represents the position of the cell in the configuration. Expanding $\mathbf{id}(x)$ using the rules below generates a configuration.

$$
\begin{aligned}
&\mathbf{id}(x) : -firstcell(x,y), \mathbf{cellseq}(x,y)\\
&\mathbf{cellseq}(x,y) : -lastcell(x,y), \mathbf{cell}(x,y)\\
&\mathbf{cellseq}(x,y) : -\mathbf{cell}(x,y), nextcell(y,z), \mathbf{cellseq}(x,z)\\
&\mathbf{cell}(x,y) : -cellof(x,y), state(y,s), firstbit(s,u), \mathbf{bitseq}(s,u),\\
&\qquad\qquad cellnum(y,t), firstbit(t,v), \mathbf{bitseq}(s,v)\\
&\mathbf{bitseq}(x,y) : -lastbit(x,y), \mathbf{bit}(y)\\
&\mathbf{bitseq}(x,y) : -\mathbf{bit}(y), nextbit(y,z), \mathbf{bitseq}(x,z).
\end{aligned}
\tag{3.9}
$$

Here expanding $\mathbf{cellseq}(x,y)$ generates the sequence of cells of configuration x, with y as the first cell. An atom $cellof(x,y)$ indicates that y is one of the cells of x. Each cell y has associated with it objects s and t, which point to sequences of bits representing the state and the sequence number respectively. These sequences of bits are obtained by expanding an atom of the form $\mathbf{bitseq}(x,y)$, which generates a sequence of bits associated to object x, of which the first bit is y.

Let the database $\mathbf{D}$ have program Σ consisting of the rules (3.6) and (3.9). The defined facts D of the database will be the single atom $\mathbf{node}(r)$. The set of basic facts B of the database is empty. We will describe for every alternating Turing machine which runs in space 2^m on an input of length n, a basic query φ such that $\mathbf{D} \models \varphi$ if and only if the Turing machine accepts that input. As before, the query will be the disjunction of a number of "error conditions" with the query $one(r)$.

We must ensure that for each cell the sequence number and the state number are comprised of sufficiently many bits. A sequence number or a state number x containing fewer than m bits is detected by the query

$$
\bigvee_{1 \le i < m} \exists x_1 \ldots x_i \{ firstbit(x,x_1) \wedge nextbit(x_1,x_2) \wedge \ldots \\
\ldots \wedge nextbit(x_{i-1},x_i) \wedge lastbit(x,x_i) \}.
$$

To insure that sequence numbers are assigned properly, and that the number of cells in the configuration is exactly 2^m, we check three conditions. The sequence number

of the first cell must be 0, the sequence number of the last cell must be $2^m - 1$, and the sequence number of the successor of a cell must be one greater than the sequence number of the cell. The first of these conditions is guaranteed by the query

$$\exists xyt\{firstcell(x,y) \wedge cellnum(y,t) \wedge \\ \exists t_1 \ldots t_m[firstbit(t,t_1) \wedge nextbit(t_1,t_2) \wedge \ldots \\ \ldots \wedge nextbit(t_{m-1},t_m) \wedge \bigvee_{1 \leq i \leq m} one(t_i)]\}$$

A similar query checks the that last sequence number is $2^m - 1$ and that no cell with a sequence number $2^m - 1$ has a successor. Assuming that the first bit of a number is the least significant bit, the following query detects successive cells with incorrect sequence numbers

$$\exists xyuv\{nextcell(x,y) \wedge cellnum(x,u) \wedge cellnum(y,v) \wedge \\ \exists t_1 \ldots t_m s_1 \ldots s_m\{firstbit(u,t_1) \wedge firstbit(v,s_1) \wedge \\ nextbit(t_1,t_2) \wedge \ldots \wedge nextbit(t_{m-1},t_m) \wedge \\ nextbit(s_1,s_2) \wedge \ldots \wedge nextbit(s_{m-1},s_m) \wedge \\ \bigvee_{1 \leq i \leq m} [one(t_1) \wedge \ldots \wedge one(t_{i-1}) \wedge zero(t_i) \wedge \\ \left\{ \left(\bigvee_{j<i} one(s_j) \right) \vee zero(s_i) \vee \bigvee_{i<j \leq m} diff(t_j,s_j) \right\}]\}\}$$

where $diff(x,y)$ expresses that the bits are different.

As we have already mentioned, the symbol occupying a cell is represented in the expansion by the state sequence associated with that cell. For example, if the code for the symbol a is the binary sequence 010 then the formula

$$a(x) = \exists y_1 y_2 y_3[state(x,u) \wedge firstbit(u,y_1) \wedge nextbit(y_1,y_2) \wedge \\ nextbit(y_2,y_3) \wedge zero(y_1) \wedge one(y_2) \wedge zero(y_3)]$$

detects cells x containing the symbol a. Using this idea it is straightforward to write formulae which check that the cells of the root node are correctly initialized. Suppose now that we are given the transition functions f_l, f_r of the alternating Turing machine. The left successor of a configuration should have $f_l(a,b,c)$ in cell i if it has a,b,c in

cells $i-1, i, i+1$ respectively. The following formula detects a transition error.

$$\exists xyuvwt$$
$$\{left(x,y) \wedge cellof(y,t) \wedge cellof(x,u) \wedge nextcell(u,v) \wedge nextcell(v,w) \wedge$$
$$\exists z_1, z_2 [cellnum(v,z_1) \wedge cellnum(t,z_2) \wedge samenum_m(z_1,z_2) \wedge$$
$$\bigvee_{d \neq f_l(a,b,c)} \{a(u) \wedge b(v) \wedge c(w) \wedge d(t)\}]\}$$

Here $samenum_m(x,y)$ is the formula

$$\exists z_1 \ldots z_m t_1 \ldots t_n \{firstbit(x,z_1) \wedge nextbit(z_1,z_2) \wedge \ldots \wedge nextbit(z_{m-1},z_m) \wedge$$
$$firstbit(x,t_1) \wedge nextbit(t_1,t_2) \wedge \ldots \wedge nextbit(t_{m-1},t_m) \wedge$$
$$\bigvee_{i=1}^{m} [one(z_i) \wedge one(t_i)] \vee [zero(z_i) \wedge zero(t_i)]\}$$

expressing that the first m bits of x and y are identical. Similar formulae check the right transitions, the labeling of nodes by quantifiers and that all leaves of the tree are terminal. It is straightforward to check that if m is polynomial in the input size then so is the disjunction of the queries described above. (We comment that if one insists that queries be in disjunctive normal form, this is no longer true, because of the formula *samenum*. However, it is still possible to show the same result, by using constants to encode the bits instead.) This establishes the following:

Theorem 3.6.5: There exists a database $\mathbf{D}$ such that the set $AS_{Basic}(\mathbf{D})$ is complete for 2-EXPTIME under logspace reductions.

A similar proof yields a lower bound for linear databases. In this case the program generates a single branch representing a deterministic space bounded computation using rules (3.8) and (3.9), linearized using Lemma 3.6.3.

Theorem 3.6.6: There exists a linear database $\mathbf{D}$ such that the set $AS_{Basic}(\mathbf{D})$ is complete for EXPSPACE under logspace reductions.

These lower bounds for expression complexity also show that the upper bounds for combined complexity obtained in the previous section are tight. We comment that all our lower bound results except Theorem 3.6.1 required disjunctive queries. We do not know if the assumption that queries are conjunctive leads to a decrease in complexity.

Our present techniques for obtaining upper bounds do not seem to be sufficient to yield an improvement in complexity.

3.7 Decidable Intensional Queries

We now briefly reconsider intensional queries. In spite of the general undecidability result, there exist classes of intensional queries besides monadic queries which are decidable. We demonstrate one such class, and comment on the possibility of classifying intensional queries according to their decidability.

Courcelle [25] has established a very general theorem concerning decidability of properties of hypergraph grammars. He defines for each natural number k a sorted second order language $\mathcal{L}_k$. There are two sorts, the edge sort $\mathbf{e}$ and the vertex sort $\mathbf{v}$. The language has first order quantifiers $\exists_{\mathbf{e}}, \exists_{\mathbf{v}}$ of both sorts and also monadic second order quantifiers of both sorts. The language has constants c_i of sort $\mathbf{v}$ for $i = 1, \ldots, k$, interpreted to denote the sources of a k-graph. For each label a there is a predicate symbol $edg_a(u, v_1, \ldots, v_n)$ of sort $\mathbf{ev} \ldots \mathbf{v}$, interpreted to mean that the edge u has vertices $v_1, \ldots, v_n$ and label a. Besides this, the language can express equality of vertices and set membership.

Theorem 3.7.1: (Courcelle)

Let φ be a sentence of $\mathcal{L}_k$ and let Γ be a context free hypergraph edge replacement grammar of type k. It is decidable whether $G \models \varphi$ for all $G \in L(\Gamma)$.

The decidability of the class of monadic queries in recursively indefinite databases is a straightforward consequence of this result. However, Courcelle's proof does not yield the optimal bounds on complexity derived in the present chapter. (His procedure requires an exponential for every level of quantification.) We now show how some other classes of decidable queries can be derived from this result.

Call a program Σ *singular* if it has a single defined predicate $\mathbf{R}$ and there exists an index i such that all rules $\mathbf{R}(x_1 \ldots x_n) : -\psi$ have all occurrences of $\mathbf{R}$ in the body ψ of the form $\mathbf{R}(x_1 \ldots x_{i-1}, y, x_{i+1} \ldots x_n)$ for some y, which may be either a variable or

a constant. For example, the linear program

$$\mathbf{R}(x_1, x_2) : -edge(x_1, x_2)$$
$$\mathbf{R}(x_1, x_2) : -\mathbf{R}(x_1, y), edge(y, x_2)$$

defining transitive closure is singular, with $i = 2$.

If Σ is a singular program defining $\mathbf{R}$ then the query $\mathbf{R}(x_1 \ldots x_{i-1}, z, x_{i+1}, \ldots x_n)$ is expressible in monadic second order logic as

$$Q_{\mathbf{R}}(x_1 \ldots x_{i-1}, z, x_{i+1}, \ldots x_n) \equiv \forall U \left\{ \left[\forall x_i \bigwedge_{r \in \Sigma} r' \right] \Rightarrow z \in U \right\}$$

where r' is obtained from r by substituting $y \in U$ for each occurrence of an atom of the form $\mathbf{R}(x_1 \ldots x_{i-1}, y, x_{i+1} \ldots x_n)$ in the rule r, including the head. We assume that each rule r has been normalized by making the sequence of variables in the head be $x_1 \ldots x_n$ and binding all other variables in the body with existential quantifiers. For example, in the case of transitive closure the query $\mathbf{R}(x_1, z)$ is expressed by

$$\begin{aligned} \forall U \{ \forall x_2 [(edge(x_1, x_2) \Rightarrow x_2 \in U) \wedge \\ (\exists y (y \in U \wedge edge(y, x_2)) \Rightarrow x_2 \in U)] \\ \Rightarrow z \in U \}. \end{aligned}$$

We note that the query $Q_{\mathbf{R}}$ contains only basic predicates, and is preserved under homomorphism. Thus, it follows from Lemma 3.3.7 and Theorem 3.7.1 that intensional queries containing only basic predicates and predicates defined by singular programs are decidable in recursively indefinite databases. In particular, queries about transitive closure are decidable. (Wanke [129] has investigated the complexity of deciding if all graphs in the language generated by a graph grammar are connected.) The class of singular queries may be extended to allow programs with more than one defined predicate if the interaction of predicates is suitably constrained.

One might hope to make this technique general by providing an algorithm that translates intensional queries into monadic second order logic, whenever this is possible. Unfortunately, no such algorithm can exist. Gaifman et al. [39] establish a 'Rice' style theorem stating that any 'non-trivial' property of Datalog programs containing boundedness is undecidable. This result applies to monadic second order expressibility

because boundedness implies first order expressibility, and therefore implies monadic second order expressibility also. Hence, the translation to monadic second order logic cannot be made automatic for all cases in which it is possible.

Finally, we note that there are some examples of apparently straightforward inferences that do not fall into any of the decidable cases we have mentioned. One of these is Example 3.1.4, which involves a double-sided recursion. We do not know of any approach that would include such rules in a decidable case.

3.8 Discussion

Closely related to the completion semantics discussed in Section 3.1 are the logical databases with 'skolem rules' of Imielinski [57]. These are logical databases containing rules of the form $\forall \mathbf{x}[\varphi \Rightarrow \exists \mathbf{y}(\psi)]$ where ψ is an atom and φ is a conjunctions of atoms, and allowing recursion. Imielinski establishes that logical databases containing such rules have an undecidable query problem for atomic queries, although he is able to identify a variety of sufficient conditions under which queries are decidable. Like the completion, Skolem rules are satisfied in models containing 'infinite chains' (see discussion after Example 3.1.2.) It would be interesting to have a careful comparison between the completion semantics and the minimization semantics we have adopted. Some of Imielinski's methods may be applicable to recursively indefinite databases. Conversely, we suspect that the techniques we have used could be applied to obtain decidable cases for the completion semantics and for databases with skolem rules.

Courcelle [23] has previously used graph grammars to show the decidability of a problem related to a notion of optimization of Datalog programs due to Sagiv and Naughton [94]. We expect that these techniques will find other applications to analysis of Datalog programs.

The notion of adorned glue type is a modification of the notion of glue type due to Lenguaer and Wanke [71]. They obtain complexity results for a variety of specific queries, such as k-colourability, Hamiltonicity, connectivity etc., on the graph languages generated by graph grammars. For monadic queries, their equivalence relation would

correspond to the relation defined on graphs over the basic relations by $G_1 \approx^{\Pi}_{\varphi} G_2$ if for all H, $\Pi^{\infty}(G_1 \circ H) \models \varphi$ if and only if $\Pi^{\infty}(G_2 \circ H) \models \varphi$. While this is identical to our relation for basic queries, it does not appear yield optimal algorithms in the case of monadic queries. Our use of adornment derives from Cosmadakis et al. [21] who use similar techniques to prove boundedness of monadic Datalog. The decidability of monadic queries generalizes their result that containment of monadic Datalog is decidable.

One might consider expanding the class of queries permitted to include second order formulae. In order to obtain decision procedures for such queries using Theorem 3.7.1 one must first check preservation under homomorphism. In general, this appears to be undecidable. However, the class of positive queries in the language obtained by adding a 'monadic' fixpoint operator to first order logic appears to be a reasonable class of queries. All such queries are preserved under homomorphism, and Theorem 3.7.1 yields the decidability of this class.

Rather than pursue such extensions of the decidable class of queries, we will investigate in later chapters a variety of extensions of the class of allowable definitions, and seek to determine to what extent the basic queries remain decidable under such extensions. In the next chapter, we consider the effect of negation in the body of rules. Then, after a preparatory investigation of indefiniteness resulting from linear order in Chapter 5, we consider the effect of linear order constraints in definitions in Chapter 6.

Chapter 4

Negation and Inequality

4.1 Introduction

In our discussion so far we have considered only programs and queries which contain no negation. We now relax this restriction, and study the effect of allowing various forms of negation. The results of this chapter[1] will provide a reasonably tight delineation of how negation may occur without losing the decidability of basic queries. The decidable class turns out to be small, but still contains some useful cases. However, admitting negation comes at the cost of a substantial increase in complexity.

Adding negation to programs quickly leads to undecidability of basic queries. However, basic queries remain decidable in recursively indefinite databases whose programs contain only basic predicates in negated positions. At the level of combined complexity, querying these databases is not more complex than querying database not using negation. However, this is misleading, since in fact answering such queries requires considerable extra work. This comes about because expansions may now be inconsistent, and it is necessary to ensure that only consistent expansions are considered by the decision procedure.

This is reflected in the data complexity results for such databases, shown in Table 4.1. (In general these results require an unbounded set of basic predicates.) Recall that for negation free programs of bounded arity we were able to establish co-NP data complexity. Once basic predicates may occur negated in the program, data complexity in the bounded arity case may be as high as EXPTIME complete, in the case of non-linear programs. (This is not known to be a strict increase of course, so it merely

[1] The results of this chapter have been presented in condensed form in [87].

Program restrictions		Data Complexity
Fixed program		co-NP
Bounded arity	linear	PSPACE
	nonlinear	EXPTIME
Unbounded arity	linear	EXPSPACE
	non-linear	2-EXPTIME

Table 4.1: Data complexity for programs with negated base predicates

suggests higher complexity.) Things are even worse in the case of non-linear programs of unbounded arity, for which basic queries have 2–EXPTIME complete data complexity. That is, data complexity for programs with negated basic predicates is as high as combined complexity in the negation free case. These results may be summarized by saying that negating basic predicates leads to an exponential increase in complexity.

We also consider the effect of inequality in queries. The results here turn out to be entirely negative: there does not appear to be an interesting class of recursive rules for which basic queries containing inequality are decidable: linear monadic definitions (not containing negation or $\neq$) suffice to give undecidability of basic queries.

The structure of the chapter is as follows. Section 4.2 sets up the semantic framework for programs with negation and extends the semantics of the previous chapter to databases containing such programs. In Section 4.3 we show that basic queries remain decidable in programs in which only basic predicates may be negated, and we study the effects on complexity of moving to this more general class of programs. Section 4.4 shows that decidability also holds for a slightly more general class of programs, namely those with only two strata, but if we move to a number of strata greater than two consequence becomes undecidable. Section 4.5 shows that the combination of recursion and inequality in the query is undecidable.

4.2 Preliminaries

The problem of providing semantics for logic programs containing negated literals in the body of rules has been a subject of intensive study. There are a numerous competing

semantics for such programs, prominent among which are the well-founded semantics [41], the stable semantics [42] and the perfect model semantics [106]. There is, however, a broad class of programs about which there is general agreement: the *stratified* programs introduced by Apt, Blair and Walker [4]. All of the semantics of [41, 42, 106] are equivalent on this class of programs. We will see that further restrictions are required even on the class of stratified programs in order to obtain decidable classes of databases. The main decidable case we consider, in which only base predicates are negated, falls entirely within the stratified programs. Thus, this class is sufficiently general for our purposes.

We assume that programs containing negation are *safe*, that is, satisfy the syntactic constraint that all variables which occur in the head of a rule, or in a negated literal, also occur in a positive literal in the body of the rule. This constraint has the consequence that in computing fixpoints it suffices to consider only constants mentioned in the set of basic facts from which the fixpoint computation starts. A program is *stratified* if it is possible to rank the set of predicates into a number of disjoint *strata* $Pred = S_0 \cup S_1 \cup \ldots \cup S_n$, subject to the constraints that

1. if a rule contains a predicate from stratum i in its body then the predicate in the head of the rule is in stratum at least i, and

2. if a rule contains a predicate from stratum i *negated* in its body then the predicate in the head of the rule is in stratum at least $i + 1$.

All basic predicates may be considered to be in the lowest stratum S_0. Given a stratification of the predicates of a program, one may partition the program itself as $\Sigma = \Sigma_0 \cup \ldots \cup \Sigma_n$, where the program Σ_i contains all the rules with head in S_i.

The *stratified semantics* of the program is then defined as follows. Given a set M of basic atoms, we extend to the model

$$\Sigma_n^\infty(\Sigma_{n-1}^\infty(\ldots\Sigma_0^\infty(M)))$$

of the defined predicates. That is, we first compute the fixpoint $M_0 = \Sigma_0^\infty(M)$ corresponding to the lowest stratum. Since the rules in Σ_0 contain no negated atoms this

is just the standard fixpoint computation. In the next step, we use M_0 as a basis for the computation for the fixpoint of the rules Σ_1. Here we assume for the purpose of applying the rules Σ_1 that the atoms in M_0 concerning the predicates S_0 are true, and that all atoms in the predicates S_0 not in the set M_0 are false. Note that this is legitimate because the stratification guarantees that the only predicates which may occur negated in Σ_1 are in the set S_0, and these predicates are not further extended by rules in Σ_1. Thus, the predicates in S_0 may be considered to have their extensions totally computed. After reaching the fixpoint of the rules Σ_1 the same may be said for the predicates S_1, and we proceed with the computation of higher strata.

There may be several different stratifications of a given program. It is therefore interesting that the semantics is *independent* of the stratification: the fixpoint computed is always the same, regardless of the stratification used. Hence we are justified in introducing the notation $\Sigma^\infty(M)$ for this fixpoint, which does not refer to any particular stratification.

Example 4.2.1: If A, B, E and V are basic predicates then the program

$$\mathbf{R}(x,y) :- E(x,y)$$
$$\mathbf{R}(x,y) :- E(x,z), \mathbf{R}(z,y)$$
$$\mathbf{S}(x,y) :- V(x), V(y), \neg\mathbf{R}(x,y)$$
$$\mathbf{T}(x) :- A(x), B(y), \neg\mathbf{S}(x,y)$$

may be stratified into three strata: $S_0 = \{A, B, E, V, \mathbf{R}\}$, $S_1 = \{\mathbf{S}\}$ and $S_2 = \{\mathbf{T}\}$. (There is also a stratification into four strata obtained by placing $\mathbf{R}$ into a stratum of its own, rather than with the basic predicates.) Let M contain the basic facts $A(a), E(a,b), E(b,c), B(c)$ and $V(a), V(b), V(c)$. The stratified semantics is computed for this model as follows. The program Σ_0 consists of the first two rules of Σ, the program Σ_1 contains the third rule and the program Σ_2 contains the last rule. The first stage of the computation calculates $\Sigma_0^\infty(M)$ which yields the interpretation $\mathbf{R}(a,b), \mathbf{R}(b,c), \mathbf{R}(a,c)$ for the predicate $\mathbf{R}$. This interpretation, and the interpretation M of the basic predicates are now fixed, and in computing $\Sigma_1^\infty(\Sigma_0^\infty(M))$ we obtain $\mathbf{S}(a,a), \mathbf{S}(b,b), \mathbf{S}(c,c), \mathbf{S}(b,a), \mathbf{S}(c,b), \mathbf{S}(c,a)$ for the interpretation of the predicate $\mathbf{S}$. Finally, computing the last stratum yields the

interpretation $\mathbf{T}(a)$ for the predicate $\mathbf{T}$. □

The definition of the semantics now proceeds as before. We change notation slightly, since it will not be necessary to refer separately to the basic and defined facts. Thus, a recursively indefinite database now consists of a stratified program Σ and a set of facts D in both the basic and defined predicates. We take $Mod(\Sigma, D)$ to be the set of models M' of the form $\Sigma^{\infty}(M)$ for some set of basic atoms M, and for which M' supports all facts in D. The consequence relation $\models$ is interpreted with respect to this set of models.

Example 4.2.2: Let Σ be the program of Example 4.2.1, and let D be the database containing the basic facts $V(a), V(b), V(c)$ and the defined facts $\mathbf{R}(b, c), \mathbf{S}(a, c)$. The first of these defined facts asserts that there exists a path from b to c; the second that there is no path from a to c. It follows that there is no path from a to b. That is, we have the relation $\Sigma, D \models \mathbf{S}(a, b)$. □

We emphasize that although we have now admitted negative information in our databases, we retain the "open world assumption". In other words, we still *do not* assume that any fact not entailed by the database is false, nor do we make any other such "completion" assumption. Negative information in the database does, however, serve to prevent certain rules from providing the derivation of defined facts in the database.

Example 4.2.3: The database with program

$$\mathbf{P} : -A$$
$$\mathbf{P} : -B$$
$$\mathbf{Q} : -\neg B$$

and facts $\{\mathbf{P}, \mathbf{Q}\}$ entails the query A. Here the defined fact $\mathbf{Q}$ must expand as $\neg B$, which prevents the second rule from providing the derivation of $\mathbf{P}$, and leaves only the first rule as a possible justification for this fact. □

Notice that in this example we also have that $\neg B$ is a consequence of the database. This illustrates that once one has negation in programs, it becomes possible for non-trivial queries containing negation to be consequences of the database. In this example

the negative information is explicitly obtained by the expanding the database. However, negative information may follow in less obvious ways.

Example 4.2.4: Consider the program containing the two rules

$$\mathbf{A} :- P(x)$$
$$\mathbf{B} :- Q(x), \neg\mathbf{A}$$

Then the database $\Sigma, \{\mathbf{B}\}$ entails the query $\exists x[\neg P(x)]$. For, in order for $\mathbf{B}$ to be an element of the stratified fixpoint, there must be a constant a such that $Q(a)$ holds. Furthermore, $\mathbf{A}$ must not be an element of the fixpoint, so we cannot have $P(a)$. In other words, the query is satisfied with $x = a$. □

These examples suggests that in studying databases involving negation one should allow negative literals also as part of queries. However, we will not do this, primarily because we do not presently know how to answer such queries. The procedures we present for dealing with negation in various classes of rules do not appear to admit modifications that would make them applicable to queries containing negation.

In the negation free case, there are two ways in which databases can be inconsistent: a database may contain a defined fact whose definition is empty, or the definition may require equality of two distinct constants under the unique names assumption (see Example 3.2.2). Both of these types of inconsistency are straightforward to detect. Databases containing negation in the program, however, may be inconsistent in more subtle ways. Our proofs of lower bounds for the decidable cases will show that most of the complexity of query processing now lies in handling inconsistency. It will follow from these proofs that detecting inconsistency is as hard as answering queries.

4.3 Negating Base Predicates

In this section we consider programs in which only base predicates are negated. It has previously been noted by Courcelle [24] that for this class of programs basic queries are decidable. However, he has not considered the complexity of querying such databases. We will show in the next section that certain generalizations this class remain decidable.

Example 4.3.1: Consider the program Σ consisting of the rules

$$\mathbf{A}(x,y) :- R(x), E(x,z), \neg R(z), \mathbf{A}(z,y)$$
$$\mathbf{A}(x,y) :- G(x), E(x,z), \neg G(z), \mathbf{A}(z,y)$$
$$\mathbf{A}(x,y) :- R(x), E(x,y), \neg R(y)$$
$$\mathbf{A}(x,y) :- G(x), E(x,y), \neg G(y)$$

in which the predicates R, G and E are basic and $\mathbf{A}$ is the only defined predicate. If D is the database containing the basic facts $R(a), \neg G(a), R(b), \neg G(b)$ and the defined fact $\mathbf{A}(a,b)$ then

$$\Sigma, D \models \exists xyz[R(x) \wedge E(x,y) \wedge G(y) \wedge E(y,z) \wedge R(z)].$$

Intuitively, the predicate $\mathbf{A}(x,y)$ asserts that there exists a path from x to y through vertices which alternate in colour between "red" (R) and "green" (G). Notice that certain of the expansions of this program are inconsistent. For example, we cannot expand $\mathbf{A}(a,b)$ using the first and then the third rule, because this yields the inconsistent set of facts

$$R(a), E(a,c), \neg R(c), R(c), E(c,b), \neg R(c).$$

Similarly, expansions must alternate between the first and second rule, or else they will be inconsistent. □

The decidability of programs with negated base predicates follows from the negation-free case by a simple transformation. Given a program Σ, introduce for each base predicate P a new base predicate $not\text{-}P$ of the same arity. Let Σ' be the program obtained from Σ by replacing each occurrence of a literal of the form $\neg P(\mathbf{x})$ by the atom $not\text{-}P(\mathbf{x})$. For each query Φ take Φ' to be the query obtained by disjoining the queries

$$\exists \mathbf{x}[P(\mathbf{x}) \wedge not\text{-}P(\mathbf{x})] \tag{4.1}$$

where P ranges over the base predicates in Σ. We defer the proof of the following result to Section 4.4, where we establish a more general version.

Lemma 4.3.2: For every program Σ in which only base predicates are negated, and for every basic query Φ we have $\Sigma, D \models \Phi$ if and only if $\Sigma', D' \models \Phi'$.

Let us now analyze the consequences of this transformation for the complexity of query processing. First, notice that the size of the transformed program Σ' is linear in the size of the size of the program Σ, the only cost being in the few extra bits required to represent the new predicates. The size of the transformed query Φ' depends on both the size of the query Φ and the number of basic predicates which occur negated in the program Σ: we have that $|\Phi'|$ is linear in $|\Sigma| + |\Phi|$. It follows from these considerations that the transformation affects neither expression complexity nor combined complexity. Upper bounds for these problems are unchanged from the corresponding bounds for the negation free case. Similarly, if we consider the problem of data complexity for databases whose programs contain only a fixed, finite set of negated base predicates, then the reduction uses a fixed query Φ', so again upper bounds for this problem may be lifted directly from the negation free case.

However, notice that the reduction does not yield unchanged bounds for data complexity if databases are permitted to vary their programs in such a way as to use an unbounded set of negated base predicates. In this situation the query Φ' varies with the database, so we cannot lift bounds for data complexity, since these require that the query be fixed. Let us now analyze the structure of the glue-types of the queries Φ' in order to calculate bounds on data complexity in this situation.

It is not difficult to see that for each k-ary defined predicate R it suffices to consider Φ'-types K of expansions in $L(\Sigma', R)$, which can be written in the form $G \circ H$ where G is a Φ–type in $L(\Sigma', R)$ and H is a k-graph all of whose edges are of the type P or $not-P$, and which has no internal vertices. Intuitively, H bears the information necessary to determine whether the result of gluing the graph K to another graph is "inconsistent", in the sense that it satisfies one of the queries (4.1). Notice that if two graphs K and K' have the property that $K \circ K'$ satisfies $\exists \mathbf{x}(P(\mathbf{x}) \wedge not-P(\mathbf{x}))$, whereas neither K nor K' satisfies this query individually, then there exist in K and K' two edges, all of whose vertices are sources, but one of which is labelled P and one of which is labelled $not-P$. Thus, the size of glue types of Φ' will be the sum of the size of the glue types of Φ and the size of the graphs H.

To determine the size of the graphs H, notice that if P is a predicate of arity n then we may have up to k^n different assignments of the n arguments to the k sources. Thus, H may contain up to $|\Sigma|.k^n$ edges. Together with the bounds from Proposition 3.4.6 this is already enough to show that if the arity of *basic* predicates is bounded then glue types will be of size polynomial in the size of Σ. However, we can do somewhat better than this by noting that we require only glue types for expansions in $L(\Sigma', R)$. Recall from Section 3.5 that we may obtain the set of glue types of $L(\Sigma', R)$ as the fixpoint of a procedure which constructs new glue types by substituting previously computed glue types into the body of rules. This procedure cannot generate all the possible assignments of the n arguments to the sources. Specifically, we obtain from the rules an initial set of possible assignments of the n arguments to k sources. All assignments generated at later stages of the fixpoint computation are the result of composing one of these 'basic' assignments with a mapping from k sources to k sources. Thus, in fact the number of possible assignments that may be generated is bounded by $|\Sigma|.k^k$. It follows from this that if the arity of *defined* predicates is bounded then we also have polynomial size glue types for Φ'. Thus, we have shown that a bound on the arity of either basic or defined predicates suffices to yield glue types of polynomial size. A direct application of the algorithms of Section 3.5 now results in the following bounds on data complexity.

Theorem 4.3.3: The data complexity of basic queries on databases whose programs contain negated basic predicates and have bounded arity is in PSPACE for linear programs and in EXPTIME for non-linear programs.

On the other hand, if we relax the constraint that the arity of either defined or basic predicates be predicates be bounded then we can only say that glue types are of size exponential in the size of the program. We obtain an increase in the upper bound on data complexity, as stated by the following result.

Theorem 4.3.4: The data complexity of basic queries on databases whose programs contain negated basic predicates and have unbounded arity is in EXPSPACE for linear programs and in 2–EXPTIME for non-linear programs.

We now show that the upper bounds on data complexity stated in Theorem 4.3.3 and Theorem 4.3.4 are tight. Notice that these results seem to indicate a jump in complexity in the passage from bounded arity programs to unbounded arity programs. The lower bounds will establish that this increase in complexity is real. We begin by considering the bounded case.

Theorem 4.3.5: There exists a query Φ which on databases with negated basic predicates has PSPACE complete data complexity for binary linear programs, and EXPTIME complete data complexity for binary non-linear programs.

Proof: We will establish the result only in the linear case, by showing how to simulate polynomial space bounded computations with linear rules. It is straightforward to modify our arguments to yield the EXPTIME lower bound for non-linear rules by using the non-linearity to simulate alternating polynomial space bounded computations instead.

Suppose we wish to simulate a Turing machine computation which uses N tape cells. As usual, we consider a representation of configurations in which cell contents record information about the head and state of the machine. Let $a_1 \ldots a_K$ be the possible contents of cells. Thus, we have a relation $Comp \subseteq \{1, \ldots, K\}^6$ with the property that if three contiguous cells of a configuration contain the symbols $a_{j_1}, a_{j_2}, a_{j_3}$ then the next configuration will have symbols $a_{j'_1}, a_{j'_2}, a_{j'_3}$ in the corresponding cells just when $(j_1, j_2, j_3, j'_1, j'_2, j'_3) \in Comp$. Introduce the monadic basic predicates $P_{i,j}$ where $1 \leq i \leq N$ and $1 \leq j \leq K$. Intuitively, $P_{i,j}(x)$ will represent the fact that x is a Turing machine configuration in which cell i contains the symbol a_j. For $1 \leq i \leq N$ and $1 \leq j \leq K$ we will use the notation $\pi_{i,j}(x)$ for the conjunction of the atom $P_{i,j}(x)$ with the atoms $\neg P_{i,j'}(x)$ for all $j' \neq j$. Intuitively, this conjunction asserts that a_j is the contents of the j-th cell, together with the negative information that this excludes the possibility that any other symbol inhabits that cell.

If the initial configuration of the machine is $a_{i_1} \ldots a_{i_N}$ then this is represented in the database by the facts $\pi_{j,i_j}(start)$ for $j = 1 \ldots N$. The database also contains the defined fact **next**$(start)$, from which the next configuration will be generated. The rules of the program are as follows. We have for each i in the range 2 to $N-1$ a defined

predicate $\mathbf{C}_i(x, y)$. Intuitively, these predicates will take a configuration x and make a contribution to the next configuration y by guessing a symbols for the cells $i-1, i, i+1$ of that configuration. This configuration construction is started off by the rule

$$\mathbf{next}(x) : -\mathbf{C}_2(x, y)$$

which "creates" a new configuration y and "issues the instruction" to guess the contents of the first three cells of that configuration. Of course, random guessing of cell contents will not do to simulate the computation. Instead, what we do is guess the contents of the i-th cell and the two adjacent cells in the configuration x, and compute the contents of the corresponding cells in the configuration y from these. To verify our guess about x we simply assert what we believe to be the contents of these cells. If our guess about the cells in x was incorrect, this will generate an inconsistency. Thus, in fact only one of the possible rules may be consistently used. Formally, we have the rules

$$\mathbf{C}_i(x, y) : -\pi_{i-1,j_1}(x), \pi_{i,j_2}(x), \pi_{i+1,j_3}(x), \pi_{i-1,j'_1}(y), \pi_{i,j'_2}(x), \pi_{i+1,j'_3}(x), \mathbf{C}_{i+1}(x, y)$$

for each sex-tuple $(j_1, j_2, j_3, j'_1, j'_2, j'_3)$ in $COMP$ and i in the range 2 to $N-2$. The construction of the configuration y is completed by one of the rules

$$\mathbf{C}_{N-1}(x, y) : -\pi_{N-2,j_1}(x), \pi_{N-1,j_2}(x), \pi_{N,j_3}(x), \pi_{N-2,j'_1}(y), \pi_{N-1,j'_2}(x), \pi_{N,j'_3}(x), \mathbf{next}(y)$$

provided the symbol $a_{j'_3}$ does not indicate a halting state, or by the rule

$$\mathbf{C}_{N-1}(x, y) : -\pi_{N-2,j_1}(x), \pi_{N-1,j_2}(x), \pi_{N,j_3}(x), \pi_{N-2,j'_1}(y), \pi_{N-1,j'_2}(x), \pi_{N,j'_3}(x).$$

in which $a_{j'_3}$ is a halting state. (Without loss of generality the machine always halts at the end of the tape.) It is straightforward to verify that the database just described is consistent just in case there is a halting computation on the input. Thus, the database entails *False* if and only if the computation does not halt. □

The next hardness result will show that predicates of unbounded arity result in higher complexity. Recall from the discussion above that glue types are of polynomial size if either basic or defined predicates have bounded arity. Thus, the proof to follow necessarily requires both basic and defined predicates of unbounded arity.

Theorem 4.3.6: There exist queries Φ which on databases with negated basic predicates have EXPSPACE complete data complexity for linear programs of unbounded arity, and 2–EXPTIME complete data complexity for non-linear programs of unbounded arity.

Proof: Again we prove this for the linear case only. We will reuse an idea from the proof of Theorem 4.3.5: we construct the next configuration of the computation by guessing the contents of cells of the current configuration, and using negation to verify the guess. However, we need a different representation of configurations, since we must now represent an exponential amount of space. As before, let $a_1 \ldots a_K$ be the possible contents of cells and let $Comp$ be the six-ary relation describing the transitions of the Turing machine. Suppose we wish to represent a computation using space N.

Rather than use a distinct predicate to represent the contents of each cell, we now have an M-ary basic predicate P_i for each symbol a_i. Similarly to the above proof, the notation $\pi_i(\mathbf{x})$ will be the conjunction of $P_i(\mathbf{x})$ with the $\neg P_j(\mathbf{x})$ for $j \neq i$, where $\mathbf{x} = x_1 \ldots x_M$ is a sequence of M variables. We also have a set of predicates Q_i for $i = 1 \ldots M$, and write $\lambda(\mathbf{x})$ for the conjunction of the facts $Q_i(x_i)$ for $1 \leq i \leq M$, with the facts $\neg Q_i(x_j)$ for all $1 \leq i, j \leq M$ with $i \neq j$. The purpose of the predicates Q_i is to keep track of the initial cell of configurations. This will be explained below.

We will represent cell numbers using the pattern of variables of the predicates P_i. Formally, consider the sequence $2, 3, 5, 7, 11, \ldots, p$ consisting of the first K primes. The number M will be the sum of these primes. Let θ be the permutation of the set $\{1 \ldots M\}$ composed of the cycles $\{1,2\}, \{3,4,5\}, \{6,7,8,9,10\} \ldots$ in which the i-th cycle has length equal to the i-th prime. If $\mathbf{x}$ is the sequence of variables $x_1, \ldots, x_M$, the n-th cell of the configuration will be represented by the pattern of variables

$$x_{\theta^n(1)}, x_{\theta^n(2)}, \ldots, x_{\theta^n(M)}.$$

For brevity we will write this simply as $\theta^n(\mathbf{x})$. A configuration $a_{i_1} \ldots a_{i_N}$ is now represented as the conjunction

$$\pi_{i_1}\left(\theta^0(\mathbf{x})\right), \pi_{i_2}\left(\theta^1(\mathbf{x})\right), \ldots, \pi_{i_N}\left(\theta^{N-1}(\mathbf{x})\right). \tag{4.2}$$

For example, if $M = 5$ and $\mathbf{x} = x_1x_2x_3x_4x_5$ then the configuration a_3a_2 is represented by the conjunction

$$P_3(x_1x_2x_3x_4x_5) \wedge \bigwedge_{i\neq 3} \neg P_i(x_1x_2x_3x_4x_5) \wedge$$
$$P_1(x_2x_1x_4x_5x_3) \wedge \bigwedge_{i\neq 1} \neg P_i(x_2x_1x_4x_5x_3)$$

In order for all elements of the list (4.2) to be distinct, we take to N be equal to the degree of the permutation, that is, the least number n for which θ^n is the identity permutation. Since the lengths of the cycles are relatively prime the degree is $N = 2.3.5 \ldots p \geq 2^K$ where K is the number of primes in the product.

Suppose we are given a starting configuration $a_{i_1} \ldots a_{i_n}$. We initialize the computation by the basic facts

$$\pi_{i_1}\left(\theta^0(\mathbf{c})\right), \pi_{i_2}\left(\theta^1(\mathbf{c})\right), \ldots, \pi_{i_n}\left(\theta^{n-1}(\mathbf{c})\right)$$

together with the basic facts $\lambda(\mathbf{c})$, where $\mathbf{c}$ is a vector of M distinct constants. We have one defined fact, namely $\mathbf{blanks}(\theta^n(\mathbf{c}))$. Intuitively, this defined fact is used to fill the rest of the initial configuration with the blank symbol. Thus, we have a rule

$$\mathbf{blanks}(\mathbf{x}) : -\pi_b(\mathbf{x}), \mathbf{blanks}(\theta(\mathbf{x}))$$

which lays down one blank symbol and moves to the next cell of the configuration. Here we assume that a_b is the symbol representing a blank cell not containing the head. The laying down of blanks is stopped by the rule

$$\mathbf{blanks}(\mathbf{x}) : -\pi_b(\mathbf{x}), \mathbf{next}(\theta(\mathbf{x}))$$

which lays down the last blank and issues the command to start the calculation of the next configuration. Of course we need some way to ensure that exactly the right number of blanks is laid down. This is achieved partially by the fact that if the permutation cycles the first of these two rules will assert $\pi_b(\mathbf{c})$, which will contradict with the $\pi_{i_1}(\mathbf{c})$ already part of the expansion. However, we also need to guarantee that we have a sufficient number of blanks to make up a full configuration. This is taken care of by the predicate **next**, which verifies that it has not been called too early. Specifically, we

have the rule

$$\mathbf{next}(\mathbf{x}) : -\lambda(\mathbf{x}), \lambda(\mathbf{y}), \mathbf{C}(\theta(\mathbf{x}), \theta(\mathbf{y}))$$

which checks that its input sequence of variables corresponds to the first cell of a configuration by asserting $\lambda(\mathbf{x})$: if the input is different from the first cell, this will contradict the numbering of the variables $\mathbf{x}$ by the predicates Q_i already part of the configuration.

The predicate $\mathbf{C}$ computes the configuration $\mathbf{y}$ from the configuration $\mathbf{x}$ three cells at a time, exactly as in the previous proof. As before, we guess the contents of the three cells of $\mathbf{x}$, and verify the guess by generating a contradiction if it is incorrect. Thus, we have a rule

$$\mathbf{C}(\mathbf{x}, \mathbf{y}) : -\pi_{j_1}\left(\theta^{-1}(\mathbf{x})\right), \pi_{j_2}(\mathbf{x}), \pi_{j_3}(\theta(\mathbf{x})), \pi_{j_1'}\left(\theta^{-1}(\mathbf{y})\right), \pi_{j_2'}(\mathbf{y}), \pi_{j_3'}(\theta(\mathbf{y})), \mathbf{C}(\theta(\mathbf{x}), \theta(\mathbf{y}))$$

for each sex-tuple $(j_1, j_2, j_3, j_1', j_2', j_3')$ in the relation $Comp$. We exit from the construction of the configuration $\mathbf{y}$ via the rule

$$\mathbf{C}(\mathbf{x}, \mathbf{y}) : -\mathbf{next}(\theta(\mathbf{y}))$$

It is intended that this rule call the predicate $Next$ with a pattern of variables corresponding to the initial cell of configuration $\mathbf{y}$: notice that if this is the case then the last cell of $\mathbf{y}$ has just been assigned a symbol. Verification that the call has been made at the right time is done by the predicate $Next$, as we have already remarked above. It may still be possible for configuration construction to proceed too far and start to cycle, but this can be prevented by placing end-markers in the first and last cell. Finally, the computation is terminated by the rules

$$\mathbf{C}(\mathbf{x}, \mathbf{y}) : -\pi_{j_1}\left(\theta^{-1}(\mathbf{x})\right), \pi_{j_2}(\mathbf{x}), \pi_{j_3}(\theta(\mathbf{x})), \pi_{j_1'}\left(\theta^{-1}(\mathbf{y})\right), \pi_{j_2'}(\mathbf{y}), \pi_{j_3'}(\theta(\mathbf{y}))$$

for which $(j_1, j_2, j_3, j_1', j_2', j_3')$ is in $Comp$ and either $a_{j_1'}, a_{j_2'}$ or $a_{j_3'}$ is a symbol representing a final state of the Turing machine. Exactly as in the result above, we have that the database just described is consistent if and only if the computation terminates.

It remains only to verify that the database corresponding to an input of size n has size polynomial in n. Recall that N is the amount of space the computation may use,

and the arity M of the predicates is the sum of the first K primes. We have seen that $N \geq 2^K$, so $K = p(n)$ suffices to represent computations using space $2^{p(n)}$. Hadamard's theorem [50] states that the K-th prime has size less than $K.log\ K$. Thus, we have $M \leq (p(n).log\ p(n))^2$. It follows from this that the database described has polynomial size. □

4.4 Negating Defined Predicates

The previous section has shown that even a limited amount of negation in programs leads to a dramatic increase in the complexity of query processing. In this section we study the effect of a further liberalization of the way in which negation occurs in the database: we now permit defined predicates to be negated. Unfortunately, this rapidly leads to undecidability. Nevertheless, we will be able to show that decidable generalizations of the class of databases of the previous section do exist.

It turns out that the reduction used in the previous section to show decidability for negated basic predicates works for a certain class of negated defined predicates also. Now we introduce for each negated predicate P, basic or defined, a new *basic* predicate $not{-}P$. Given a program Σ containing negation, we let Σ' be the program without negation obtained by replacing each occurrence of $\neg P(\mathbf{x})$ by $not{-}P(\mathbf{x})$. As in the previous construction, the query Φ is replaced by the query obtained by disjoining $\exists \mathbf{x}[P(\mathbf{x}) \wedge not{-}P(\mathbf{x})]$ for each predicate which occurs negated in the program.

Theorem 4.4.1: If Σ is a program with no more than two strata, and Φ is a basic query then $\Sigma, D \models \Phi$ if and only if $\Sigma', D \models \Phi'$.

Proof: Suppose $\Sigma', D \not\models \Phi'$. The program Σ' contains no negated atoms. Thus, we may apply Lemma 3.3.7, which yields that there exists an expansion E of D by Σ' for which the structure $(\Sigma')^\infty(E)$, obtained by computing the least fixpoint of Σ' over E, does not satisfy the query Φ'. Let M be the set of basic atoms which hold in this structure, excluding those of the form $not{-}P(\mathbf{c})$. We show that the model $\Sigma^\infty(M)$ supports D but not the query Φ. First, notice that for predicates in the first stratum, the programs Σ and Σ' contain an identical set of rules. Thus, the fixpoints

are identical with respect to these predicates. Since the query Φ' is not satisfied in $(\Sigma')^\infty(E)$, whenever E contains an atom of the form $not\text{-}P(\mathbf{c})$, it is in fact the case that $P(\mathbf{c})$ is not in $(\Sigma')^\infty(E)$, and hence it is not in $\Sigma^\infty(M)$ either. It follows from this that for predicates P in the second stratum of Σ, if the atom $P(\mathbf{c})$ is in $(\Sigma')^\infty(E)$ then it is in $\Sigma^\infty(M)$ also. This implies that $\Sigma^\infty(M)$ supports D. To see that this a countermodel to Φ we use the fact that the query contains only positive basic atoms, and these hold in M just when they hold in E. This completes the proof that $\Sigma, D \models \Phi$ implies $\Sigma', D \models \Phi'$.

For the converse, suppose that $\Sigma, D \not\models \Phi$. Then there exists a set M of basic facts such that $\Sigma^\infty(M) \not\models D$ but $\Sigma^\infty(M) \models \Phi$. In particular, there exists an expansion E of D by Σ' and a mapping h from the constants of E to those of M with the property that

1. for every positive basic atom $P(\mathbf{c})$ in E the atom $P(h(\mathbf{c}))$ is in M, and

2. for every atom $not\text{-}P(\mathbf{c})$ in E we have $P(h(\mathbf{c})) \notin \Sigma^\infty(M)$.

It follows from the first condition that $E \not\models \Phi$. Suppose that we had $(\Sigma')^\infty(E) \models P(\mathbf{c}) \wedge not\text{-}P(\mathbf{c})$ for some predicate P. This predicate must be defined in the first stratum, so its definition involves no negation. Since h is a homomorphism on the "positive" atoms, it follows that $P(h(\mathbf{c})) \in \Sigma^\infty(M)$ and from the second condition that $P(h(\mathbf{c})) \notin \Sigma^\infty(M)$, a contradiction. Thus, $(\Sigma')^\infty(E)$ does not satisfy any of the consistency checking disjuncts either. This shows that $\Sigma', D \not\models \Phi'$. $\Box$

Theorem 4.4.1 has previously been established by Courcelle [24] in the restricted case of programs in which only basic predicates appear negated, but he apparently did not notice that it applies more generally. Clearly one effect of this reduction is to place defined predicates in the query. We have seen that in general this leads to undecidability. Indeed, take some program Σ (without negation) and query Φ for which $\Sigma, D \models \Phi$ is undecidable. Extend the program to a program Σ' by adding the rules $R :- \Phi$ and $S :- \neg R$. Then it straightforward to show that $\Sigma', D \cup \{S\}$ is inconsistent just in case $\Sigma, D \models \Phi$. Thus, by Proposition 3.3.8, the query problem for programs in which binary recursive predicates are negated is undecidable.

However, we retain decidability for classes of defined predicates which may decidably occur in the query. Thus, we obtain as a corollary of Theorem 4.4.1 that non-recursive predicates and monadic predicates from the first stratum of the program may be negated while retaining decidability of basic queries. We note that the reduction does not work if we permit defined predicates in the query Φ, as the following example shows.

Example 4.4.2: Consider the program Σ containing a single rule $Q :- \neg P$ in which P is taken to be a defined predicate in the first stratum and Q is a defined predicate in the second stratum. (That is, P has a vacuous definition.) Take D to be the empty database. Then $\Sigma, D \models Q$ since the fixpoint computed never contains P. On the other hand, Σ' consists of the rule $Q :- not\text{-}P$. Since D is empty it has a single expansion E by this program, equal to the empty set. The set $(\Sigma')^{\infty}(E)$ is therefore also empty, so does not satisfy $Q \vee [P \wedge not\text{-}P]$. □

We do not know if it is possible to modify the reduction so as to work for queries containing defined predicates. However, we can show that it is not possible to extend the result by increasing the number of strata permitted in the program.

Proposition 4.4.3: The problem $\Sigma, D \models \Phi$ is undecidable for basic queries if Σ ranges over nonrecursive programs with three strata.

Proof: Programs with three strata can represent universal quantification. We use this fact to give a reduction from the implication problem for template dependencies, which is known to be undecidable [127, 46]. Template dependencies are first order formulae of the form

$$\forall \mathbf{x} \exists \mathbf{y} [B(\mathbf{x}, \mathbf{y}) \Rightarrow P(\mathbf{x}, \mathbf{y})] \tag{4.3}$$

in which B is a conjunction of positive literals and P is a positive literal. The implication problem is to determine for a template dependency Σ if $\Delta \models \sigma$, where Δ is a set of template dependencies. Let us first note that (4.3) can be written in the form

$$\forall \mathbf{x} [\forall \mathbf{y} B(\mathbf{x}, \mathbf{y}) \Rightarrow \exists \mathbf{y} P(\mathbf{x}, \mathbf{y})]$$

The deduction theorem now enables us to write the implication problem in the form $\Delta \models \exists \mathbf{y} P(\mathbf{y})$ where Δ is a set of dependencies. Programs with three strata may be

used to represent the set Δ as follows. Given a dependency (4.3) we introduce defined predicates $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$ and the following rules

$$\mathbf{a}(\mathbf{x}) : -B(\mathbf{x}, \mathbf{y})$$
$$\mathbf{b}(\mathbf{x}) : -P(\mathbf{x}, \mathbf{y})$$
$$\mathbf{c} : -\mathbf{a}(\mathbf{x}) \wedge \neg \mathbf{b}(\mathbf{x})$$
$$\mathbf{d} : -\neg \mathbf{c}$$

It is not difficult to see that the defined atom $\mathbf{d}$ now expresses the dependency. Thus, the implication problem may be reduced to querying databases with three strata by taking the program to consist of a collection of such rules, and the database to contain the facts $\mathbf{d}$. □

4.5 Inequality

In this section we consider the effect of including inequality in our queries. The results will be entirely negative: as soon as we admit inequality into the query even linear monadic programs suffice to yield undecidability. Thus, there exists no interesting combination of recursion and inequality with a decidable query problem.

We begin by noting that, already in the context of definite databases, admitting inequality in queries leads to increased data complexity. Thus, the negative results of this section regarding the combination of inequality and recursion are not entirely unexpected. Inequality constraints on null values in relational databases have been studied by [123, 45, 1]. The following example illustrates the high complexity of queries in this context, even when the database does not contain inequality.

Example 4.5.1: Suppose that $G = (V, E)$ is an undirected graph with vertices V and edges E. A colouring of G is just a mapping of G to some set of colours: the graph is said to be k-colourable if there exists a colouring using k colours in which no two adjacent vertices have the same colour. An alternate representation of a colouring of the graph is obtained by identifying vertices which have the same colour. If no two adjacent vertices have the same colour this results in a graph with no loops, that is, edges from a vertex to itself. Thus, a graph is k-colourable

just when it is possible to make a set of identifications of vertices which results in a graph with k vertices and no loops. Conversely, if every reduced graph without loops has more than k vertices, then the graph is not k-colourable.

We may use this to simulate graph non-3-colourability as follows. The set of constants occurring in the database is the set V of vertices. For each vertex $v \in V$, we include in the database the atom $vertex(v)$. For every edge (v, w), we include in the database the atom $edge(v, w)$. Identification of vertices will correspond simply to two constants having the same denotation in some model of the database. Using inequality, one may write a conjunctive query Φ which expresses "there exist at least four vertices". Consider the query $\Phi \vee \exists x[edge(x, x)]$ By the discussion above, this query is entailed by the database just in case the graph G is not three-colourable. Since three-colourability is an NP complete problem, this shows that queries containing inequality have co-NP complete data complexity on relational databases, provided these are not subject to the unique names assumption. □

Moving to the recursive case, let us consider what happens to the notion of glue type in the presence of inequality. First, we need to define the appropriate generalization of glue type. Prior to the introduction of inequality, we had a one-to-one correspondence between flat databases and graphs, or models. As we have seen, a flat database must now be taken to correspond not to a single model, but to a set of models, obtained by identifying some of the constants, subject to the inequality constraints in the database. However, we may still define the appropriate notions over such databases.

Definition 4.5.2: A k-database consists of a flat database D containing inequality, together with an ordered sequence of length k of constants of the database (the *sources*). If D_1 and D_2 are two k-databases, the result $D_1 \circ D_2$ of *gluing* these databases together is the database obtained from the following operations. First, we take the disjoint union of D_1 and D_2. That is, we rename the constants of these databases so that no constant appears in both, and then take the union. Next we identify the i-th source of D_1 with the i-th source of D_2, for $i = 1 \ldots k$.

Now say that two k-databases have the same *glue type* with respect to a query

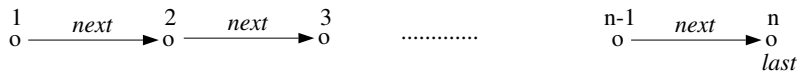

Figure 4.1: The Database D_n

Φ if for all k-databases D we have $D_1 \circ D \models \Phi$ if and only if $D_2 \circ D \models \Phi$.

Note that the relation $\models$ in this definition refers to consequence over all models of the database, whereas in our earlier definition of glue type this was consequence in a single model, since we could work with a unique "minimal" model. The fact that each basic query has a finite number of glue types underlies the decidability results of Chapter 3. The following example shows that queries containing inequality no longer have a finite number of glue types.

Example 4.5.3: Consider the 1–databases D_n for $n \in \mathbf{N}$ depicted in Figure 4.1. Here there are n vertices, and the single source is the vertex 1. Let Φ be the query

$$\exists xyz[next(x,y) \wedge next(x,z) \wedge y \neq z] \vee \exists xy[last(x) \wedge next(x,y)].$$

We show that $D_n \circ D_m \models \Phi$ if and only if $n \neq m$. This implies that Φ has the infinite set $D_1, D_2, D_3, \ldots$ of glue types, since if $n \neq m$ then $D_n \circ D_n \not\models \Phi$, whereas we do have $D_m \circ D_n \models \Phi$.

First note that $D_n \circ D_n$ has a model which is isomorphic to D_n, obtained simply by identifying the corresponding vertices. Since D_n, considered as a model, does not support Φ, we have that $D_n \circ D_n \not\models \Phi$. Suppose now that n and m are different, and, without loss of generality, that n is the smaller. Write $c_1, \ldots, c_n$ for the constants of D_n, and $d_1, \ldots, d_m$ for the constants of D_m, and consider the models of $D_m \circ D_n$. All such models have $c_1 = d_1$ by the identification of the sources. Consequently, a model satisfies the first disjunct of Φ unless also $c_2 = d_2$. Continuing this argument, we eventually obtain that $c_n = d_n$. But then we have a model in which the second disjunct holds. This shows that, in fact, the query Φ holds in all models of $D_m \circ D_n$. $\Box$

This example shows that we cannot expect to generalize the decidability results of the inequality free case once we allow inequality in queries. (Notice that the example

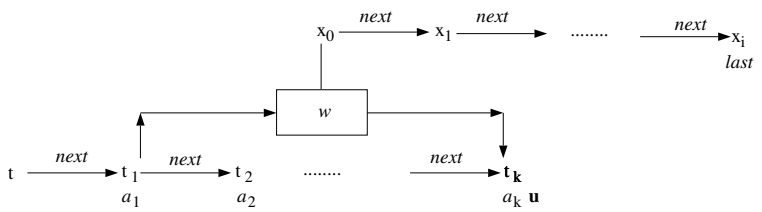

Figure 4.2: The rule body corresponding to a word

did not require inequality in the database.) We now show that under very restricted circumstances, the combination of recursion in the database and inequality in queries is undecidable. For the proof we require a version of the Post Correspondence Problem. Suppose we have a fixed collection W consisting of words $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ over some alphabet. Call the Post Correspondence problem for W, the problem of determining, given as input a word x, whether there exists a number K and indices $i_1, i_2, \ldots, i_K$ such that

$$u_{i_1} u_{i_2} \ldots u_{i_K} = x\, v_{i_1} v_{i_2} \ldots v_{i_K}.$$

There exists a collection W of words for which this problem is undecidable. This follows from the proof that the usual formulation of the Post Correspondence Problem is undecidable (see for example [55]) and the existence of a universal Turing machine.

Theorem 4.5.4: There exists a fixed linear monadic program Σ and a query Φ containing $\neq$ such that it is undecidable to determine for databases D whether $\Sigma, D \models \Phi$.

Proof: The proof uses a set W of words for which the Post Correspondence Problem is undecidable. The idea is to use the program to generate linear sequences corresponding to $u_{i_1} u_{i_2} \ldots u_{i_K}$ and $v_{i_1} v_{i_2} \ldots v_{i_K}$. The database D will contain the input x. We will have that the problem for x has no solution just when $\Sigma, D \models \Phi$.

This is done as follows. For each letter a of the alphabet of W we have a monadic basic predicate $a(t)$. The word $x = a_1 \ldots a_k$ is represented by a sequence of constants $t_1 \ldots t_k$ together with the basic facts $a_i(t_i)$ and $next(t_i, t_{i+1})$ for each i. For technical reasons to be explained shortly we also have the basic facts $w(c_0, t_1, d), w(c_1, t_k, d)$. The two sequences of words from W are generated from the intensional facts $u(t_1), v(t_k)$.

The two intensional predicates $\mathbf{u}, \mathbf{v}$ are defined by the program as follows. For each

word $u_i = a_1 \ldots a_k$ we have a rule

$$\mathbf{u}(t) :- w(t_1, t_k, x_0), next(x_0, x_1), \ldots, next(x_{i-1}, x_i), last(x_i),$$
$$next(t, t_1), a_1(t_1), next(t_1, t_2), a_2(t_2), \ldots, next(t_{k-1}, t_k), a_k(t_k), \mathbf{u}(t_k).$$

See Figure 4.2. Each word v_i corresponds to a similar rule for the intensional predicate $\mathbf{v}$. Intuitively these rules construct a sequence $t_1 \ldots t_k$ corresponding one of the words u_i, v_i, then make a recursive call for the next choice. The predicate w is used to mark out portions of the total sequence constructed by a number of such successive calls as corresponding to a word in W. This is done by the first two arguments of w. The third argument of w contains a pointer to a sequence $x_0 x_1 \ldots x_i$ whose length is the index i of the word u_i or v_i selected. We also need some rules to terminate the choices of words. This is done by the rule $\mathbf{u}(t) :- last(t)$. There is a similar rule for the predicate $\mathbf{v}$.

The problem we now need to deal with is that the sequences generated from the facts $\mathbf{u}(t_1), \mathbf{v}(t_k)$ are totally independent: there need be no correspondence between the length of these sequences, nor between the indices of corresponding choices for u and v. For this we will use a number of disjuncts to "align" the two sequences. Consider the expansions of the atoms $\mathbf{u}(t_1), \mathbf{v}(t_k)$ corresponding to the sequences $u_{i_1} u_{i_2} \ldots u_{i_K}$ and $v_{j_1} v_{j_2} \ldots v_{j_L}$. We ensure that the sequences of t_i generated are aligned, and have the same length, by using the disjunction

$$\exists xyz[next(x, y) \wedge next(x, z) \wedge y \neq z] \vee \exists xy[last(x) \wedge next(x, y)] \tag{4.4}$$

exactly as in Example 4.5.3. This formula is satisfied unless the vertex corresponding to the i-th letter of the word $u_{i_1} u_{i_2} \ldots u_{i_K}$ is equal to the vertex corresponding to the i-th letter of the word $x v_{j_1} v_{j_2} \ldots v_{j_L}$. Next, in order to make the sequence of indices correspond we use the query

$$\exists xyz t_0 t_1 t_2 t_3 t'_0 t'_1 t'_2 t'_3$$
$$[w(t_0, t_1, x) \wedge next(t_1, t_2) \wedge w(t_2, t_3, y) \wedge$$
$$w(t'_0, t'_1, x) \wedge next(t'_1, t'_2) \wedge w(t'_2, t'_3, z) \wedge y \neq z]$$

depicted in Figure 4.3.

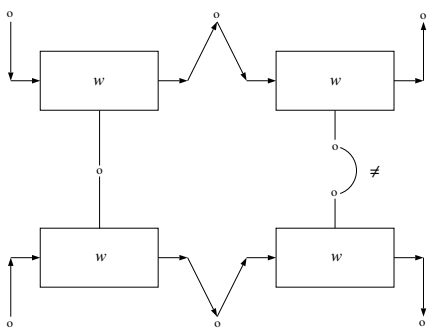

Figure 4.3: Aligning the indices

Recall that the database contains the atoms $w(c_0, t_1, d)$ and $w(c_1, t_k, d)$. If $w(t'_2, t'_k, x)$ and $w\left(t''_{k+1}, t''_{k+l}, y\right)$ are the atoms generated by the bodies corresponding to u_{i_1} and v_{j_1}, respectively, then the above query will be true unless $x = y$. Continuing this argument, we find that the query is true unless all the corresponding vertices in the third argument of w are equal. We still need to ensure that the sequences $i_1 \ldots i_K$ and $j_1 \ldots j_L$ are of the same length. This is done by the query

$$\begin{aligned}
&\exists x\, y t_0 t_1 t'_0 t'_1 t'_2 t'_3 \\
&\quad [w(t_0, t_1, x) \wedge last(t_1) \wedge \\
&\quad\ w(t'_0, t'_1, x) \wedge next(t'_1, t'_2) \wedge w(t'_2, t'_3, z)]
\end{aligned}$$

which detects that one of the sequences has terminated while the other is proceeding. Finally, we need to ensure that $i_k = j_k$ for each k. This is already achieved by the disjuncts (4.4), once the appropriate third arguments of w have been identified. □

This result suggests that there is no interesting class of recursive definitions with respect to which queries containing $\neq$ are decidable. The linear monadic rules are a very restricted class. There does not appear to be any interesting additional restriction on rules that would help to recapture decidability.

Nevertheless, we shall establish a result in Chapter 6 that implies that there is a class of rules for which one obtains decidability of certain basic queries containing $\neq$. This decidability result involves the following observation. Expanding recursive rules generates a large number of "skolem" constants, obtained from the existential quantifiers in the body of rules. The proof of Theorem 4.5.4 exploits the fact that any of these constants is potentially equal to any other of these constants. As we will see,

when dealing with linearly ordered domains, the presence of constraints in rules may restrict these potential equalities in such a way as to defeat the undecidability result. We use this observation to show that with respect to a certain class of definitions, the linear order constraint $<$ may occur in basic queries without loss of decidability. This result implies that $\neq$ also may occur, since over linearly ordered domains $x \neq y \equiv x<y \vee y<x$. First, however, we study in the next chapter the effect of linear order in databases not containing defined relations.

Chapter 5

Linear Order

5.1 Introduction

We now turn to a consideration of a second type of indefinite information. Databases and queries in this chapter[1] will be formulae in a first order language with two sorts: the usual 'object' sort Ob, together with an 'order' sort Ord, whose elements may participate in order relations. Thus we require that the arguments of predicates be typed, and that the occurrence of constants and variables in atoms respect the typing. Atomic formulae will be either

1. *proper atoms*, of the form $P(x_1, \ldots, x_n)$, where the x_i are constants or variables of the appropriate sort, or

2. *order atoms*, of the form $u < v$, where u and v are constants or variables of sort Ord.

An *indefinite order database,* or $\{<\}$-database, will be a set of ground atomic formulae of either type. Order databases will be semantically interpreted in an open world fashion in models in which the relation $<$ denotes a linear order. Queries will be positive existential formulae containing proper atoms and order atoms. Note that all queries in this chapter are *basic,* in the terminology of prior chapters. We insist that queries containing disjunctions be in disjunctive normal form, that is, such queries must be disjunctions of a number of existentially quantified conjunctions.

Although indefinite order databases appear to be merely incomplete, rather than indefinite, it turns out that they are indeed indefinite, according to our characterization.

[1] This chapter is an expanded version of [88].

The following example illustrates how indefiniteness arises from incomplete information about linearly ordered domains.

Example 5.1.1: A highly classified document is discovered to have been leaked during the night from the security compound at the US embassy in Moscow. There are no duplication facilities in the compound: the guilty party must have removed the document, copied it, and then replaced it. Thus the culprit was in the compound at least twice. The security guard's log shows Schultz entering the compound, then leaving. Some time later, North is recorded entering. The guard's watch was broken, so exact times are not recorded. Worse, he confesses to having dozed off frequently during the night, so this is all the information his log shows. He is dishonourably discharged for dereliction of duty. Interrogation of Schultz and North yields the following information: Schultz admits to having been in the compound, and claims that while there, North also came into the compound. Schultz says he left before North did, but does not have a precise recollection of what times he entered and left. As is to be expected, he will not admit to having been in the compound twice. North "takes the Fifth" and refuses to testify. This evidence does not appear to be much to go on, but it is enough to encourage the Internal Affairs officer to start further investigations into the activities of North and Schultz: he has deduced that one of the two was in the compound twice[2].

We may formalize this problem as follows. Let the predicate $IC(u, v, x)$ represent the fact that x was in the compound for a continuous period starting at time u and ending at time v. Then the guard's log may be expressed as

$$IC(z_1, z_2, S), IC(z_3, z_4, N), z_1 < z_2 < z_3 < z_4$$

Here $z_1 \ldots z_4$ are order constants representing unknown points of time and the object constants S, N refer to Schultz and North, respectively. The testimony of Schultz may be represented as

$$IC(u_1, u_3 S), IC(u_2, u_4, N), u_1 < u_2 < u_3 < u_4$$

where again the u_i are unknown order constants.

Suppose this set of facts has been entered in a knowledge base. Since time is a linearly ordered domain, the constraints on the constants z_i, u_i underdetermine the temporal relationships holding between these constants in any model of the data. For

[2]The example is purely fictional. All resemblances to members of the Reagan administration are coincidental.

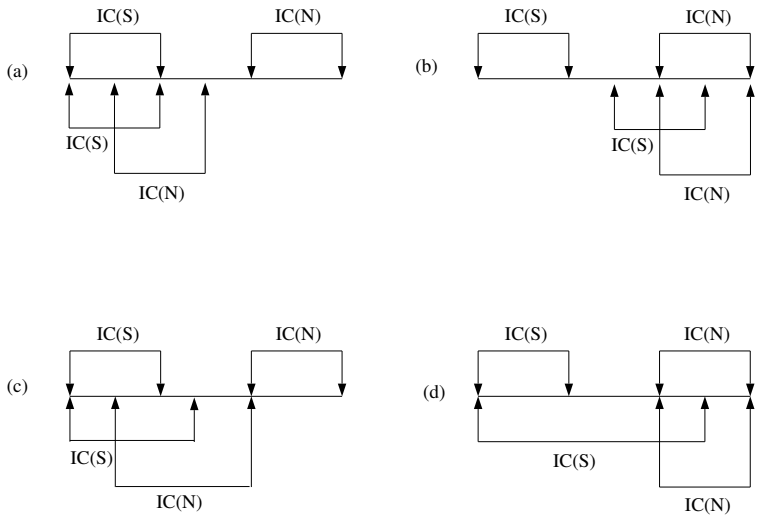

Figure 5.1: Some models of the data

example, we could have any of the relationships $z_1 < u_1$, $z_1 = u_1$, or $z_1 > u_1$ holding in models of the data. Thus, to obtain models of the data it is necessary to "topologically sort" the partial order in the data, adding additional constraints so as to obtain a linear order. Figure 5.1 shows some of the models resulting from this process. Here the top portion of each model derives from the guard's log, the bottom portion derives from Shultz' testimony. Note that distinct order constants may refer to the same point in the linear order, e.g., in model (a) $z_1 = u_1$.

We also need some integrity constraints: for example, the facts mentioned so far have a model (d) in which $z_1 = u_1$ and $z_2 < u_3$, so that we have two overlapping, but not identical intervals representing periods for which Schultz was in the compound. Clearly the intended semantics does not permit this. We need to eliminate models which have such overlapping but not identical intervals. Rather than incorporate such 'negative' information in the database, we will handle this by modifying queries. Thus, let Ψ be the formula

$$\exists x t_1 t_2 t_3 t_4 w [IC(t_1, t_2, x) \wedge IC(t_3, t_4, x) \wedge t_1 < w < t_2 \wedge$$
$$t_3 < w < t_4 \wedge (t_1 < t_3 \vee t_2 < t_4)]$$

which detects the condition we wish to eliminate. The effect of the integrity constraint is then obtained by using a query $\Psi \vee \Phi$ in place of the query Φ. (This particular integrity constraint allows simultaneous departure and reentry.) The investigating officer may now reach his conclusion by noting that the formula $\Phi(x) = \exists t_1 t_2 t_3 t_4 [IC(t_1, t_2, x) \wedge IC(t_3, t_4, x) \wedge t_1 < t_3]$ expresses that x entered the compound at two distinct times t_1, t_3. Thus he may pose the query $\Psi \vee \Phi(S) \vee \Phi(N)$ ("Did either Schultz or North enter the compound twice?") or, more generally, $\Psi \vee \exists x \Phi(x)$ ("Did someone enter the compound twice ?"). We leave it to the reader to verify that both of these queries should be answered "yes". Note however, that the queries $\Psi \vee \Phi(S)$ and $\Psi \vee \Phi(N)$ should both fail (consider models (a) and (b)): there is not yet enough evidence for charges to be laid against either suspect. □

Notice that this example shows that indefinite order databases satisfy neither property P1 nor property P2 from Section 2.1. Thus such databases indeed record indefinite information according to our characterization. As with databases containing defined relations, this indefiniteness may be understood as arising from the background information. In this case, the background information states that the disjunctive formula

$$x < y \vee x = y \vee y < x$$

holds for all points x, y in the linearly ordered domain. It is also necessary for the example to assume that the linear order is dense, that is, that there exists a point between any two distinct points.

Many applications give rise to indefinite data about linear order. As in the example, the linearly ordered domain is often a time line. In the problem of seriation in archeology [63] each type of artifact is assumed to have been in use for some historical interval. Absolute data for these intervals is rarely available, but coincidence of two artifacts in a grave indicates that their intervals overlap. Golumbic [43] describes this and many other examples of indefinite order data in various domains, including behavioural psychology, biology, scheduling problems in operations research, and combinatorics.

Order indefinite data arises in a variety of contexts in Artificial Intelligence. Allen [2] has pointed out that in natural language most temporal reports, rather than give

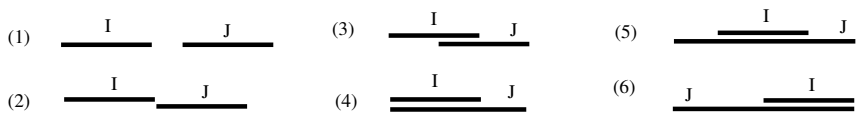

Figure 5.2: Allen's primitive interval relations

absolute times, describe relations between intervals, such as "We found the letter while John was away." To model such qualitative temporal information, he proposes an algebra based on thirteen primitive temporal relations between intervals. Allen's primitive relations are the relations (1) "I is before J," written $I < J$, (2) "I meets J," written ImJ, (3) "I overlaps J," written IoJ, (4) "I starts J," written IsJ, (5) "I is during J," written IdJ, (6) "I finishes J," written IfJ, together with the inverses of these relations (e.g. "I is after J" is the inverse of "I precedes J"), denoted $>, mi, oi, si, di, fi$ respectively, as well as the relation "I equals J," written $I = J$. These relations are shown in Figure 5.2. The interval algebra consists of these thirteen relations, together with all the relations that may be composed as disjunctions of these primitives. For example $I\{<, >\}J$: "I is before J or I is after J," is an element of the interval algebra, expressing non-intersection of the intervals. Allen gives a polynomial time algorithm for making inferences about interval relations, based on a table of transitivity relations. For example, if I is before J and J overlaps K then I is before K. Allen's algorithm repeatedly applies a set of such rules until no new inferences can be made. Unfortunately, this algorithm is incomplete.

An explanation for this is provided by Vilain and Kautz [128] who study the complexity of reasoning in Allen's interval algebra. They show that satisfiability of a set of interval algebra expressions, and the associated "minimal labelling problem," of finding a minimally disjunctive expression in the interval algebra representing all the possible primitive relations between two intervals, are both NP hard. Golumbic and Shamir [44] have recently presented a finer grained analysis of the complexity of these problems, showing the effect on complexity of various restrictions on the set of primitive relations. Even when the number of primitive relations is reduced to three by abstracting Allen's thirteen relations, the complexity remains NP hard.

$I = [u_1, v_1]$	$J = [u_2, v_2]$
I before J	$v_1 < u_2$
I meets J	$v_1 = u_2$
I overlaps J	$u_1 < u_2 < v_1 < v_2$
I begins J	$u_1 = u_2 < v_1 < v_2$
I contained by J	$u_2 < u_1 < v_1 < v_2$
I ends J	$u_2 < u_1 < v_1 = v_2$
I equals J	$u_1 = u_2, v_1 = v_2$

Table 5.1: Allen's primitive relations represented using point relations

As a remedy for the high complexity of deriving interval relations, Vilain and Kautz proposed to restrict the expressiveness of temporal data by using a point based language with relations $\{<, \leq, \neq\}$. They show how this restricted language is still able to represent a subset of Allen's interval algebra. This is done by representing an interval I by its endpoints $[u, v]$, and then asserting point relationships on the u's and v's. All of Allen's primitive relations can be represented using only equality and $<$: see Table 5.1. The problem of deriving point relationships of the form $u(<, \leq, \neq)v$ in the restricted language may be shown to have polynomial time complexity [121, 8].

Another example of indefinite order data in Artificial Intelligence is *nonlinear planning* [113, 120, 15]. Here, rather than the solution to a planning problem being a linear sequence of actions, one deals with partially ordered sets of actions. This allows for greater flexibility in the order of execution of actions, and is also able to express concurrently executable plans. However, it is still necessary to reason about the compatible linear orders, since these correspond to the actual executions of the plan. Allen has proposed to use the interval algebra to formalize planning; we will discuss this proposal in the conclusion of this chapter.

The problem of reasoning with indefinite information about intervals with "duration constraints" has been considered in the AI literature by a number of authors [28, 29, 69, 81]. This work deals with partially ordered networks of points, in which the arcs are labelled with a pair of numbers $[a, b]$ constraining the distance between the initial point u and the endpoint v. These arcs are interpreted as linear inequalities $a \leq v - u \leq b$.

The most comprehensive study of this type of constraint network is [29], which considers three inference problems: finding all feasible times that an event can occur, finding all possible relationships between two events, and generating scenarios consistent with the constraints.

The problem of inferring inequalities from other inequalities has also been studied in other areas of computer science. In the context of database systems this problem has been considered in connection with predicate locking [111]. The more general problem of inferring linear inequalities of the form $ax + by + \ldots \leq p$ from other such inequalities has been considered in the context of applications to constraint logic programming [70, 117].

All of this work, when it considers complexity, concerns problems of inferring linear order relations of one sort or another from data of the same sort. None of these results treat the more general class of queries we study, which may contain predicates and existential quantifiers. We know of only one exception to this rule, and this is the work of Klug [64] on the optimization of relational queries containing inequalities. Klug noted that the classical homomorphism theory [14] for containment of conjunctive queries, which shows that this problem is NP-complete, does not extend to queries containing inequalities, although he was able to show that this theory still applies for the "right semi-interval queries" in which all inequalities are of the form $c(<, \leq)x$ where c is a constant and x is a variable. For the general problem, he was able to provide an upper bound of Π_2^p, but no lower bound. There has since been some related work by Kanellakis et al. [62] which shows that containment of queries containing quadratic equation constraints is Π_2^p-complete, but the complexity of the more restricted problem has remained open.

As we discussed in Example 3.1.7, the problem of testing containment of a query $Q1$ in a query $Q2$ may be solved by asserting the query $Q1$ as a database and evaluating the query $Q2$ in this database. In the case that the query $Q1$ contains inequalities, we obtain an indefinite order database instead of a relational database. Thus, the complexity of containment is exactly the combined complexity of querying order indefinite databases.

Our contribution in this chapter is to provide an analysis of the complexity of querying indefinite order data using positive existential queries for databases and queries in which the only order relation is $<$. In general, one would like to have a more complete characterization of the complexity of querying databases containing various combinations of the relations $\leq, \neq$ and $<$, but we will not do this here[3]. As we shall see, the theory for even the limited case we study is very rich.

We begin with a study of the connections between three different semantics, depending on whether the order is over a finite domain, integer order, or isomorphic to the rationals. We establish reductions that enable us to transfer bounds for the finite case to the other two, showing that these three assumptions on the structure of the order domain are equivalent from the point of view of complexity of positive existential queries.

We then turn to an analysis of complexity of positive existential queries for the finite semantics. This analysis shows that even for the restricted case of databases and queries containing only the relation $<$ the complexity of query answering is intractable. As in earlier chapters, we consider three measures of complexity: data complexity, expression complexity and combined complexity. In their general forms, most of the query problems considered turn out to be (probably) intractable, as indicated by Table 5.2. Here each entry gives a complexity class for which the corresponding query problem is complete, as well as a reference to the proof of the relevant bound. Note that for these unconstrained problems, conjunctive queries have the same complexity as disjunctive queries: in general, the reference for a conjunctive case is to a lower bound result and that for a disjunctive case is for an upper bound result. Once we constrain other parameters of the problems the equivalence in complexity of conjunctive and disjunctive queries no longer holds. One of our results is the lower bound for combined complexity matching Klug's upper bound of Π_2^p. Thus, whereas queries about point relationships are tractable, as discussed above, this tractability is lost once one admits predicates and existential quantification. This intractability persists for data complexity, which

[3]The full analysis will be presented in a forthcoming paper. We point out that databases containing $\neq$ only have already been extensively studied, as we remarked in Chapter 2.

Query Type		data complexity	expression complexity	combined complexity
n-ary predicates $n \geq 2$	conjunctive	co-NP (5.3.2)	NP (5.3.4)	Π_2^p (5.3.3)
	disjunctive	co-NP (5.3.1)	NP (5.3.1)	Π_2^p (5.3.1)
monadic predicates	conjunctive	PTIME (5.4.4)	PTIME	co-NP (5.4.5)
	disjunctive	PTIME (5.5.10)	PTIME (5.5.6)	co-NP (5.5.1)

Table 5.2: Complexity of unconstrained query problems

we show to be co-NP complete.

This indicates that further constraints are required to obtain tractable inference problems. We consider a number of different parameters, and provide a characterization of the classes of problems stated in terms of these parameters that have polynomial time complexity. One of the constraints we consider is severe: it is the restriction that all predicates be monadic. While monadic predicates are insufficiently expressive to represent the interval data required in many applications, this restriction is nevertheless of interest.

Example 5.1.2: In gene alignment in biology one wishes to compare a number of sequences of bases, represented by symbols C, G, A and T, for evidence that they are related. Mutations correspond to insertions and deletions of fragments in such sequences, so it is necessary to permit "gaps" when doing the comparison: see Figure 5.3 for a sample alignment. The space of possible alignments may be represented by an indefinite order database in which C, G, A and T are monadic predicates and a sequence $s_1 s_2 \ldots s_n \in \{C, G, A, T\}^*$ corresponds to the facts

$$s_1(u_1), s_2(u_2), \ldots, s_n(u_n), u_1 < u_2 < \ldots < u_n.$$

By adding integrity constraints as above one may represent various restrictions on the acceptable alignments: for example $\neg \exists t[A(t) \wedge G(t)]$ disallows the alignment of A and G. While we do not wish to claim that the class of queries we consider in this dissertation is biologically realistic, our results on the monadic case may provide an interesting basis for comparison for more general classes of queries. □

$$GCAAGTCGATCGAG$$
$$AC--GTCG-GCGA-$$

Figure 5.3: Aligning two sequences

Certain queries using n-ary predicates also reduce to the monadic case. Suppose, for example, that all predicates have a single "temporal" argument that may participate in order relations. That is, we are interested in facts holding at time points rather than over intervals. Then queries which quantify only over time reduce to the monadic case. For example, consider the query

$$\exists uv[P(a,u) \wedge u<v \wedge P(b,v)]$$

expressing that there exists a time when $P(a)$ holds and a later time when $P(b)$ holds. This query may be answered using algorithms for the monadic case by treating the expressions $P(a,x)$ and $P(b,x)$ as (distinct) monadic predicates on the variable x.

Surprisingly, even the restriction to monadic predicates is not enough to obtain tractable combined complexity: we show that even this problem has co-NP hard complexity. Data complexity with monadic predicates is in PTIME. There is a catch, however. While conjunctive monadic queries may be shown to have PTIME data complexity by a simple "greedy" argument, the proof in the disjunctive case is highly non-trivial. We present for the disjunctive case a *non-constructive* proof that each query has PTIME data complexity. The proof yields the fact that the data complexity is in PTIME, but without explicitly describing for each query a specific algorithm that solves the problem with that complexity. Although we conjecture that with further analysis the proof can be made constructive, we have not been able to achieve this.

Thus, although the monadic case has very limited expressive power from the point of view of applications, its non-triviality makes it of theoretical interest. Moreover, we feel that an understanding of the monadic case is a prerequisite for finding tractable classes of problems with higher arity. (The fruits of this approach will appear in Section 6.5, where we adapt methods used in the monadic case to find a tractable class of databases capable of expressing interval data.) It is therefore worthwhile to consider restricted

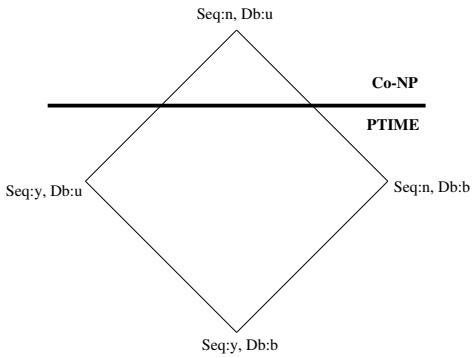

Figure 5.4: Combined complexity of conjunctive queries using monadic predicates

versions of the query problem in the case of monadic predicates to determine under what circumstances the combined complexity can be shown to be in PTIME. We consider three parameters in all.

The first of these is a constraint on the type of the query. A *sequential* query is a conjunctive query like $\exists xyz[x<y<z \wedge P(x) \wedge Q(x) \wedge P(y) \wedge Q(z)]$ which asks whether a particular sequence of events occurs. A second parameter that helps to obtain tractable combined complexity in the monadic case is the *width* of the database. This is a property of the partial order that the data imposes on the order constants. Formally, the width of a database is the largest number k for which there exists a set of k mutually independent elements in this partial order. Intuitively, width is a measure of the number of order constants that potentially refer to the same point in the linear order. For example, suppose that the database is a record of the reports of a number of agents independently observing the world. If each provides a linearly ordered set of observations, then the width of the database is the number of agents. Thus, the database of Example 5.1.1 has width two. We believe that placing a bound on the width of the database is a realistic constraint for some applications.

The proof that conjunctive queries using monadic predicates have co-NP hard combined complexity involves nonsequential queries and databases of unbounded width. However, if we restrict either to sequential queries or to databases of width bounded by a constant, combined complexity may be shown to be in PTIME, as indicated in Figure 5.4. Here 'Seq' refers to sequentiality of the query and has a value of either yes ('y') or no ('n'). The expression 'Db' refers to the boundedness of the database: 'b'

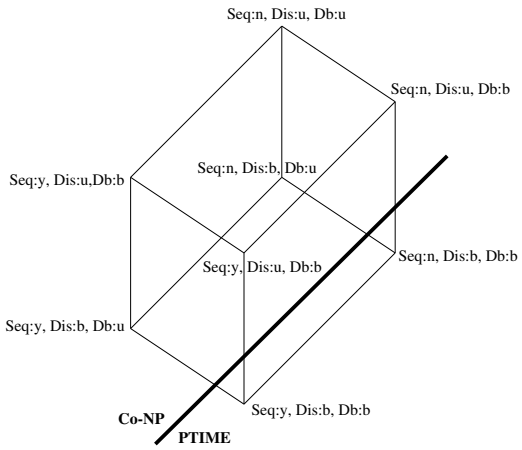

Figure 5.5: Combined complexity of disjunctive queries using monadic predicates

means bounded, 'u' means unbounded.

For disjunctive queries using only monadic predicates, it turns out that even with respect to databases of bounded width, queries which are disjunctions of sequential conjunctive queries have co-NP hard combined complexity. However, a final restriction, that bounding the number of disjuncts of queries, does help. In databases of bounded width, monadic queries with a bounded number of disjuncts have PTIME combined complexity. The proof is by a reduction to finite state automaton containment. Figure 5.5 shows the precise demarcation between the problems stated in terms of the three parameters which have PTIME complexity and those that have co-NP hard complexity. Here 'Dis' refers to the boundedness of the number of disjuncts: 'b' means bounded, 'u' means unbounded.

We remark that the restrictions that help to obtain PTIME complexity in the monadic case do not directly apply to queries involving binary predicates. In particular, a bound on the width of the database does not reduce data complexity in the binary case. However, the fact that we are able to find a variety of conditions under which we are able to show PTIME complexity in the monadic case suggests that it may be worthwhile reconsider the binary case. We will do so at the end of the next chapter, where we find a class of databases containing binary predicates for which data complexity is in PTIME. This result uses an interesting combination of the ideas from Chapter 3 and the present chapter.

The structure of this chapter is as follows. Section 5.2 considers the relations between a number of different semantics for indefinite order databases, depending on the type of the linear order. Section 5.3 is concerned with upper and lower bounds for queries containing predicates of arbitrary arity. The remainder of the chapter studies the monadic case. Section 5.4 deals with upper and lower bounds for conjunctive monadic queries. A number of the results developed in this section are crucial to the proofs of the bounds established in Section 5.5 for the disjunctive monadic case. Section 5.6 discusses some loosely related literature.

5.2 Definitions and Preliminary Results

This section is devoted to setting up the semantic framework for order databases by formally defining three consequence relations, depending on the structure of the linear order in models. We establish reductions between these three relations that permit us to focus on just one type of semantics, the finite model semantics, when developing complexity results, and give a technical characterization of this semantics that will be helpful in establishing complexity results.

A structure for an order database D will be a (two-sorted) first-order structure M in which the relation $<$ denotes a linear order $<_M$ on the order sort. We require that the object sort and the order sort be disjoint. Such a structure will be a model of a database just in case it supports the database as a first order theory. We reserve the word 'points' to refer to elements of the order sort; elements of the object sort will be called 'objects'. We do not consider the unique names assumption in this chapter: for order constants this is because we assume that we have no information on the points denoted; for object constants the adoption of the unique names assumption would have no effect on query entailment. We will consider various semantics for $\{<\}$-databases, corresponding to different restrictions on the linear order. If $\mathcal{O}$ is a class of linear order types we will write

$$Mod_{\mathcal{O}}(D) = \{M \mid M \models D \text{ and } <_M \text{ is of type } \mathcal{O}\}$$

The class $\mathcal{O}$ will be either the class **Fin** of finite linear orders, the class **Z** of linear orders

isomorphic to the natural numbers or the class **Q** of dense linear orders isomorphic to the rationals. For each class $\mathcal{O}$ we obtain a consequence relation $\models_{\mathcal{O}}$ defined by

$$D \models_{\mathcal{O}} \Phi \quad \text{iff} \quad M \models \Phi \text{ for all } M \in Mod_{\mathcal{O}}(D)$$

For the restricted form of database and query we are considering, these consequence relations are closely related, as we now show.

Proposition 5.2.1: The following containments hold between the consequence relations: $\models_{\mathbf{Fin}} \subseteq \models_{\mathbf{Z}} \subseteq \models_{\mathbf{Q}}$.

Proof: We show that $D \models_{\mathbf{Z}} \Phi$ implies $D \models_{\mathbf{Q}} \Phi$, by proving the contrapositive. Suppose that $D \not\models_{\mathbf{Q}} \Phi$. Then there exists a model $M \in Mod_{\mathbf{Q}}(D)$ with $M \not\models \Phi$. Let S be the image of the constants of D in M. Add additional elements of the order domain of M to S so as to obtain an order isomorphic to **Z**. Now let M' be the restriction of the model M to the resulting subset of the domain. Clearly there exists a homomorphism from M' to M, from which $M' \not\models \Phi$ follows. Hence $D \not\models_{\mathbf{Z}} \Phi$ also. This shows that $D \models_{\mathbf{Z}} \Phi$ implies $D \models_{\mathbf{Q}} \Phi$. A similar argument shows that $D \models_{\mathbf{Fin}} \Phi$ implies $D \models_{\mathbf{Z}} \Phi$. □

To see that these consequence relations are inequivalent, observe that $\models_{\mathbf{Z}} \exists t_1 t_2 [t_1 < t_2]$ but not $\models_{\mathbf{Fin}} \exists t_1 t_2 [t_1 < t_2]$, since **Fin** contains the linear order consisting of a single point. Similarly, note that if $D = \{P(u), P(v), u<v\}$ and $\Phi = \exists t_1 t_2 t_3 [P(t_1) \wedge t_1 < t_2 < t_3 \wedge P(t_3)]$ then $D \models_{\mathbf{Q}} \Phi$ but not $D \models_{\mathbf{Z}} \Phi$. In both of these examples we have variables which occur only in order atoms. This is in fact a necessary condition for such examples. Say that a query is *tight* if in each disjunct, every variable occurs in some proper atom. Then we have the following:

Proposition 5.2.2: If Φ is a tight query then $D \models_{\mathbf{Fin}} \Phi$ iff $D \models_{\mathbf{Z}} \Phi$ iff $D \models_{\mathbf{Q}} \Phi$.

Proof: By Proposition 5.2.1 it suffices to show that $D \models_{\mathbf{Q}} \Phi$ implies $D \models_{\mathbf{Fin}} \Phi$. We establish the contrapositive. Suppose that there exists $M \in Mod_{\mathbf{Fin}}(D)$ with $M \not\models \Phi$, where the order domain is the set $\{0 \ldots n\}$. Modify M by enlarging the order domain to the set of rational numbers. This produces a model M' in $Mod_{\mathbf{Q}}(D)$. Suppose that $M' \models \Phi$ and let θ be a satisfying assignment for the variables of Φ. Since Φ is tight, for each variable t we must have $\theta(t)$ in the set $\{0 \ldots n\}$. But this implies $M \models \Phi$, a contradiction. Thus $M' \not\models \Phi$, establishing that $D \not\models_{\mathbf{Q}} \Phi$. □

In many applications queries will generally be tight. One example of an application of order databases satisfying this constraint is the containment problem for relational database queries containing inequalities (see Section 5.1.) In formulating such queries we are generally interested in retrieving data values from the database and doing comparisons on these values. This means that we are only interested in values actually occurring in some database relation. The result shows that for these purposes the three types of semantics are equivalent.

However, there are applications which make use of non-tight queries. We have already seen an instance of this in Example 5.1.1, in which we used the query

$$\exists x t_1 t_2 t_3 t_4 w[IC(t_1, t_2, x) \wedge IC(t_3, t_4, x) \wedge t_1 < w < t_2 \wedge$$
$$t_3 < w < t_4 \wedge (t_1 < t_3 \vee t_2 < t_4)]$$

to express the integrity constraint that overlapping intervals of the form $IC(u, v, x)$ must be identical. Note that the occurrence of w in this query is not tight. Therefore it is of some interest to understand the relations between the three semantics for non-tight queries. We will establish polynomial time reductions of the relations $\models_{\mathbf{Q}}$ and $\models_{\mathbf{Z}}$ to the relation $\models_{\mathbf{Fin}}$. These reductions enable query answering procedures for one semantics to be used for another, with no more than a polynomial loss in complexity. We note that these reductions will be established in one direction only, so they do not serve to transfer lower bounds on complexity from the relation $\models_{\mathbf{Fin}}$ to the relations $\models_{\mathbf{Q}}$ and $\models_{\mathbf{Z}}$. The reason we do not need to establish the converse reductions is that all of the lower bounds to be proved for $\models_{\mathbf{Fin}}$ will make use of tight queries only, so these bounds apply to all three semantics, by Proposition 5.2.2.

We begin with the reduction for $\models_{\mathbf{Z}}$. Suppose that the query Φ contains n distinct variables. Given a database D, let $l_1, \ldots, l_n, r_1, \ldots, r_n$ be $2n$ new constants. Add to D the atoms $l_1 < l_2 < \ldots < l_n$ and $r_1 < r_2 < \ldots < r_n$, as well as $l_n < u < r_1$ for each order constant u of D, and call the resulting database D'.

Proposition 5.2.3: For every $\{<\}$-database D, $D \models_{\mathbf{Z}} \Phi$ if and only if $D' \models_{\mathbf{Fin}} \Phi$.

Proof: We first show that $D' \models_{\mathbf{Fin}} \Phi$ implies $D \models_{\mathbf{Z}} \Phi$. Suppose $D' \models_{\mathbf{Fin}} \Phi$ and let $M \in Mod_{\mathbf{Z}}(D)$. Consider the finite model M' obtained from M by restricting the

domain to the image of the constants in D. Add n points $u_1 < \ldots < u_n$ less than the least point in M' and another n points $v_1 < \ldots < v_n$ greater than the greatest point in M', and interpret l_i as u_i and r_i as v_i, for $i = 1 \ldots n$. The resulting structure M'' is a clearly a finite model of D'. Since $D' \models_{\mathbf{Fin}} \Phi$, we must have $M'' \models \Phi$. Clearly there exists a homomorphism from M'' to M, so it follows that $M \models \Phi$ also. This establishes that $D' \models_{\mathbf{Fin}} \Phi$ implies $D \models_{\mathbf{Z}} \Phi$.

For the converse, suppose that $D \models_{\mathbf{Z}} \Phi$. Let M' be a finite model of D'. By restricting the domain to the image of the constants of D' in M', restricting the proper facts to just those needed to support D, and renaming elements of the order domain, we obtain a finite model M'' of D' such that

1. The order domain of M'' is the set $\{-n, \ldots, k+n\}$.
2. The order constants of D are interpreted by M'' in the set $\{0, \ldots, k\}$.
3. The constant l_i is interpreted in M'' as $-i$, and the constant r_i is interpreted as $k+i$, for $i = 1 \ldots n$.
4. There exists a homomorphism from M'' to M'.

Let M be the model obtained from M'' by extending the order domain to $\mathbf{Z}$. Since $D \models_{\mathbf{Z}} \Phi$ there exists a satisfying assignment θ for Φ in the model M. Consider the order variables V of Φ that θ maps outside of the set $\{0, \ldots, k\}$. By construction of M, none of these variables can occur in a proper atom in Φ, hence they occur only in order atoms. There are at most n of these variables. Hence, by changing the assignment θ on V, we may construct an assignment θ' that maps the variables V into the set $\{-n, \ldots, k+n\}$ without changing the order relationships that hold. That is, for any order variables u, v occuring in Φ, $\theta'(u) < \theta'(v)$ if and only if $\theta(u) < \theta(v)$. It then follows that $M'' \models \Phi$, hence also that $M' \models \Phi$. This establishes that $D \models_{\mathbf{Z}} \Phi$ implies $D' \models_{\mathbf{Fin}} \Phi$. $\square$

It is also possible to give a reduction of the semantics based on the rationals to finite models. To do this, it is necessary to do some massaging of queries. First, we introduce the notion of the graph associated with a database or *conjunctive* query. The vertices of this graph are the order constants of D, or the order variables of the query

Φ, respectively. For each atom $u < v$ in the database or query there is an edge from u to v.

The edges of the graph G associated with a database D need not correspond to all the order relations between order constants that may be inferred from the database. For example, if we have atoms $u < v$ and $v < w$ then we may infer $u < w$. More generally, if there exists a path from u to v in G, then $D \models u < v$. It turns out that this rule is complete, in the sense that if the database entails an atom of the form $u < v$ then there will exist a path from u to v in its graph. We will say that a database is *normalized* if its graph is transitively closed, i.e., contains all deduced relations. Similarly, a query is normalized if each disjunct contains all relations between order variables that may be deduced from the relations in that disjunct. For example, the query

$$\exists uvw[Q(u,v,w) \wedge u{<}w] \vee \exists uvw[Q(u,v,w) \wedge u{<}v \wedge v{<}w \wedge u{<}w]$$

is normalized, but the query $\exists uvw[Q(u,v,w) \wedge u{<}v \wedge v{<}w]$ is not, since it does not contain the derived atom $u < w$.

Clearly, a database or conjunctive query is inconsistent if there exists a cycle in its graph, for there exists a constant (or variable) u such that $u < u$ is entailed. Conversely, if the graph is acyclic (that is, if it is a directed acyclic graph, or *dag*) then the database or query is consistent. This is simply the well known fact that it is possible to "topologically sort" a dag, that is, produce a linear order satisfying all the order relations entailed by the dag. We will assume throughout that we are dealing only with consistent databases and queries.

Associated with any directed acyclic graph is a partial order on its vertices, given by $u \leq v$ if either $u = v$ or there exists a path from the vertex u to the vertex v. (Acyclicity entails that this relation on vertices is a partial order.) It follows that we may talk of the *minimal* elements of a dag: these are simply the minimal elements of the associated order, that is, the elements v such that there does not exist a vertex u with an edge from u to v.

We will use the phrase "topological sort" of a dag to refer to a slightly more general class of compatible linear orders than usually meant by this term. For us, a topological

sort will be any mapping f from the points in the graph to a linear order, which preserves the order relations, and such that the domain of the linear order is equal to the image of the mapping f. Such mappings are obtained in a series of stages by the following procedure. At each stage we have a partially constructed linear order, together with a subgraph of the original graph, which contains all the vertices not yet mapped into the linear order. Initially the linear order is empty and the graph of unsorted vertices is the original graph. We repeat the following steps until the entire graph has been sorted. First, we non-deterministically select some set S of vertices, subject to the constraint that each element of S is minimal in the graph of unsorted vertices. We map the elements of S to the 'next' point of the finite linear order being constructed, and delete the vertices S from the graph of unsorted vertices.

Example 5.2.4: Suppose we are given the set of order atoms $u < v < w, t < w$. We demonstrate one of the topological sorts of this set. We begin with an empty linear order and the original set of atoms. The minimal unsorted points at this stage are u and t. Let us choose $S = \{u\}$ as the elements mapping to the first point x_1 of the linear order. Deleting the elements in S leaves the graph $v < w, t < w$. This has minimal elements v and t, so let us now choose $S = \{v, t\}$ as the elements mapping to the next point x_2 of the linear order. Deleting these elements leaves just the point w, which we map to the last point x_3 of the linear order. Thus the topological sort obtained consists of the linear order with three points $x_1 < x_2 < x_3$, together with the mapping f with $f(u) = x_1$, $f(v) = f(t) = x_2$ and $f(w) = x_3$. Other topological sorts of this order are obtained by making different choices for the set of minimal elements S. □

We may now state the reduction for the dense order semantics. Let Φ be a normalized query. Delete from each disjunct of Φ any order variables that occur only in order atoms in that disjunct, (in other words, those that do not occur in any proper atom) as well as any quantifiers or order atoms containing those variables, and call the resulting query Φ'. For example, if Φ is the normalized query

$$\exists uvw[P(u, w) \wedge u < v \wedge v < w \wedge u < w]$$

in which the variable v does not occur in any proper atom then Φ' is the query $\exists uw[P(u,w) \wedge u<w]$ in which all order atoms containing this variable have been deleted.

Lemma 5.2.5: If Φ is a normalized $\{<\}$-query then $D \models_{\mathbf{Q}} \Phi$ if and only if $D \models_{\mathbf{Q}} \Phi'$, for every $\{<\}$-database D.

Proof: We assume without loss of generality that Φ is consistent. It is obvious that $D \models_{\mathbf{Q}} \Phi$ implies $D \models_{\mathbf{Q}} \Phi'$. For the converse, assume that $D \models_{\mathbf{Q}} \Phi'$ and let M be any model in $Mod_{\mathbf{Q}}(D)$. Then there exists a satisfying assignment θ for some disjunct Ψ' of Φ'. We show that θ may be extended to a satisfying assignment θ' of the corresponding disjunct Ψ of Φ by providing values for the 'deleted' variables in such a way as to satisfy all the deleted order atoms.

Let V be the set of undeleted order variables and U the set of deleted variables of Ψ. We construct the extension θ' of θ by a applying to the graph G of the disjunct Ψ a modified version of the topological sorting process discussed above, in which we map to $\mathbf{Q}$ instead of a finite linear order. Suppose that the image of θ consists of the points $t_1<t_2<\ldots<t_n$. It is convenient to use (any) two other points $t_0<t_1$ and $t_{n+1}>t_n$. We maintain variables T_L and T_R. The variable T_L records the greatest point in the image of the function so far constructed, and has initial value t_0. The variable T_R records the next point of the form t_i not yet in the sorted portion. Initially we have $T_R = t_1$. We will maintain the invariant that if t_i is in the sorted portion then all the variables $u \in V$ satisfying $\theta(u) = t_i$ have been mapped to t_i. The modified topological sort proceeds as follows. Whenever possible, we choose for the set S of minimal unsorted variables a single variable u in the set U of values unassigned by θ. We put $\theta'(u) = t$ where t is a point satisfying $T_L<t<T_R$. We then delete u from the unsorted portion of G as usual, and set $T_L = t$ before proceeding to the next stage. If there does not exist a variable in U which is minimal in the unsorted portion, it must be the case that all elements of the set $S = \{v \in V \mid \theta(v) = T_R\}$ are minimal in the unsorted portion. For otherwise, there exists a variable $v \in S$ and a variable $u \in V$ such that the atom $u < v$ is in the normalized disjunct Ψ. Since both u and v are elements of V this atom is also in Ψ', and therefore satisfied under the assignment θ. By the invariant mentioned above we must have that $\theta(u) \geq T_R$. Combining this with the fact that the constraint $u < v$ is satisfied

under the assignment θ, we get that $T_R \leq \theta(u) < \theta(v) = T_R$, a contradiction. Thus, since all of S is minimal in the unsorted portion we may let θ' map this set of variables to the point $T_R = t_i$. Notice that this maintains the invariant. Before proceeding to the next stage we put $T_L = t_i$ and $T_R = t_{i+1}$. It is clear that once this process is done we have constructed an assignment θ' extending θ which satisfies Ψ. □

Corollary 5.2.6: For every normalized $\{<\}$-query Φ and $\{<\}$-database D we have $D \models_{\mathbf{Q}} \Phi$ if and only if $D \models_{\mathbf{Fin}} \Phi'$

Proof: Follows directly from Proposition 5.2.2 and Lemma 5.2.5 on noting that the query Φ' is tight. □

Propositions 5.2.3 and 5.2.2 and Corollary 5.2.6 show that it suffices to restrict attention to finite models. In fact we may use an even smaller class of models, the *minimal models*, which are just those models obtained from the atoms of the database by interpreting the object constants as themselves, and interpreting the order constants by topologically sorting the graph of a database. We write $Mod(D)$ for this class of models and let $\models$ be the corresponding consequence relation.

Example 5.2.7: Let D be the database consisting of the order atoms $u < v < w$,$t < w$ from Example 5.2.4 together with the proper atoms $B(a,t)$, $B(b,w)$, in which a and b are object constants. Then one minimal model is the minimal model with object domain $\{a,b\}$ and order domain consisting of the three points $x_1 < x_2 < x_3$. The object constants are interpreted as themselves, and the order constants are interpreted by the mapping f with $f(u) = x_1$, $f(v) = f(t) = x_2$ and $f(w) = x_3$ obtained from the topological sort of Example 5.2.4. The atomic facts holding in the model are $B(a,x_2), B(b,x_3)$. □

We now establish a result which explains why we use the term "minimal model" to refer to the models obtained by topologically sorting a database. The set of all models (with any type of linear order) may be (quasi) ordered by $M \leq N$ when there exists a homomorphism from M to N. The following states that the minimal models of a database are in fact minimal in this order.

Proposition 5.2.8: For every model N (of any order type) of a $\{<\}$-database D

there exists a minimal model M with a homomorphism $h : M \rightarrow N$.

Proof: Let the model N interpret the object constants of D by the function f and the order constants by the function g. For the object domain of the model M we take the set of object constants of D, with each object constant interpreted as itself. For the order domain of M we take the image of the function g, with each order constant u interpreted as $g(u)$. The linear order on the order domain of M is that induced from N. Finally, the proper atoms holding in M are just those that are the image under the interpretation of the constants of some proper atom of D. Let h be the function that maps an element a of the object domain of M (i.e. a constant of D) to $f(a)$, and acts as the identity function on the order domain. It is straightforward to verify that this is a homomorphism, and that M is in fact a minimal model. □

In the context of definite Horn theories minimal models are in fact initial objects in the category with objects the models of the theory and arrows the homomorphisms between models [80], and are characterized by the slogan "no junk and no confusion". That is, these models have no unnecessary atoms (no junk), and do not identify any distinct ground terms (no confusion). Since we are dealing with indefinite information, we cannot expect to have the initiality property. However, our minimal models do satisfy the "no junk" property with respect to proper atoms. We have no confusion of object constants, since these must denote themselves in minimal models, but we do have confusion of order constants, since a topological sort may map two object constants to the same point.

Corollary 5.2.9: For every database D and query Φ we have $D \models \Phi$ if and only if $D \models_{\mathbf{Fin}} \Phi$.

Proof: Immediate from Proposition 5.2.8. □

The reductions of this section show that even if we are interested in integer and rational order it suffices to restrict our attention to the finite model semantics. For the remainder of the chapter we will develop our results with respect to this semantics. However, it is technically more convenient to work with minimal models, so we will state and prove our results in terms of the consequence relation $\models$. Corollary 5.2.9

provides the justification for this.

We will adopt one more simplifying assumption: queries will be assumed not to contain constants. By a well known construction, there is for most purposes no loss of generality in this. We may introduce a new monadic predicate P_u for each constant symbol u, and add the facts $P_u(u)$ to the database. Then the query $\Phi(u)$ containing the constant u is equivalent to the query $\exists t[P_u(t) \wedge \Phi(t)]$ in which the constant has been eliminated. The advantage of this construction is that it enables us to discard from models the mappings interpreting constants. This will be important in some of our proofs.

Finally, we introduce here a parameter of databases that will be important in the sequel. Two elements x, y of a partial order are said to be *independent* if neither $x \leq y$ nor $y \leq x$ holds. If G is a partial order then the *width* of G is the cardinality of the largest independent subset of G, that is, the largest set S such that for no pair of distinct elements $x, y \in S$ do we have $x < y$. The width of a database, or of a conjunctive query, is just the width of the associated partial order.

If two order constants u, v in a $\{<\}$-database are independent in the associated partial order, then they may be viewed as being potentially concurrent. In models of the database any of the relations $u < v$, $u = v$ or $v < u$ may hold. Intuitively, the width of a database is a measure of the extent of indefiniteness at each stage of a topological sort of the database. For example, a database recording the reports of k observers, each providing a linear sequence of events, has width k. We will see below that width of databases is an important parameter in the complexity of query processing. Broadly speaking, query processing has lower complexity in databases with bounded width.

5.3 Order Databases with n-ary Predicates

In this section we consider the complexity of query problems in unconstrained order databases, establishing both upper and lower bounds. These bounds will show that the general versions of these problems are probably intractable. The rest of the chapter will be devoted to a search for natural restrictions under which the complexity decreases.

In this section, and for the remainder of the chapter, we confine ourselves to results for the finite model semantics.

Upper bounds for the finite model semantics follow directly from the observation in the previous section that it suffices to restrict attention to minimal models. It is clear that the non-deterministic process constructing the minimal models operates in a polynomial number of steps. Thus, noting that positive existential queries have expression complexity in NP with respect to first order structures (see Section 2.2), we have the following immediate consequences of Corollary 5.2.9. (Part (1) has previously been noted by Klug [64].)

Proposition 5.3.1: (1) The combined complexity of $\{<\}$-databases and positive existential queries is in Π_2^p.

(2) The data complexity of positive existential $\{<\}$-queries is in co-NP.

(3) The expression complexity of $\{<\}$-databases is in NP.

This result holds for disjunctive as well as conjunctive queries. We now set about showing that these bounds can be met by corresponding lower bounds stated in terms of conjunctive queries.

Theorem 5.3.2: There exists a conjunctive $\{<\}$-query containing only binary predicates with co-NP hard data complexity on $\{<\}$-databases.

Proof: The proof is by reduction from monotone 3–satisfiability. We show that there exists a query Φ and a polynomial time reduction from sets of 3–clauses $\mathcal{S}$ to $\{<\}$-databases D such that $D \models \Phi$ if and only if $\mathcal{S}$ is unsatisfiable. Given object constants a, b, c and order constants u, v, w, t define the database $D(a, b, c; u, v, w, t)$ to be the set of atoms

$$\{P(u,a), P(u,b), u{<}v, P(v,a), P(v,c), v{<}w, P(w,b), P(w,c),$$
$$P(t,a), P(t,b), P(t,c)\}$$

depicted in Figure 5.6. Let $\varphi(x)$ be the query

$$\exists t_1 t_2 t_3 [P(t_1, x) \wedge t_1{<}t_2 \wedge P(t_2, x) \wedge t_2{<}t_3 \wedge P(t_3, x)]$$

also shown in Figure 5.6. The combination of this database and query "expresses" the disjunction "a or b or c" in the following sense. First, in every model of $D(a, b, c; u, v, w, t)$

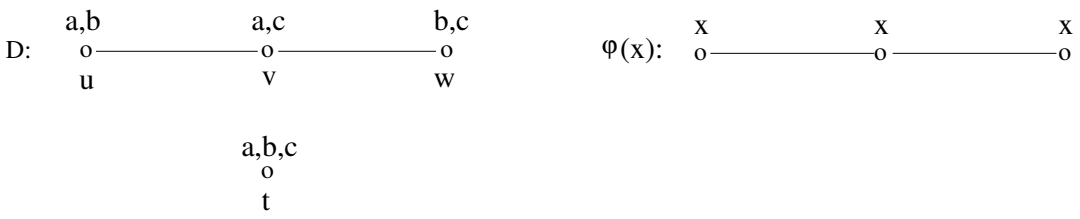

Figure 5.6: Simulating Ternary Disjunctions

either $\varphi(a)$ or $\varphi(b)$ or $\varphi(c)$ holds. Furthermore, there exists a model in which $\varphi(a)$ is true but $\varphi(b)$ and $\varphi(c)$ are false, namely the model in which $t = w$. Similarly, there exist models in which only $\varphi(b)$ is true, or in which only $\varphi(c)$ is true.

Suppose we are given a set S of positive 3–clauses over a set of propositional letters L. The simulation of ternary disjunctions just described will be used to construct a $\{<\}$-database $D(S)$ and a query $\psi(x)$ for which the set $\{l \in L \mid M \models \psi(l)\}$ simulates, as M varies over minimal models of $D(S)$, the set of valuations satisfying the set S. One apparent way to do this is to let the database contain a component of the form $D(l_1, l_2, l_3; u, v, w, t)$ for each clause $l_1 \vee l_2 \vee l_3$. Unfortunately, propositional letters may occur in more than one clause and this may result in interference among the components. Instead, we will generate disjunctions independently and then transmit these disjunctions to the propositional letters. Specifically, for the i-th clause $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ in S we introduce *new* object constants a_i, b_i, c_i and order constants u_i, v_i, w_i, t_i, and let D_i be the set of facts

$$D(a_i, b_i, c_i; u_i, v_i, w_i, t_i) \cup \{Q(l_{i,1}, a_i), Q(l_{i,2}, b_i), Q(l_{i,3}, c_i)\}.$$

Note that we treat the propositional letters l as object constants. Define $D(S)$ to be the union of the databases D_i, and let $\psi(x)$ be the query $\exists t[Q(x, t) \wedge \varphi(t)]$.

We can do the same for a set S' of negative clauses, using complemented constants $\bar{l}$ and facts $\left\{Q\left(\overline{l_{i,1}}, a_i\right), Q\left(\overline{l_{i,2}}, b_i\right), Q\left(\overline{l_{i,1}}, c_i\right)\right\}$ instead. Take F to be the set of facts of the form $Comp\left(l, \bar{l}\right)$ for l in the set L of propositional letters. Then given a set S of positive 3–clauses and a set S' of negative 3-clauses, we claim that $D \models \Phi$ exactly when the set of clauses $S \cup S'$ is unsatisfiable, where $D = D(S) \cup D(S') \cup F$ and $\Phi = \exists x y(\psi(x) \wedge Comp(x, y) \wedge \psi(y))$.

To see this, assume first that $D \models \Phi$, and suppose V is a valuation of L which

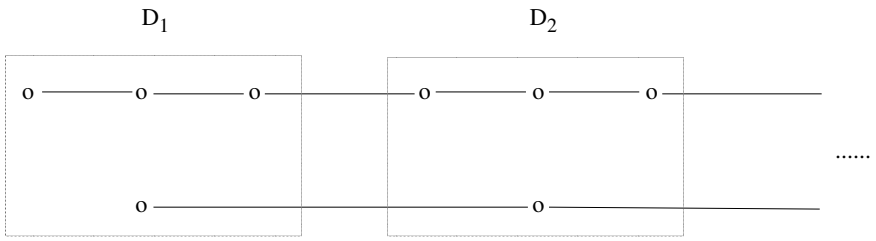

Figure 5.7: A width two database

satisfies $S \cup S'$. Then for each clause $l_{i,1} \vee l_{i,2} \vee l_{i,3}$ in S there exists an index j such that $V(l_{i,j}) = 1$. By the construction above we may choose a model M_i of D_i such that $M_i \models \psi(l_{i,k})$ only if $k = j$, and similarly for the negative clauses. Composing these models, we obtain a model M of D with the property that $M \models \psi(l)$ implies $V(l) = 1$ and $M \models \psi\left(\overline{l}\right)$ implies $V(l) = 0$. But since M is a model of D, we must have $D \models \Phi$. This is readily seen to imply that there exists a propositional letter l such that both $V(l) = 1$ and $V(l) = 0$, a contradiction.

Conversely, suppose that M is a model of D with $M \not\models \Phi$. We construct a valuation V as follows: if $M \models \psi(l)$ we put $V(l) = 1$, else $V(l) = 0$. We claim this is a satisfying valuation of $S \cup S'$. For the positive clauses, this follows directly from the construction of the D_i, since there must exist an index j such that $M \models \psi(l_{i,j})$. Consider next the negative clauses $\overline{l_1} \vee \overline{l_2} \vee \overline{l_3}$. If this clause is not satisfied, then $V(l_1) = V(l_2) = V(l_3) = 1$, which together with the fact that M falsifies the query, implies that $M \not\models \psi\left(\overline{l_i}\right)$ for each i. But this contradicts the fact that D was constructed so as to ensure that in each model at least one of the $\psi\left(\overline{l_i}\right)$ is satisfied. □

We note that it is possible to make the database of this proof have bounded width. The "disjunction-generating" parts of the D_i are the only components in this proof containing order constants, and the proof still goes through if these are placed in a linear order as depicted in Figure 5.7, yielding a database of width two. Furthermore, the proof does not depend on the fact that the constants a_i, b_i, c_i and the constants $l_i, \overline{l_i}$ are object constants: it still goes through if we take these to be order constants instead. Indeed, we may also place all of these constants in a linear sequence following one of the sequences of Figure 5.7: this way the width of the database is still two.

The proof of the next result uses the following complete problem: A Π_2 formula of

quantified propositional logic is an expression

$$\forall p_1 \ldots p_n \exists q_1 \ldots q_m [\alpha] \tag{5.1}$$

where α is a formula of propositional logic containing only the variables $p_1 \ldots p_n q_1 \ldots q_m$.

Such a formula is true if for every assignment of boolean truth value to the variables p_i there exists an assignment of truth values to the variables q_i under which the formula α is true. The set Π_2-SAT is the set of all true Π_2 formulae. This is a generalization of the problem of satisfiability to the polynomial hierarchy. It is known that the set Π_2-SAT is complete for the level Π_2^p of this hierarchy [13].

Theorem 5.3.3: The combined complexity of $\{<\}$-databases and conjunctive $\{<\}$-queries is Π_2^p hard.

Proof: We use a reduction from Π_2-SAT. We reuse some ideas from the previous proof to express binary disjunctions. Consider the database

$$D_i = \{P_i(u_i, t), P_i(v_i, f), u_i < v_i, P_i(w_i, t), P_i(w_i, f)\}$$

As before, we may use the formula

$$\varphi_i(x) = \exists t_1 t_2 [P_i(t_1, x) \wedge P_i(t_2, x) \wedge t_1 < t_2]$$

to express the disjunction $\varphi_i(t) \vee \varphi_i(f)$. We will use the databases D_i and the formulae $\varphi_i(x)$ to simulate the assignment of truth values to the variables p_i. To simulate the calculation of the truth value of the formula α, we let E be the set of facts

$$Istrue(t) \quad \begin{array}{lll} And(t,t,t) & Or(t,t,t) & \\ And(t,f,f) & Or(t,f,t) & Not(t,f) \\ And(f,t,f) & Or(f,t,t) & Not(f,t) \\ And(f,f,f) & Or(f,f,f) & \end{array}$$

Now we define inductively the query $TV(\alpha, \mathbf{z}, x)$ where α is a formula of propositional logic in the propositional variables $p_1 \ldots p_k$ and $\mathbf{z}$ is the vector of variables $z_1 \ldots z_k$. Intuitively this asserts that the truth value of the formula α under the assignment $z_1 \ldots z_k$ to the variables $p_1 \ldots p_k$ is x. If α is a propositional variable p_i then $TV(\alpha, \mathbf{z}, x)$

is the formula $x = z_i$. For the inductive case we have

$$\begin{aligned} TV(\neg\alpha, \mathbf{z}, x) &= \exists t(Not(t, x) \wedge TV(\alpha, \mathbf{z}, t)) \\ TV(\alpha \wedge \beta, \mathbf{z}, x) &= \exists t_1 t_2[And(t_1, t_2, x) \wedge TV(\alpha, \mathbf{z}, t_1) \wedge TV(\beta, \mathbf{z}, t_2)] \\ TV(\alpha \vee \beta, \mathbf{z}, x) &= \exists t_1 t_2[Or(t_1, t_2, x) \wedge TV(\alpha, \mathbf{z}, t_1) \wedge TV(\beta, \mathbf{z}, t_2)] \end{aligned}$$

Note that at each level we need to use fresh existentially quantified variables. A straightforward induction shows that if each z_i is either the constant t or f, representing truth and falsity of the propositional constant p_i respectively, then the database E entails $TV(\alpha, \mathbf{z}, x)$ if and only if x is the truth value of the formula α under the assignment $\mathbf{z}$. (The use of equality in the definition of the operator TV is purely for convenience. Strictly, we have not permitted equality in our query language, but it is straightforward to eliminate. For example, $TV(\neg p_1, z_1, x)$ is the formula $\exists t(Not(t, x) \wedge t = z_1)$ which is equivalent to $Not(z_1, x)$.)

Now encode the quantified boolean formula (5.1) using the query

$$\begin{aligned} \Phi = \exists z_1 \ldots z_n[&\varphi_1(z_1) \wedge \ldots \wedge \varphi_n(z_n) \wedge \\ &\exists x\, z_{n+1} \ldots z_{n+m}\{Istrue(x) \wedge TV(\alpha, z_1 \ldots z_{n+m}, x)\}] \end{aligned}$$

and let the database D be the union of the databases D_i and E. We claim that $D \models \Phi$ exactly when the quantified boolean formula is true. For, suppose that the formula is true, and let M be any model of D. By construction of the D_i the model M supports either $\varphi_i(t)$ or $\varphi_i(f)$. The truth of Φ in M now follows from the truth of the quantified boolean formula and the meaning of the formula TV. Conversely, suppose Φ is true in every model of D. By construction, there exists for every vector $z_1 \ldots z_n$ of truth values a model M of D such that $M \models \varphi_i(x)$ if and only if $x = z_i$. Since M supports Φ there exist truth values $z_{n+1} \ldots z_{n+m}$ such that the formula α is true under the assignment $z_1 \ldots z_{n+m}$. This shows that the quantified formula (5.1) is true. □

We note that it is possible to modify the proof to use only a fixed finite set of binary predicates instead of the infinite set of predicates P_i. One way to do this is to use a chain of facts $P(u, v, u_0), R(u_0, u_1), R(u_1, u_2), \ldots, R(u_{i-1}, u_i), Q(u_i)$ of length i instead of the atom $P_i(u, v)$ in the database, and use

$$\exists t_0 \ldots t_i[P(x, y, t_0) \wedge R(t_0, t_1) \wedge \ldots \wedge R(t_{i-1}, t_i) \wedge Q(t_i)]$$

in the query in place of each occurrence $P_i(x, y)$. We may then make all predicates binary by means of the well-known reduction of n-ary predicates to binary.

Theorem 5.3.4: There exists a $\{<\}$-database with NP hard expression complexity for conjunctive $\{<\}$-queries.

This follows from the fact that already relational databases have NP hard expression complexity for conjunctive queries. The proof of this is implicit in the proof of Theorem 5.3.3: if α is a propositional formula containing propositional variables $x_1, \ldots, x_n$ then the query

$$\exists x \exists z_1 \ldots z_n [Istrue(x) \wedge TV(\alpha, z_1 \ldots z_n, x)]$$

is entailed by the set E just in case the formula α is satisfiable.

5.4 Conjunctive Monadic Queries

We now embark on our study of restricted forms of the query problems with complexity lower than the general case, which we have just seen to be probably intractable. The lower bounds of Section 5.3 all required the use of at least binary predicates. As we argued in Section 5.1, there are significant applications for which monadic predicates suffice, so we are led to investigate the case in which all predicates are monadic. In this section we will consider only conjunctive queries. The next section will deal with the disjunctive case. Some of the results we develop here will be crucial to the characterizations of the next section.

We have seen that the combined complexity of order databases is Π_2^p-complete when we have binary predicates. It will emerge in this section that the restriction to monadic predicates does not suffice to reduce this to a polynomial time bound. Therefore, we investigate what further natural restrictions are required to achieve this reduction in complexity. A bound on the width of the database will be shown to be one restriction that suffices. Another, orthogonal case involves a restriction on the query: a certain class of conjunctive queries, sequential queries, will be shown to have polynomial time complexity. We will show that if either of these two parameters is relaxed then complexity one again becomes probably intractable.

Since all predicates are monadic in this section and the next, we may confine our attention to predicates in which the single argument is an order argument. This is because there can now be no interaction between order arguments and object arguments. Any conjunctive query containing only monadic proper predicates can be written in the form $\exists \mathbf{x}[\Phi_1(\mathbf{x})] \wedge \exists \mathbf{t}[\Phi_2(\mathbf{t})]$ in which the first component contains all and only those parts of the query concerning objects. More precisely, the variables $\mathbf{x}$ are all object variables, the variables $\mathbf{t}$ are all order variables, the query Φ_1 contains only proper atoms whose single argument is of type object, and the query Φ_2 contains no proper predicates with object arguments. That is, Φ_2 contains only order atoms and proper atoms with order arguments. Since the component $\exists \mathbf{x}[\Phi_1(\mathbf{x})]$ involves no order variables it does not interact with indefiniteness in the database and may be directly evaluated, in time $O(nlog\ n)$, where $n = |\Phi_1| + |D|$, against the definite proper facts in the database. Thus the main source of complexity in the query is the component $\exists \mathbf{t}[\Phi_2(\mathbf{t})]$, which contains no predicates with object arguments.

Once we discard object constants, a very useful way to understand monadic $\{<\}$-databases is as labelled versions of the dags in Section 5.2, in which we label vertices by one or more predicate symbols. If u is an order constant we will write $D[u]$ for the set of predicates P such that D contains the atom $P(u)$. All of these predicates label the corresponding vertex in the dag. Conjunctive queries Φ may similarly be interpreted as labeled dags. In this case the vertices are the order variables of Φ, and we write $\Phi[t]$ for the set of predicates P such that Φ contains the atom $P(t)$. For example, if Φ is the query

$$\exists t_1 t_2 t_3 t_4 [P(t_1) \wedge Q(t_1) \wedge P(t_2) \wedge R(t_3) \wedge S(t_4) \wedge t_1 < t_2 < t_3 \wedge t_2 < t_4]$$

then we have $\Phi[t_1] = \{P, Q\}$ and $\Phi[t_4] = \{S\}$. The dag of this query is depicted in Figure 5.8. Clearly there is a one to one correspondence between monadic databases or queries and their dag representations: we will therefore sometimes describe monadic queries and databases by presenting diagrams of their dags.

A $\{<\}$-query Φ will be said to be *sequential* if it is a conjunctive query of the form

$$\exists t_1 < t_2 < \ldots < t_n [\Psi(t_1, \ldots, t_n)]$$

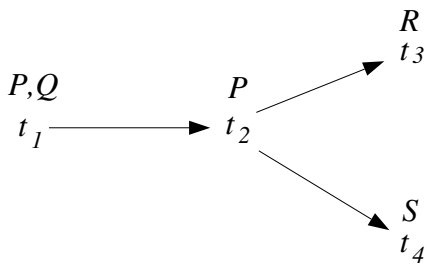

Figure 5.8: The dag associated with a query

where Ψ is quantifier free. Note that sequential queries have width one. Sequential queries and finite models of monadic databases may be perspicuously represented as words over a special alphabet. Given a set $Pred$ of monadic predicates, let the alphabet $A = \mathcal{P}(Pred)$ be the power set of $Pred$. The set A^* is the set of all finite sequences of symbols from A. Then the sequential query Φ may be represented by the word $\Phi[t_1]\Phi[t_2]\ldots\Phi[t_n]$. Conversely, to each element of A^*, there corresponds a sequential query, unique up to renaming of variables. We will switch at our convenience between the two representations. The representation of finite models as words over A^* is similar. That is, a finite model with order domain consisting of the points $u_1 < u_2 < \ldots < u_n$ corresponds to the word $D[u_1]D[u_2]\ldots D[u_n]$.

If Φ is a conjunctive monadic query then a *path* in Φ is a maximal sequential subquery of Φ. In terms of the labelled dag interpretation, a path of a query is just a maximal linear labelled subgraph of its dag. Thus the paths of the query of Figure 5.8 are the queries

$$\exists t_1 t_2 t_4 [P(t_1) \wedge Q(t_1) \wedge P(t_2) \wedge S(t_4) \wedge t_1 < t_2 < t_4]$$

and

$$\exists t_1 t_2 t_3 [P(t_1) \wedge Q(t_1) \wedge P(t_2) \wedge R(t_3) \wedge t_1 < t_2 < t_3]$$

corresponding to the words $\{P,Q\}\{P\}\{S\}$ and $\{P,Q\}\{P\}\{R\}$. We write $Paths(\Phi)$ for the subset of A^* corresponding to paths of Φ. Similarly, if D is a database then we may define $Paths(D)$ to be the set of words in A^* corresponding to the maximal linear databases contained in D.

Lemma 5.4.1: If D is a monadic $\{<\}$-database and Φ is a conjunctive monadic $\{<\}$-query then $D \models \Phi$ if and only if $D \models p$ for every path $p \in Paths(\Phi)$.

Proof: It is clear that if $D \models \Phi$ then D entails every path of Φ. For the converse, suppose that $D \models p$ for every path $p \in Paths(\Phi)$. We show by induction on the size of the query Φ that $D \models \Phi$. The base case, where Φ is the empty query, is trivial. Assume therefore that the dag of Φ has minimal vertices $t_1, \ldots, t_k$. Suppose that M is an arbitrary minimal model of D. Let u be the earliest point in M with the property that $\Phi[t_i] \subseteq M[u]$ for one of the minimal points t_i of Φ. Without loss of generality, we may assume that M supports $\Phi[t_1], \ldots, \Phi[t_l]$ at the point u, and fails to support $\Phi[t_{l+1}], \ldots, \Phi[t_k]$ there. Write the model as the word $M = M_1; a; M_2$, where the symbol $a = M[u]$ in this expression corresponds to the point u. Now let Φ' be the query constructed from Φ by deleting the quantifiers for the variables $t_1, \ldots, t_l$ as well as the atoms $\Phi[t_1], \ldots, \Phi[t_l]$. (In terms of the dag representation, we simply delete the vertices $t_1, \ldots, t_l$.) It is straightforward to see that M_2 supports every path of Φ'. Since Φ' is a query shorter than Φ, it follows by the induction hypothesis that $M_2 \models \Phi'$. Since M supports $\Phi[t_1], \ldots, \Phi[t_l]$ at u, this means that $M \models \Phi$. $\Box$

Lemma 5.4.1 shows that the problem of answering arbitrary conjunctive queries may be reduced to the problem of answering sequential queries. The next result shows that the sequential queries entailed by a database are of a rather simple form. If $p = a_1 \ldots a_n$ and $q = b_1 \ldots b_m$ are words in A^* then we will say that p is a *subword* of q if there exist indices $i_1 < i_2 < \ldots < i_n$ such that for each $j = 1 \ldots n$, the set a_j is a subset of the set b_{i_j}. For example, the word $\{P, Q\}\{P\}\{R\}$ is a subword of the word $\{P, Q, R\}\{R\}\{P, R\}\{P, Q, R\}$. If an element of a word is a singleton set then we will omit the braces. Thus, the first of the two words above will also be written as $\{P, Q\}PR$.

Lemma 5.4.2: If p is a sequential query then $D \models p$ if and only if p is a subword of some path of the database D.

Proof: It is clear that if p is a subword of some path of the database D then D entails p. Suppose therefore that p is a subword of no path in D. Let T be the set of order constants of D. We show by induction on the size of T that there exists a model of D in which the query p is false. The basis is when T is empty, in which case the result is trivial. Suppose that T is not empty, and write the word p as aq, where a is

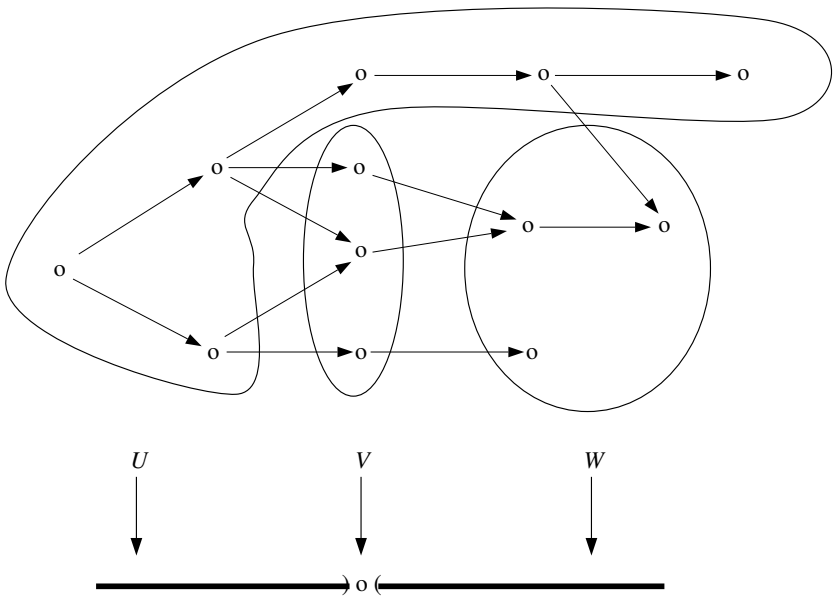

Figure 5.9: Constructing the counter-model M

the first symbol of p. Define the subsets U, V and W of T by

$$U = \{u \in T \mid \forall v \leq u (a \not\subseteq D[v])\}$$
$$V = \{u \in T \mid u \text{ is a minimal element of } T \text{ such that } a \subseteq D[v]\}$$
$$W = T \setminus (U \cup V).$$

Here the order used in the definition of U and V is the partial order associated with the dag D. Note that for each $w \in W$ there exists an element $v \in V$ with $v < w$. Figure 5.9 illustrates the possible relationships between these three sets.

Now let D_U, D_W be the restrictions of the database D to the sets of constants U, W respectively. Since no point of D_U supports a, we may construct by topological sort a minimal model M_U of D_U in which no point supports a. Since for each $w \in W$ there exists an element $v \in V$ with $v < w$, the word q cannot be a subword of any path of D_W, else p would be a subword of some path of D. Therefore, by the induction hypothesis, there exists a minimal model M_W of D_W in which the query q is false. Put

$$b = \bigcup_{v \in V} D[v].$$

Now consider the model M described by the word $M_U; b; M_W$. It is straightforward to check that M is indeed a model of D. If p is true in M, then some point supports a. By construction, this point is not in the portion M_U of M. Thus the least point at

which a holds is the point given by b. But this implies that M_W supports q, which is false by construction. This establishes that p is false in M. $\Box$

We comment that the proof actually provides a procedure which given as input a database D and a sequential query p either constructs a model of D in which p fails, or else reports that D entails p. It is readily verified that this procedure can be implemented to run in time polynomial in the size of the input. This establishes the following corollary:

Corollary 5.4.3: The combined complexity for arbitrary monadic $\{<\}$-databases and sequential $\{<\}$-queries is in polynomial time.

The exact complexity depends on what assumptions we make about the set of predicates. If this set is fixed then the procedure may be implemented to run in time $O(n)$ on a random access machine; if it is permitted to grow arbitrarily large (as is the case if constants may occur the query and we employ the constant elimination reduction, see p. 123), then the best we can ensure is $O(nlog\ n)$. Let us first note that simply performing a topological sort of a dag can be done in linear time. The main difficulty here is to be able to efficiently calculate after each vertex deletion the new set of minimal vertices. The standard trick for this (see for example [56]) is to associate with each vertex v the number of vertices u such that there exists a edge from u to v. Whenever we delete a vertex, we decrement the count of all of its successors. Should this operation decrease the count associated with a vertex to zero, this vertex is added to the list of current minimal elements. This results in a linear time topological sort. To implement our procedure, we use two lists of minimal elements instead of one. The first of these lists, L_a, contains the minimal vertices u such that $a \subseteq D[u]$, the other, $L_{\overline{a}}$, contains the minimal vertices such that $a \not\subseteq D[u]$, where a is the current symbol of the word p being processed. The procedure then performs the topological sort which selects a single vertex from $L_{\overline{a}}$ whenever possible, and selects all elements from the list L_a simultaneously when $L_{\overline{a}}$ is empty. The main contribution to complexity, then, is how to decide whether a vertex should go onto L_a or $L_{\overline{a}}$. If the set of predicates is fixed, this can be done in constant time, after a preliminary linear time pass over the data. If not, it is possible to use standard methods from sorting algorithms to set up

in time $O(nlog\ n)$ a data structure which will permit this test to be carried out in time $O(log\ n)$, giving an algorithm with total cost $O(nlog\ n)$. We note that set containment is known to require at least $nlog\ n$ comparisons [107], so this bound is in some sense optimal.

By Lemma 5.4.1 a database entails a conjunctive query Φ just in case it entails each path of Φ. Combining this with Lemma 5.4.2, we obtain the following characterization of entailment of conjunctive queries: D entails Φ just in case every path of Φ is a subword of some path of D. If the query Φ is fixed, so is its set of paths, so Corollary 5.4.3 yields:

Corollary 5.4.4: The data complexity of conjunctive monadic $\{<\}$-queries with respect to monadic $\{<\}$-databases is in linear time.

Note, however, that the number of paths of a query can grow exponentially in the size of the query, so the constant of proportionality may be very large. It also follows from this that an algorithm which tests entailment of each path of a query will not run in polynomial time if the size of the query is permitted to grow. This suggests that the combined complexity when queries are non-sequential may be high. This is indeed the case:

Theorem 5.4.5: The combined complexity of $\{<\}$-databases and width two conjunctive $\{<\}$-queries over a fixed set of two monadic predicates is co-NP hard.

Proof: We use a reduction from the problem of determining if a formula in disjunctive normal form is a tautology. Since this is the complement of determining if a conjunctive normal form formula is satisfiable, this is a co-NP hard problem. Suppose that $\alpha = \bigvee \delta_i$ is a disjunctive normal form formula over the propositional constants $P_1, \ldots, P_m$ where the δ_i are conjunctions of literals. We fix the set $\{T, F\}$ of predicates. All paths in the database or the query will be words of length m over the alphabet $\{T, F\}$, and correspond to valuations of the set of propositional constants. Define $\Phi(\alpha)$ to be the query whose dag is depicted in Figure 5.10. Here there are two rows, each containing m vertices. Thus the query $\Phi(\alpha)$ has width two. Clearly $Paths(\Phi) = \{T, F\}^m$ is the set of all words of length m over $\{T, F\}$.

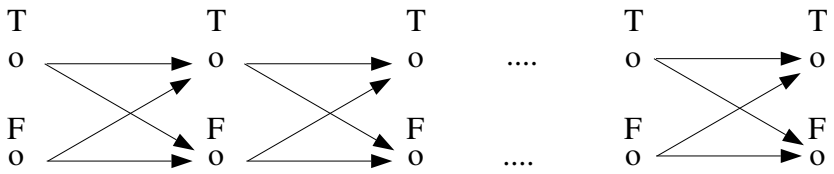

Figure 5.10: The query $\Phi(\alpha)$.

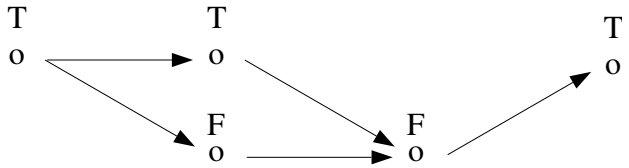

Figure 5.11: The graph of the conjunction $P_1 \wedge \overline{P_3} \wedge P_4$.

The database $D(\alpha)$ corresponding to the formula α will be the union of a number of disconnected components, one for each disjunct δ_i. Each component will be isomorphic to the subgraph of the graph of $\Phi(\alpha)$ generated by set of nodes chosen as follows. For each $j = 1 \ldots m$, if the disjunct contains neither the literal P_j nor the literal $\overline{P_j}$ then we retain both vertices in the j-th column of $\Phi(\alpha)$. If the disjunct contains the literal P_j then we retain from the j-th column of $\Phi(\alpha)$ only the vertex labelled T. Finally, if the disjunct contains the literal $\overline{P_j}$ then we retain from the j-th column of $\Phi(\alpha)$ only the vertex labelled F. (We assume that each disjunct is consistent.) For example, the graph of the component corresponding to the conjunction $P_1 \wedge \overline{P_3} \wedge P_4$ is illustrated in Figure 5.11. We let $D(\alpha)$ be the disjoint union of the components corresponding to the disjuncts δ_i. Note that all paths of $D(\alpha)$ have length m. It is readily seen that a word is a path of $D(\alpha)$ just in case α is true under the corresponding valuation of the propositional constants.

Since the paths of $D(\alpha)$ and $\Phi(\alpha)$ all have length m, a path of $\Phi(\alpha)$ is a subword of a path of $D(\alpha)$ just in case it is in fact a path of $D(\alpha)$. Thus $D(\alpha)$ entails $\Phi(\alpha)$ if and only if every word in $\{T, F\}^m$ is a path of $D(\alpha)$. Interpreted in terms of valuations, this holds exactly when α is true under every valuation, that is, when α is a tautology. □

It will be shown in the next section (Proposition 5.5.1), that this lower bound is the best possible, and that even disjunctive monadic queries have combined complexity

in co-NP. Notice that the databases $D(\alpha)$ constructed in this proof may grow to have arbitrary width, since the formula α may have an arbitrary number of disjuncts. We have seen in Section 5.1 some applications, like databases recording the reports of a fixed number of observers, in which it is natural to assume a width bound. Hence, it is natural to ask whether constraining the database to have bounded width results in a decrease in complexity. The following result shows this to be the case. We will actually prove a result in the next section which subsumes this, showing that even certain classes of disjunctive queries have low complexity in databases of bounded width. However, the construction required for that result is a slightly more complex than that for the conjunctive case, so we present the present proof in order to have a clearer presentation of the role of width bound.

Theorem 5.4.6: For each k, the combined complexity of $\{<\}$-databases of width less than or equal to k and conjunctive $\{<\}$-queries, over an infinite set of monadic predicates, is in polynomial time.

Proof: We reduce the problem of determining $D \models \Phi$ to the problem of checking if the language accepted by a non-deterministic finite state automaton is empty. Let $A(\Phi)$ be the collection of sets of predicates of the form $\Phi[t]$ for some variable t of Φ. We use this alphabet rather than the power set of the set of predicates which occur in the query because the latter may be of exponential size. We associate with the database D and with the query Φ two languages, both over the alphabet $A(\Phi)$. The language $L(\Phi)$ associated with the query will be simply the set $Paths(\Phi)$ of paths of the query. The language associated with the database is given by

$$L(D) = \{p \in A(\Phi)^* \mid p \text{ is a subword of some path of } D\}.$$

Note that this set does not contain all subwords of paths in D, only those subwords whose symbols are all in the set $A(\Phi)$. By Lemma 5.4.1 and Lemma 5.4.2, the database D entails Φ if and only if $L(\Phi)$ is a subset of $L(D)$, or equivalently, if the language $L(\Phi) \cap \overline{L(D)}$ is empty.

It is straightforward to construct a nondeterministic finite state automaton accepting $L(\Phi)$, which is linear in the size of Φ. The states of this automaton are either the

start state or one of the vertices of the graph of Φ, and there is a transition from a state u to a state v labelled a if either

1. u is the start state and v is a vertex of the graph of Φ which is minimal and has $\Phi[v] = a$, or

2. u and v are vertices of Φ, there is an edge from u to v and $\Phi[v] = a$.

The final states of the automaton are the maximal vertices of Φ. We now show that it is possible to construct a *deterministic* finite state automaton, polynomial in the size of D, which accepts $L(D)$. Because this automaton is deterministic, it may be complemented without blow-up. Hence we obtain using the product construction a polynomial size non-deterministic automaton accepting $L(\Phi) \cap \overline{L(D)}$. This may be checked for emptiness in polynomial time. (We refer the reader to [55] for details on these standard results on finite state automata.)

The automaton for $L(D)$ is constructed as follows. Let k be the bound on the width of the databases D. Augment the dag of D with an additional vertex u_0, and add an edge from this vertex to each minimal vertex of D. Call the resulting dag D', and let U be the set of vertices of this dag. The states of the automaton will be the *nonempty* subsets S of U such that $|S| \leq k$ and there do not exist vertices $x, y \in S$ with $x < y$ in the order on D'. The initial state is the set $\{u_0\}$, and every state is final. If S, T are two states then there is a transition from S to T labelled a just in case $T = a(S)$, where

$$\begin{aligned} a(S) = \{u \in U \mid u \text{ is a miminal point in } D' \text{ such that} \\ a \subseteq D'[u] \text{ and } \exists v \in S(v<u)\} \end{aligned} \tag{5.2}$$

Note that for any state S and symbol a the right hand side of this equation is either empty or another state of the automaton, because if D has width k then the set of minimal vertices satisfying any property always has cardinality at most k. Clearly the automaton has size $O\left(|D|^{2k}.|\Phi|\right)$.

We now verify that the automaton accepts $L(D)$. First, suppose that $p = a_1 \ldots a_n \in L(D)$. Then there exists a sequence $u_1 < \ldots < u_n$ of vertices of D with $a_i \subseteq D[u_i]$. This implies that the sequence $S_0 \ldots S_n$ defined by $S_0 = \{u_0\}$ and $S_i = a_i(S_{i-1})$ for

$i = 1 \ldots n$ consists of non-empty sets, which are states of the automata by the remark above. Since all states are final, the automaton accepts p. Conversely, suppose $S_0 \ldots S_n$ is a sequence of states with $S_0 = \{u_0\}$ and $S_i = a_i(S_{i-1})$ for $i = 1 \ldots n$. Choose some vertex $u_n \in S_n$. By equation (5.2), we have that $a_n \subseteq D[u_n]$. Furthermore, there exists a vertex $u_{n-1} \in S_{n-1}$ such that $u_{n-1} < u_n$. Continuing this argument yields a sequence of vertices witnessing the fact that $a_1 \ldots a_n$ is a subword of some path of D. This completes the proof that the automaton accepts $L(D)$. □

As a special case of this result, consider a fixed database D, which clearly has bounded width. We obtain as a corollary of Theorem 5.4.6 the following characterization of the expression complexity of monadic databases.

Corollary 5.4.7: Every $\{<\}$-database D has polynomial time expression complexity with respect to conjunctive $\{<\}$-queries.

5.5 Disjunctive Monadic Queries

We consider in this section the case of disjunctive queries in monadic order databases. Disjunction turns out to introduce some extra complications in querying monadic order databases. We have seen that even for conjunctive queries it is necessary to introduce one of two restrictions in order to obtain PTIME combined complexity: a constant bound on the width of the database, or the restriction that the query be sequential. We show in this section that these restrictions are not sufficient for PTIME complexity once one permits disjunctive queries. However, in the case of bounded width databases we are able to save the PTIME result by imposing another reasonable restriction, this time on queries, that bounds the number of disjuncts. We also show in this section that data complexity and expression complexity remain in PTIME for disjunctive monadic queries. The proof for data complexity is much more complex than in the conjunctive case, and far less satisfactory since it is non-constructive. While it shows that disjunctive queries have PTIME data complexity, the proof does not explicitly tell us how to answer disjunctive queries so as to achieve this bound. Thus, we see in this section two senses in which disjunction complicates the complexity picture.

We begin by noting an upper bound on the combined complexity of monadic disjunctive queries, which improves the upper bound found in the binary case. To determine $D \models \Phi$ it suffices to verify Φ in each of the minimal models of D. Each of these models has width one, so by Theorem 5.4.6, this can be done in polynomial time. Thus we have

Proposition 5.5.1: The combined complexity of monadic $\{<\}$-databases and disjunctive queries is in co-NP.

We have already seen in Theorem 5.4.5 that the matching lower bound may be achieved using conjunctive queries of width 2. However, we also noted that sequential queries are particularly simple to evaluate (Corollary 5.4.3). A natural question is whether disjunctions of sequential queries may have low complexity. We now show that this is not the case.

If $p = a_1 \ldots a_n$ and $q = b_1 \ldots b_m$ are words over some alphabet then p will be said to be a *subsequence* of q if there exist indices $i_1 < i_2 < \ldots < i_n$ such that $a_j = b_{i_j}$ for each $j = 1 \ldots n$. (Note that the difference between this and the notion of *subword* of the previous section is that there we were dealing with words over the power set of an alphabet.) The *Shortest Common Supersequence* problem is: given an integer k and a set S of words, is there a word of length less than k which is a supersequence of each word in S. It has been shown by Maier [79] that this problem is NP complete even over a fixed alphabet of five symbols.

Proposition 5.5.2: The combined complexity of bounded disjunctions of sequential $\{<\}$-queries and arbitrary $\{<\}$-databases over a fixed set of six monadic predicates is co-NP hard.

Proof: By reduction from Shortest Common Supersequence. We assume a fixed alphabet $A = \{c_1, \ldots, c_5\}$. For a word $p = a_1 \ldots a_n$ on A let $D(p)$ be the linear database represented by the word $\{a_1, d\}, \ldots, \{a_n, d\}$ (using the correspondence from the previous section). Given a set of words S over A, let $D(S)$ be the database obtained by taking the disjoint union of the databases $D(p)$ for p in S. For each natural number k, define the query $\Phi(k)$ to be $false$ if the sum of the lengths of the words S is less than

k, and otherwise the disjunction of the queries with word representation $\{c_i, c_j\}$, for all $i \neq j$, with the query with word representation $dd \ldots d$ in which there are k occurrences of d. Then $D(S)$ entails $\Phi(k)$ if and only if there exists no word of length less than k which is a supersequence of every word in S. This is clear if the words S have total length less than k. In the remaining case we argue as follows. If such a word $a_1 \ldots a_{k-1}$ exists, then the structure described by the word $\{a_1, d\} \ldots \{a_{k-1}, d\}$ is a model of $D(S)$ which does not support $\Phi(k)$. Conversely, suppose M is a minimal model of $D(S)$ which does not support $\Phi(k)$. In particular M does not support any disjunct $\{c_i, c_j\}$, so no two nodes of $D(S)$ with distinct labels map to the same point of M. Thus M is of the form $\{a_1, d\}, \ldots, \{a_n, d\}$ for some n, and therefore a supersequence of each word in S. Since M also does not support the disjunct $dd \ldots d$ we must have $n < k$. That is, there exists a supersequence of every word in S of length less than k. $\square$

Notice that this proof required databases of unbounded width. We have seen in the case of conjunctive queries that a width bound suffices to obtain PTIME combined complexity. We now once again attempt to find a polynomial time version of this problem by bounding the width of databases. As in Theorem 5.4.6, we use finite state automata, but we require a different reduction, since we can no longer rely on Lemmas 5.4.1 and 5.4.2. It turns out that we need an additional constraint in order to obtain a polynomial time problem, namely, a bound on the number of disjuncts in the query.

It is convenient for the proof to introduce a new type of finite state automaton, which operates over the power set alphabet $\mathcal{P}(A)$, and which we will refer to as *power automata.* These are similar to standard finite state automata, except that the transitions are labelled with positive labels a or negative labels $\overline{a}$, where in either case $a \in \mathcal{P}(A)$. Given as input a symbol $b \in \mathcal{P}(A)$, we may make a transition along an edge labelled a just in case $a \subseteq b$, and along an edge labelled $\overline{a}$ just in case $a \not\subseteq b$. As usual, we have a set of starting states and a set of final states, and the automaton accepts a string p in $\mathcal{P}(A)^*$ if there exists a valid sequence of transitions on p from a starting state to a final state.

We now show that power automata are 'no more expressive' than ordinary automata,

by establishing a simulation of power automata by classical finite state automata. Suppose that the alphabet A is the set $\{1 \ldots n\}$, and let B be the extended alphabet $A \cup \{\#\}$. Given a language $L \subseteq \mathcal{P}(A)^*$ over the power alphabet, define the *flattened* representation $\Delta(L) \subseteq B^*$ to be the set of strings of the form

$$x_{11} \ldots x_{1n_1} \# x_{21} \ldots x_{2n_2} \# \ldots \# x_{m1} \ldots x_{mn_m} \#$$

for which the x_{ij} are elements of A with $x_{i1} < x_{i2} < \ldots < x_{in_i}$ for each i, and such that there exists a string $a_1 \ldots a_m \in L$ with $a_i = \{x_{i1}, \ldots, x_{in_i}\}$.

Lemma 5.5.3: For each power automaton accepting a language $L \subseteq \mathcal{P}(A)^*$ there exists a polynomial size finite state automaton accepting $\Delta(L)$.

To see this, note that Δ is the homomorphism of languages which acts by mapping each symbol $a = \{x_1, x_2, \ldots, x_k\}$ in $\mathcal{P}(A)$ to $\Delta(a) = x_1 x_2 \ldots x_k \#$ where $x_1 < x_2 < \ldots < x_k$. Thus we may obtain an automaton for $\Delta(L)$ by replacing each transition in the power automaton by a small automaton with a single initial state and a single final state and which accepts a word w on A just in case $w = \Delta(a)$ for some symbol a permitted by the transition. That is, we replace a transition labelled by a with an automaton for the language

$$(SORTED(A) \cap (A^*.x_1.A^*.x_2.A^* \ldots x_k.A^*)).\#$$

where $SORTED(A)$ is the language on A containing all words $a_1 a_2 \ldots a_m$ in A^* such that $a_1 < a_2 < \ldots < a_m$. Similarly, each transition labelled $\overline{a}$ is replaced by an automaton for the language

$$(SORTED(A) \setminus (A^*.x_1.A^*.x_2.A^* \ldots x_k.A^*)).\#$$

It is straightforward to check that these automata have size $O(n^2)$, where n is the number of letters in A.

Proposition 5.5.4: The combined complexity of bounded monadic $\{<\}$-databases and $\{<\}$-queries with a bounded number of disjuncts is in polynomial time.

Proof: Let $Pred$ be the set of predicates. We associate with the database D and the query Φ languages $L(D)$ and $L(\Phi)$ respectively, over the alphabet $\mathcal{P}(Pred)$. The

language $L(D)$ will be the set of words over $\mathcal{P}(Pred)$ corresponding to finite models of D, that is, $Mod_{\mathbf{Fin}}(D)$. Similarly, the language $L(\Phi)$ will be the set of words corresponding to finite models supporting Φ. Clearly $D \models \Phi$ exactly when $L(D) \subseteq L(\Phi)$. (Note that we have in this proof the reverse of the containment used in Theorem 5.4.6.) We show the following. First, for any bounded database D the language $L(D)$ is accepted by a power automaton of size polynomial in D. Second, for any query Φ, the complement of the language $L(\Phi)$ is accepted by a power automaton of size polynomial in Φ. Since it is clear that $\Delta(L_1 \cap L_2) = \Delta(L_1) \cap \Delta(L_2)$ and $\Delta(L)$ is empty if and only if L is empty, it follows from Lemma 5.5.3 that we may determine $D \models \Phi$ in polynomial time by testing $\Delta[L(D)] \cap \Delta\left[\overline{L(\Phi)}\right]$ for emptiness.

The power automaton for $L(D)$ is constructed as follows. Let U be the set of vertices of D. We suppose that k is the bound on the width of D. States of the automaton will be *all* independent subsets of U of cardinality k or less. The initial state is the set of minimal elements, and the final state is the empty set. Intuitively, the automaton will be in state S if the portion of D strictly less than the nodes in S has been observed. Thus, we have a transition from a state S to a state T labelled with $a \in \mathcal{P}(Pred)$ in the following circumstances: First, there exists a subset S' of S with

$$a = \bigcup_{u \in S'} D[u]$$

Second, T is the set of vertices obtained by deleting from S the vertices S', adding all vertices of D which are minimal elements of D greater than one of the deleted vertices, and then taking the set of minimal elements of the result. In addition to these transitions, we have a transition from each state to itself, labelled by the empty set. It is straightforward to verify that this automaton accepts $L(D)$.

We now describe the power automaton for the complement of $L(\Phi)$. Since the operator Δ preserves intersection it suffices to do this for the case where Φ is a conjunctive query. The set of states of the automaton for $\overline{L(\Phi)}$ consists of the vertices of the graph of Φ together with a new vertex u' for each vertex u of Φ. The starting states are the vertices u' for u a minimal vertex of Φ. The transitions are as follows:

1. For each edge from v to u in Φ there is an ϵ-transition from the state v to the

state u'.

2. If u is labelled a in Φ,there is a transition from the state u' to the state u with the positive label a.

3. If u is labelled a in Φ,there is a transition from the state u' to itself with the negative label $\overline{a}$.

All states except those corresponding to maximal states u in Φ are final. Intuitively, the meaning of this construction is as follows: there exists a computation of the automaton on input M which terminates in a state arising from a vertex v just in case M supports some path of Φ up to the vertex v, but no further. More formally, we now establish: the automaton accepts M if and only if $M \not\models p$ for some path p of Φ. Since $M \models \Phi$ exactly when M supports every path of Φ, this will complete the proof.

First, suppose $p = a_1 a_2 \ldots a_n$ is a path in Φ corresponding to the maximal chain $u_1 u_2 \ldots u_n$ in the graph of Φ, such that $M \not\models p$. Consider the sequence of states $u'_1, u_1, u'_2, u_2, \ldots, u'_n, u_n$ in which every state except the last is final. It follows from the fact that $M \not\models p$ that there exists a computation of the automaton on input M which terminates in a final state. Thus the string M is accepted. Conversely, suppose that M is accepted. Observing that the only cycles in the automaton are the self-loops on the u'_k, we obtain that there exists a computation on M of the form $u'_1, u_1, u'_2, u_2, \ldots, x$, where

1. x is either of the form u'_k or a non-maximal vertex u_k, and

2. the maximal subsequence $u_1 u_2 \ldots u_k$ is a chain in the graph of Φ.

By the second condition we may extend the sequence $u_1 u_2 \ldots u_k$ to a maximal chain $u_1 u_2 \ldots u_n$ in Φ. Let p be the path corresponding to this chain. Since the computation terminates before the maximal vertex, we must have $M \not\models p$, and therefore $M \not\models \Phi$ also. □

We emphasize that this result does not require a bound on the width of each disjunct of the query. This result probably cannot be generalized to queries containing an arbitrary number of disjuncts, as the next result shows.

Proposition 5.5.5: The combined complexity of bounded $\{<\}$-databases over four monadic predicates and $\{<\}$-queries with an unbounded number of sequential disjuncts is co-NP hard.

Proof: By reduction from the tautology problem for disjunctive normal form formulae. We use the monadic predicates T, F, E and $*$, corresponding to *true*, *false*, *either* truth value and a special marker respectively. For a disjunctive normal form containing n propositional variables, the database D will consist of three disjoint linear sequences corresponding to the words $\{T,E\}\{T,E\}\ldots\{T,E\}$ and $\{F,E\}\{F,E\}\ldots\{F,E\}$ (both of length n) and the point $*$. The intention is that models consisting of a sequence of n truth values before the $*$ will correspond to valuations of the n propositional variables. In order to restrict attention to models of this structure, we include in the query Φ the following disjuncts which detect models not of this form. First, to prevent points having two distinct truth values we have a disjunct given by the word $\{T,F\}$. Next, we want all points with truth values to be either before or after the $*$, so we include the disjunct $\{*,E\}$. To ensure that there are at least n points with truth values to the left of the $*$ we exclude models which have $n+1$ or more truth values to the right of the $*$ by adding the disjunct $*EE\ldots E$ in which there are $n+1$ E's. Finally, to ensure that there are not more than n truth values to the left of the $*$ we add the disjunct $EE\ldots E*$ with $n+1$ E's. The models which do not satisfy any of these disjuncts now correspond to propositional valuations. Now, given a disjunct α of a propositional disjunctive normal form formula, the query Φ contains the disjunct $V_1V_2\ldots V_n$ in which V_i is T if the i-th propositional variable occurs positively in α, F if it occurs negatively, and E if it does not occur. It is now straightforward to verify that $D \models \Phi$ exactly when the conjunctive normal form is a tautology. $\Box$

This ends our discussion of combined complexity for disjunctive queries. We have considered three parameters in all: sequentiality of disjuncts, boundedness of disjuncts and boundedness of the database. It may benefit the reader in understanding the relation between our various upper and lower bound results to consult Figure 5.5, which demonstrates that we have now given a complete characterization of the effect of these parameters on combined complexity.

We now turn our attention to expression and data complexity for disjunctive monadic queries. The former is readily disposed of. To determine whether a fixed database entails a disjunctive query involves testing the query in each of a fixed set of models. By Lemma 5.4.7 each disjunct may be tested in polynomial time. Thus we have:

Proposition 5.5.6: The expression complexity of $\{<\}$-databases with respect to disjunctive monadic queries is in PTIME.

The data complexity of monadic disjunctive queries also turns out to be in polynomial time, but we will need some more sophisticated techniques to show this. We will be able to prove that each query has data complexity in PTIME, but without being able to provide an explicit algorithm that works with this complexity. The proof relies on the following concept.

Definition 5.5.7: A *quasi-order* on a set X is a reflexive transitive relation $\leq$. The pair $(X, \leq)$ is said to be a *well-quasi-order* if every nonempty subset S of X has a non-empty finite set of (inequivalent) minimal elements.

An equivalent definition is as follows: a set is well-quasi-ordered just in case (a) every strictly decreasing sequence is finite and (b) every set of pairwise incomparable elements is finite. The theory of well-quasi-orders is well developed: see Kruskal [68] for an historical survey. If X is a quasi-ordered set then we may define a quasi-order on the finite power set $\mathcal{FP}(X)$ by $S_1 \leq S_2$ when for each $x \in S_1$ there exists $y \in S_2$ with $x \leq y$. Similarly, the set X^* of all finite sequences from X may be quasi-ordered by $x_1 \ldots x_n \leq y_1 \ldots y_m$ if there exists a sequence of indices $i_1 < i_2 < \ldots < i_n$ with $x_j \leq y_{i_j}$ for $j = 1 \ldots n$. It is known that if X is well-quasi-ordered then so are $\mathcal{FP}(X)$ and X^*. We will also use the following lemma, whose proof is straightforward.

Lemma 5.5.8: Let X be a well-quasi-ordered set. Suppose Y is any set and let f be a function from Y to X. Then the relation $\leq$ induced on Y by $x \leq y$ when $f(x) \leq f(y)$ is a well-quasi-order.

We now construct a well-quasi-order on the set $\mathcal{M}$ of monadic $\{<\}$-databases. Fix a finite set $Pred$ of monadic predicates and let the alphabet $A = \mathcal{P}(Pred)$ be ordered by containment. Then A^* is well-quasi-ordered, and the order is exactly the subword

relation of the previous section. It follows that the set $\mathcal{FP}(A^*)$ is also well-quasi-ordered. By Lemma 5.5.8 we obtain a well-quasi-order on $\mathcal{M}$ defined by $D_1 \sqsubseteq D_2$ when $Paths(D_1) \leq Paths(D_2)$.

Lemma 5.5.9: For any disjunctive monadic query Φ, if $D_1 \models \Phi$ and $D_1 \sqsubseteq D_2$ then $D_2 \models \Phi$.

Proof: Recall that we can treat models as words in A^* because we have eliminated constants from queries. More formally, we have a mapping $\omega : Mod_{\mathbf{Fin}} \rightarrow A^*$ which maps finite models over monadic predicates to words, simply by "forgetting" the part of the model that interprets constants. Since queries do not contain constants the evaluation of a query Φ in a model M does not make use of the interpretation of constants, so we have $M \models \Phi$ if and only if $\omega(M) \models \Phi$, where we treat the word $\omega(M)$ as a model without interpreted constants. If D is a database then the set of words $Paths(D)$ may also be interpreted as a database which contains a distinct linear sequence for each word in the set. It follows by an argument exactly along the lines of Lemma 5.4.1 that $\omega(Mod_{\mathbf{Fin}}(D)) = \omega(Mod_{\mathbf{Fin}}(Paths(D)))$ for any database D. Clearly if $Paths(D_1) \leq Paths(D_2)$ then $\omega[Mod_{\mathbf{Fin}}(Paths(D_2))] \subseteq \omega[Mod_{\mathbf{Fin}}(Paths(D_1))]$, so that $\omega[Mod_{\mathbf{Fin}}(D_2)] \subseteq \omega[Mod_{\mathbf{Fin}}(D_1)]$. Now notice that if $D_1 \models \Phi$ then $M \models \Phi$ for all $M \in Mod_{\mathbf{Fin}}(D)$, which holds just in case $p \models \Phi$ for all $p \in \omega[Mod_{\mathbf{Fin}}(D_1)]$. Clearly this implies that $p \models \Phi$ for all $p \in \omega[Mod_{\mathbf{Fin}}(D_2)]$, which implies that $D_2 \models \Phi$ also. $\Box$

Lemma 5.5.9 shows that for any fixed disjunctive query Φ, the set $S(\Phi)$ of monadic databases D satisfying $D \models \Phi$ is an ideal, that is, upwards closed. Thus to show that $D_2 \models \Phi$ it suffices to show that $D_1 \sqsubseteq D_2$ for some minimal element $D_1 \in S(\Phi)$. By the well-quasi-order property, these minimal elements are finite in number. For any fixed D_1 we may determine $D_1 \sqsubseteq D_2$ in time linear in the size of D_2, as shown by Corollary 5.4.4. Thus we have

Theorem 5.5.10: The data complexity of monadic disjunctive $\{<\}$-queries on $\{<\}$-databases is in linear time.

Notice that this argument does not provide us with an explicit algorithm, since we

do not as yet know how to calculate for a query Φ the set of minimal elements of $S(\Phi)$, only that this set must be finite. Thus, the proof is non-constructive. Although we have established a linear time upper bound, this result is of no practical significance until an alternate constructive proof can be found, or further analysis makes the present proof constructive. We also warn once again that the fact that combined complexity is co-NP hard indicates that the constants of proportionality may be very large: one expects that the number of minimal elements of the sets $S(\Phi)$ will be exponential in the size of the query Φ.

A number of other examples of non-constructive proofs that a set is in PTIME are known, see [35, 36]. In general such proofs are all based on well-quasi-orders. Further analysis of the structure of the order $\sqsubseteq$ and the sets $S(\Phi)$ is required to obtain for each query an explicit algorithm running in polynomial time. The possibility remains that there is no algorithm that will produce for each query Φ a set of minimal elements of $S(\Phi)$. We conjecture that this is not the case, but we have not yet been able to provide this algorithm.

There is one case in which we do know how to compute the set of minimal elements of $S(\Phi)$, and that is when the query Φ is conjunctive. Lemma 5.4.1 and Lemma 5.4.2 together show that a conjunctive monadic query Φ is entailed by a database D just when $Paths(\Phi) \leq Paths(D)$: this was the basis of our earlier result that the data complexity of conjunctive monadic queries is in PTIME (Corollary 5.4.4). Thus, if we take D_Φ to be the database with the same labelled graph representation as the query Φ, then we see that $D \models \Phi$ if and only if $D_\Phi \sqsubseteq D$. This means that the set $S(\Phi)$ has unique minimal element D_Φ, which is straightforwardly computable. We see that not only does Theorem 5.5.10 subsume Corollary 5.4.4, but the proof we gave for the latter is also a special case of the proof of the former. We do not know any other cases in which it is possible to compute the set of minimal of elements of $S(\Phi)$.

5.6 Discussion

There is an extensive literature on temporal databases and logical representations of time that is superficially related to our work. In this concluding section we briefly

mention some of this, and also point out a connection between the problems we have considered and certain formulations of non-linear planning in Artificial Intelligence.

In general, work on temporal databases and logical representations of time is distinct from ours for one of two reasons: either it is concerned exclusively with definite information, or else it deals with modal, rather than first order languages. On the side of databases storing definite information involving linearly ordered domains, there is a considerable literature on temporal query languages and temporal extensions of the relational algebra: see the bibliography [119]. Repeated activity is considered by Chomicki and Imielinski [18, 17], who study data complexity for a "temporal" logic programming language which expresses "next" using a monadic function symbol, constrained to occur in a single argument of each predicate. That is, they deal with predicates with a single "temporal" argument. The presence of rules generating information at the "next" point in time means that their databases represent an infinite set of tuples, and they study the representation of infinite answers. Kanellakis et al [62] study data complexity for a type of definite database in which constraints occur as conditions on universally quantified tuples. That is, a tuple in their model represents an infinite set of classical tuples, those satisfying the constraints. They study conditions under which Datalog programs may be efficiently evaluated in a bottom up fashion in this context. A similar model is studied by Kabanza et al [60] and Baudinet et al [7]. Kowalski and Sergot's Event Calculus [66] provides a logic programming representation for temporal projection in an event-based model of time.

For summaries of the voluminous literature on modal logics of time we refer the reader to any of the numerous collections on this topic, such as [40]. There is an ongoing controversy about the "right" representation of time, and continuing analysis of the effect on decidability, axiomatizability, compositionality and expressibility of various assumptions concerning the set of primitive operators, whether time is linear or branching, dense or discrete, etc. Since we do not deal with modality in this dissertation, we will not attempt to describe this work here. One point is worth making, however, concerning the jump in complexity in the move from point to interval based representations, mentioned above in our discussion of the point algebra and the interval

algebra, and borne out in our results about the complexity of queries involving monadic versus binary predicates. This jump appears also to occur in the context of modal logics. Halpern and Shoham [49] introduce a temporal modal logic based on intervals and show that it is very expressive, and has an undecidable validity problem. Point based temporal logics with comparable modalities are generally decidable.

Related to the literature on temporal logics is the work on modal logics of programs. We will defer our discussion of this to the next chapter, where we discuss how the combination of defined relations and linear order constraints is related to existing representations for reasoning about recursive actions.

More closely related to the problems studied in this chapter is the Artificial Intelligence literature on reasoning about partially ordered actions, motivated by non-linear planning. The problem of *temporal projection* is to determine, for a partially ordered set of actions, whether a proposition necessarily holds directly after the occurrence of an action in all linear sequences of events compatible with the partial order. In general, the model of action used in the non-linear planning literature is STRIPS operators [38]. These are operators whose effect is given by a precondition list (that is, a set of propositions) φ, an addition list α and a delete list δ. When the event occurs in a state satisfying the precondition, its effect is to add the propositions α and delete the propositions δ. Chapman [15] has considered the problem of temporal projection in partially ordered event structures for a model of event slightly more general than STRIPS operators, since he deals with events which may contain variables constrained by inequalities of the form $x \neq y$. Chapman formulates a *modal truth condition* which is equivalent to this property, and shows that this condition may be evaluated in polynomial time.

Dean and Boddy [27] have studied the complexity of temporal projection in partially ordered event structures for a somewhat more general type of event. Instead of a single precondition φ, an addition list α and a delete list δ, these events have associated with them a *set* of tuples $\langle \varphi, \alpha, \delta \rangle$. The effect of an occurrence of one of these events is to add to the state α and delete δ for *each* tuple $\langle \varphi, \alpha, \delta \rangle$ associated with the event type which has its precondition φ satisfied. In case of a conflict between an addition and deletion, the addition is performed. Dean and Boddy consider the complexity of

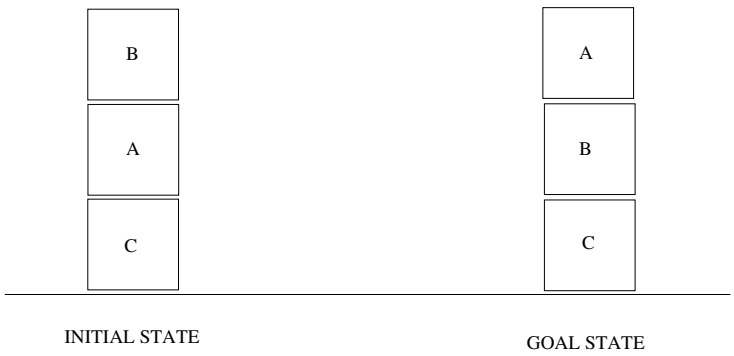

Figure 5.12: Allen and Koomens' planning problem

determining what propositions hold after a particular event (1) in all sequences of the events satisfying the precedence constraints, and (2) in some such sequence. They show that given a description of the event types and a partially ordered set of events, the latter problems is NP hard, and go on to consider approximation algorithms.

The problem of temporal projection through STRIPS-like operators is apparently more complex than the problem of querying indefinite order databases we consider in this chapter, since indefinite order databases do not have any way to express conditional events or the default persistence of facts. However, there is an alternate representation of planning problems that yields exactly the sort of problem we study. This is the proposal by Allen and Koomen [3] to use logics based on the interval algebra to represent action and plans. In this proposal, both actions and propositions are treated as predicates holding over intervals. Consider the planning problem depicted in Figure 5.12. Allen and Koomen formulate the initial state of this problem by asserting that the fact $On(A, B)$ holds over interval I_1, written $On(B, A) : I_1$, that $On(A, C) : I_2$ and that $Clear(B) : I_3$ where the intervals I_1, I_2 I_3 are constrained to contain the initial interval I. The desired final state is similarly described, and is constrained to lie after all of these initial intervals. In terms of the endpoints, this yields the partially ordered structure in Figure 5.13.

Actions are also represented as facts holding over intervals, and are connected to

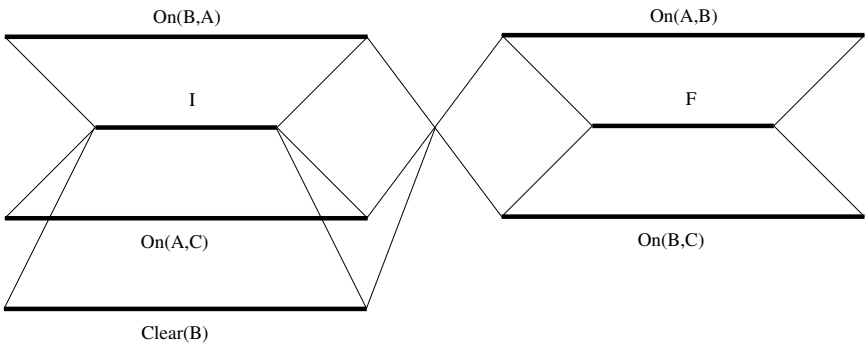

Figure 5.13: Allen and Koomens' planning problem

their preconditions and effects by means of axioms such as the following:

$$\textbf{Move}(x,t,f) : M \Rightarrow$$

$$\textit{preconditions} : \begin{cases} \exists C_x[Clear(x) : C_x \wedge C_x m M] \wedge \\ \exists O_{xf}[On(x,f) : O_{xf} \wedge O_{xf} o M] \wedge \\ \exists C_t[Clear(t) : C_t \wedge C_t \{o,s,d\} M] \wedge \end{cases}$$

$$\textit{effects} : \begin{cases} \exists C_f[Clear(f) : C_f \wedge M\{o,fi,di\}C_f] \wedge \\ \exists C_{x2}[Clear(x) : C_{x2} \wedge M m C_{x2}] \wedge \\ \exists O_{xt}[On(x,t) : O_{xt} \wedge M o O_{xt}] \wedge \end{cases} \tag{5.3}$$

$$\textit{decomposition} : \begin{cases} \exists P_u[\textbf{Pickup}(x,f) : P_u \wedge P_u s M] \wedge \\ \exists H[Holding(x) : H \wedge P_u m H] \wedge \\ \exists P_d[\textbf{Putdown}(x,t) : P_d \wedge H m P_d \wedge P_d f M] \end{cases}$$

Note that in this axiom the quantification is over intervals. The predicates **Pickup** and **Putdown** are themselves actions, and also have axioms associated with them. The axiom for **Pickup** is

$$\begin{aligned} &\textbf{Pickup}(x,y) : P_u \Rightarrow \\ &\qquad \exists C_x[Clear(x) : C_x \wedge C_x m P_u] \wedge \\ &\qquad \exists O_{xy}[On(x,y) : O_{xy} \wedge P_u f O_{xy}] \wedge \\ &\qquad \exists C_y[Clear(y) : C_y \wedge P_u m O_{xy}] \wedge \\ &\qquad \exists H[Holding(x) : H \wedge P_u m H] \end{aligned}$$

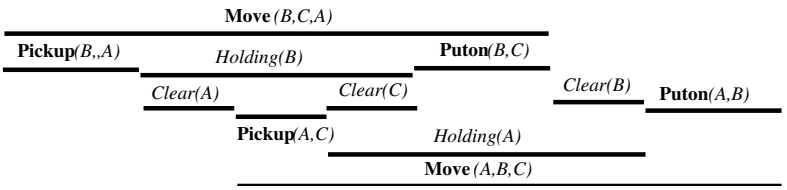

Figure 5.14: An interval plan

and that for **Putdown** is similar.

Planning then consists of adding intervals and interval relations in order to provide causal explanations for intervals that have no such explanation. For example, the proposition $On(B, C)$ over the interval F_2 has no explanation, so noticing that this fact is one of the effects of the action **Move**(B, C, x), (where x is an existential variable later to be instantiated) this action is introduced, together with the consequences that follow by the axiom (5.3). An explanation for the fact $On(A, B)$ is introduced in a similar way.

Exactly as in Example 5.1.1, it is necessary to impose a variety of integrity constraints. For example, we need to exclude from consideration models in which an interval over which $Clear(x)$ holds overlaps an interval over which $On(y, x)$ holds. These integrity constraints imply additional interval relations, beyond those entailed by the axioms associated with actions. Once these extra relations are incorporated, the plan shown in Figure 5.14 is obtained. Notice that this plan is in fact linearly ordered.

One of the advantages claimed by Allen and Koomen is that their planning method provides the ability to reason about concurrent behaviour. Notice that the plan constructed requires that the robot pick up A while still holding B. Thus, this is a plan for a robot with two arms, which does not require putting down block B.

We have a number of observations on this formulation of planning. First, let us remark that although the proposal is cast in terms of the interval calculus, it appears that it can reformulated using point relationships only. Most of the interval relationships asserted are primitive relationships, which we have already seen can be implemented in the point calculus. There are some disjunctive relationships, but these also can be translated into conjunctive point relationships. For example, the expression $C_t\{o, s, d\}M$ in the axiom for **Move** can be represented simply as $c_2 < m_2$, where $C_t = [c_1, c_2]$ and

$M = [m_1, m_2]$. Secondly, let us notice that once the translation into point relationships is done, we require the ability to reason with more than point relationships alone, since we need to allow for the integrity constraints. In other words, we need to factor into our complexity analysis the effect of predicates and quantification. It therefore appears that the results on querying indefinite order databases we have developed will bear directly on Allen and Koomens' formulation of planning. We will have more to say on this proposal in the next chapter, where we consider the combination of defined facts and linear order.

Chapter 6

Recursion and Linear Order

6.1 Introduction

In the previous chapter we considered the complexity of querying data in which the sole source of indefiniteness was incomplete knowledge about the linear order relations holding between points in the data. The present chapter[1] studies the effects of combining order indefiniteness with indefiniteness arising from defined relations. That is, we consider definitions making use of order relations.

As in the previous chapter, we will deal with a two-sorted language containing an object sort and an order sort, such that only order constants and variables may occur as arguments of the relation $<$, which will be interpreted over linear orders. Databases in this chapter will be pairs Σ, D where Σ is a Datalog program which may contain the order relation $<$ and D is a collection of ground atoms which may contain basic atoms and order atoms $u < v$, as in the previous chapter, but additionally may contain atoms in the defined predicates, whose interpretation is given by the program Σ. Queries will be positive existential formulae possibly including the order relation $<$. We defer the precise semantics to the next section.

We will not attempt to provide an exhaustive analysis of the complexity of querying indefinite information arising from the combination of defined relations and linear order. Instead, we focus on the case of recursive definitions using the linear order relation $<$, seeking to determine to what extent we retain decidability of the class of basic queries in this context. The combination of linear order and recursive defined relations is of interest because it can express certain aspects of concurrent, recursive actions. The

[1] Some of the results of this chapter have been presented in [87, 86].

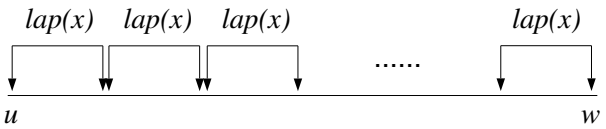

Figure 6.1: Swimming a finite number of laps

following is an example of an inference involving recursive actions.

Example 6.1.1: **Mermaids Get the Sack:** Ethel and Daphne Mermaid are synchronized swimmers in the employ of Sam Silverscreen, movie mogul. On the set of "Swimming in the Rain", Sam is giving Ethel and Daphne instructions for the next scene. "What I want you to do today is very simple," Sam says. "Just keep swimming laps of the pool. As soon as you finish a lap, start the next. *But*, whenever you both start a lap at the same time you must also finish it at the same time. Don't stop till I tell you to."
"OK, boss," reply Ethel and Daphne (even their speech is synchronized.)
"Lights! Camera! Action!" says Sam, and the Mermaids start their synchronized swimming. Sam retires to his office for a nap on the casting couch. When Sam returns some time later, he notices that Ethel has just completed a lap, whereas Daphne is still half way through hers. "Cut! Cut!" he yells when Daphne reaches the end of the pool. "Did either of you stop swimming at any stage?" he asks.
"No, boss," reply the Mermaids.
"Well, you're both fired anyway," says Sam.
Question: Why did the Mermaids get the sack?

To answer this question, we give a logical formulation. Sam gave the Mermaids two instructions: not to stop swimming, and to finish simultaneously any lap started simultaneously. We know the first instruction was followed by both Mermaids, so let us define the predicate $\mathbf{laps}(x, u, w)$ to mean that x swam, without stopping, some finite number of laps between time u and time w. This may be done using the rules

$$\mathbf{laps}(x, u, w) : -lap(x, u, w), u<w$$
$$\mathbf{laps}(x, u, w) : -lap(x, u, v), \mathbf{laps}(x, v, w), u<v<w$$

Figure 6.1 illustrates the conditions under which $\mathbf{laps}(x, u, w)$ holds.

Sam's knowledge about the actions performed by the Mermaids may then be represented by the database

$$\{\mathbf{laps}(E, a, c), \mathbf{laps}(D, a, b), lap(D, b, d), a<b<c<d\}$$

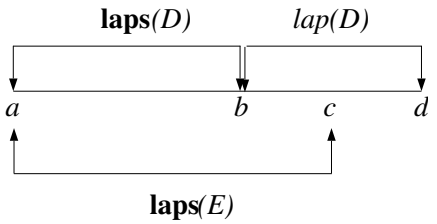

Figure 6.2: Sam's knowledge

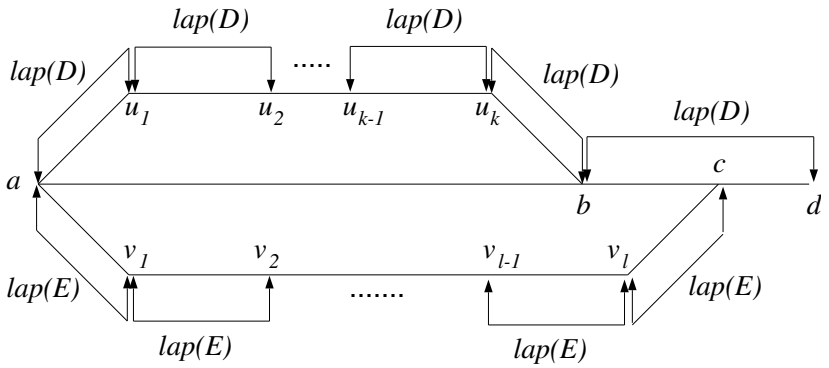

Figure 6.3: The data expanded

and is illustrated in Figure 6.2. Here c is the time at which Sam returns from his nap.

The second condition imposed by Sam, that laps started simultaneously must be finished simultaneously, is violated just in case the query

$$\Phi = \exists xyuvw[lap(x,u,v) \wedge lap(y,u,w) \wedge v{<}w]$$

holds. Here we have a lap of some person x starting at the same time u as a lap of y, but y finishes after x. As we now show, this query is entailed by the data. In other words, the Mermaids were not very good as synchronized swimmers, and Sam was justified in firing them.

To see why the query is entailed by the data, let us expand the defined atoms in this database. We obtain a set of facts as depicted in Figure 6.3. For each of the Mermaids we know that they swam a some finite number of laps over a certain period of time. However, even after we have chosen for each Mermaid some number of laps, there is still some indeterminacy, since we must also decide how these laps were "synchronized." That is, we must decide what are the order relations between the constants u_i and v_j. Informally, we may reason as follows.

If either $u_1 < v_1$ or $v_1 < u_1$, then the query is satisfied, since Ethel and Daphne started their first lap at the same time. Suppose therefore that $u_1 = v_1$ By a similar argument, we see that either the query is satisfied or $u_2 = v_2$. Continuing in this manner, we obtain that either the query is entailed or, for some $j \leq l$, we have $b = v_j$. But then we are in a situation in which Ethel swims a lap starting at time b, and finishes it at time c or earlier, in any case before Daphne completes her lap started at time b. This once again means that the query is satisfied. Thus, however many laps each Mermaid swam, there must have been a point at which they fell out of synchrony. □

We develop in this chapter the following results. We show that, in general, basic queries containing the relation $<$ are undecidable in recursively indefinite databases. However, provided we restrict the class of rules used in the definitions, we are able to recover the decidability of basic queries containing $<$. The decidable class of databases we identify requires that all definitions be *regular*. Regular rules are linear rules, like those in Example 6.1.1, in which the defined predicates contain two order arguments, the order variables in the body of the rule are linearly ordered, and the recursive call is on the last two of these variables.

The data complexity of basic queries with respect to regular databases turns out to be PSPACE-complete. However, we identify a restricted class of problems for which this complexity may be reduced to polynomial time. This class involves a modification of the bounded width constraint we found to reduce combined complexity in the monadic case. As we have mentioned in the previous chapter, the width bound does not help to reduce complexity for indefinite order databases using binary predicates. We show in this chapter that a closely related concept, a bound on the amount of "concurrency" in models of the database, does suffice to reduce data complexity to PTIME. The ideas we use to show this involve combining the notion of glue type from Chapter 3 with the automata theoretic ideas used in connection with the width bound in Chapter 5.

Section 6.2 formally describes the class of databases considered in this chapter. We explain in this section why the decidability result of basic queries breaks down when order is introduced, and introduce the class of regular rules, which will be central to the

decidable cases discussed in this chapter. Section 6.3 introduces a modified notion of glue type which can be exploited to show that databases containing only regular rules are decidable for basic queries. The details of the decision procedure, and a study of the complexity of the query problem for this class of rules are in Section 6.4. Section 6.5 adapts some of the automata theoretic ideas of the previous chapter to show that improvements in complexity are possible for regular databases of bounded concurrency. The results of this section show that Example 6.1.1 lies in a class of problems that may be solved with polynomial time data complexity.

6.2 Semantic Preliminaries and Basic Results

In this section formally describe the semantic framework we will assume when dealing with the combination of recursion and linear order. To show that the results of this chapter hold under various assumptions about the linear order relation, we briefly reprise the discussion of Section 5.2 in order to check that the reductions between the consequence relations $\models_{\mathbf{Q}}, \models_{\mathbf{Z}}$ and $\models_{\mathbf{Fin}}$ continue to apply when we are dealing with databases containing defined concepts. Finally, we introduce the class of regular rules. The main contribution of this chapter will be to show the decidability of basic queries in databases containing atoms whose definition uses regular rules.

The consequence relations we will be concerned with in this chapter for databases containing defined atoms whose definition involves linear order will be that obtained by treating the relation $<$ as a basic relation on the order domain, but banishing models in which $<$ does not denote a linear order. Formally, given a set of linear order types $\mathcal{C}$, let $Mod_{\mathcal{C}}(\Sigma, D)$ be the set of two-sorted models M for which there exists an interpretation N of the basic predicates and $<$, in which $<$ denotes a linear order in the class $\mathcal{C}$, such that $M = \Sigma^{\infty}(N)$ and $M \models D$. We write $\models_{\mathcal{C}}$ for the consequence relation associated with this semantics. That is $\Sigma, D \models_{\mathcal{C}} \Phi$ when $M \models \Phi$ for all $M \in Mod_{\mathcal{C}}(\Sigma, D)$.

We will consider once again the classes $\mathbf{Fin}, \mathbf{Z}$ and $\mathbf{Q}$, and develop our decision procedures for the case $\mathbf{Fin}$. As before, there exists an equivalent consequence relation defined using a smaller class of models, which corresponds more closely to the decision procedures we will develop. By treating the predicate $<$ as a basic predicate, we may

apply without modification the notions of *expansion* of a database (see Section 3.3) to databases containing defined facts whose definitions involve order relations. Note however, that an expansion of such a database corresponds not to a model of the database, but rather to a (flat) order indefinite database, in the sense of the previous chapter. Expansions may now have several minimal models, corresponding to different topological sorts of the order constants occurring in them.

Lemma 6.2.1: Let $\mathcal{C}$ a class of linear order types and Φ a basic $\{<\}$-query. Then $\Sigma, D \models_{\mathcal{C}} \Phi$ if and only if $E \models_{\mathcal{C}} \Phi$ for all expansions E of D by Σ.

Define a model M to be a *minimal* model of the database Σ, D if there exists an expansion E of D by Σ such that M is a minimal model of E in the sense of the previous chapter. That is, the model M is obtained by topologically sorting the order constants in the expansion E. We will write simply $\models$ for the consequence relation corresponding to truth in all minimal models, as before. The proof of the following generalization of Corollary 5.2.9 is then straightforward:

Lemma 6.2.2: For any program involving $<$ we have $\Sigma, D \models_{\mathbf{Fin}} \Phi$ if and only if $\Sigma, D \models \Phi$.

This reduction enables us to confine our attention in the decision procedure to those models obtained by a combination of expansion and topological sorting. All of the other reductions of the previous chapter between the consequence relations $\models_{\mathbf{Q}}, \models_{\mathbf{Z}}$ and $\models_{\mathbf{Fin}}$ continue to hold when dealing with defined predicates. We confine ourselves here to the equivalence result for tight queries. Modified versions of the other reductions of the previous chapter (Proposition 5.2.3 and Corollary 5.2.6) follow by similarly trivial arguments.

Lemma 6.2.3: If Φ is a tight basic query then $\Sigma, D \models_{\mathbf{Fin}} \Phi$ iff $\Sigma, D \models_{\mathbf{Z}} \Phi$ iff $\Sigma, D \models_{\mathbf{Q}} \Phi$.

Proof: By Lemma 6.2.1 we have $\Sigma, D \models_{\mathbf{Fin}} \Phi$ if and only if $E \models_{\mathbf{Fin}} \Phi$ for all expansions E of D by Σ. Since Φ is a tight query this holds by Lemma 5.2.2 just when $E \models_{\mathbf{Z}} \Phi$ for all expansions E of D by Σ. Applying Lemma 6.2.1 once more yields the first equivalence. The argument for the remaining equivalence is identical. □

Let us now consider whether the decidability of basic queries continues to hold if we admit the relation $<$. It turns out that we have already done the work needed to answer this question.

Example 6.2.4: We have seen in Example 4.5.3 that use of inequality in queries may be used to 'synchronize' two sequences. This is also possible using queries containing order relations. Specifically, suppose we have two sequences $A(t_0, t_1)$, $A(t_1, t_2), \ldots, A(t_{m-1}, t_m)$ and $A(t_0, s_1), A(s_1, s_2), \ldots, A(s_{n-1}, s_n)$ starting at the same point, with $m \leq n$. Then the query $\Phi = \exists xyz[A(x, y) \wedge A(x, z) \wedge y{<}z]$ holds unless $t_1 = s_1, t_2 = s_2, \ldots, t_m = s_m$. For example, if $t_1 \neq s_1$ then either $t_1{<}s_1$ or $s_1{<}t_1$. In either case the query holds. If $t_1 = s_1$ then we may continue this argument with t_2, s_2 etc., obtaining that either the query holds or the chain of equalities above holds. □

We may use the construction of Example 6.2.4 to show that queries containing inequalities do not have a finite number of glue types, exactly as in Example 4.5.3. In addition, we have the following Corollary:

Corollary 6.2.5: There exists a fixed linear monadic program Σ (not containing $<$) and a conjunctive query Φ which contains $<$ such that it is undecidable to determine for databases D if $\Sigma, D \models \Phi$.

The proof of this follows exactly along the lines of Theorem 4.5.4, but using the construction of Example 6.2.4 instead of that of Example 4.5.3. As before, the proof does not require that the program Σ contain any occurrences of $<$: it suffices that this relation occur in the query. In the case of queries containing inequality we concluded from Theorem 4.5.4 that there do not exist interesting combinations of recursion and inequality that lead to decidable query problems. In spite of the similarities between Theorem 4.5.4 and Corollary 6.2.5, we will not draw the same pessimistic conclusion in the case of linear order. The results of this chapter will show that for a certain constrained class of recursive definitions containing linear order we retain the decidability of basic queries. The decidable class of rules is obtained in an orthogonal fashion, not by *restricting* the order-free decidable problem (Corollary 6.2.5 shows this move to be

unproductive), but by requiring that all rules contain the linear order relation in a certain way. The following slightly modified version of Example 6.1.1 will be used as a running example.

Example 6.2.6: Let Σ be the recursive program with rules

$$\mathbf{R}(u,w) : -A(u,v), u<v<w, \mathbf{R}(v,w)$$
$$\mathbf{R}(u,w) : -A(u,w), u<w$$

defining the predicate $\mathbf{R}$. The predicate A is basic. Consider the database $D = \{\mathbf{R}(a,b), \mathbf{R}(a,c), A(b,d), a<b<c<d\}$. The query

$$\Phi = \exists xyz[A(x,y) \wedge A(x,z) \wedge x<y<z]$$

is entailed by the indefinite database Σ, D. This is just Example 6.1.1 with the mermaids deleted. □

We restrict ourselves for the remainder of the chapter to defined predicates of the form $\mathbf{R}(\mathbf{x}, u, v)$, where the arguments $\mathbf{x}$ are object arguments and there are precisely two order arguments u, v. Basic predicates may still have an arbitrary form. We will say that a program containing $<$ is *regular* if every rule is a linear rule of the form

$$\mathbf{R}(\mathbf{x}, t_1, t_k) : -B(\mathbf{x}, \mathbf{y}, t_1, \ldots t_{k-1}) \wedge t_1 < \ldots < t_{k-1} < t_k \wedge \mathbf{S}(\mathbf{x}, \mathbf{y}, t_{k-1}, t_k)$$

in which $\mathbf{x}$ and $\mathbf{y}$ are vectors of object variables, the t_i are order variables, B is a conjunction of basic atoms and $\mathbf{S}$ is an optional defined atom. We require that there be no variable that occurs more than once in the head. (This constraint is inessential, but helps to simplify the constructions of this chapter. We refer the reader interested in seeing how it can be removed back to Chapter 3, where we allowed this possibility. We also comment that this simplification obviates the need to consider the unique names semantics.) The program of Example 6.1.1 is regular. Databases Σ, D in which the program Σ is regular will be called *regular databases*. We will show that basic queries remain decidable in regular databases.

6.3 Glue Types for Order Databases

This section is devoted to adapting the notion of glue types for the linear order case. We have already noted that the number of glue types of expansions of atoms by rules involving $<$ is not in general finite, so a different approach needs to be taken to decision procedures. Anticipating somewhat, the decision procedure of the next section will attempt to construct a model of a regular database Σ, D by a left-to-right traversal: at each stage of this process we will have a partially completed minimal model on the left. Because of recursion, this procedure may construct an infinite set of minimal models. Decidability will be a consequence of the fact that we can summarize the left components in such a way as to have only a finite number of types.

Definition 6.3.1: A *left k,l-database* consists of a tuple $G = \langle E, \sigma, \tau \rangle$ in which

1. E is a set of basic atoms and order atoms,
2. σ is a sequence of k object constants of E, called the *object sources*
3. τ is a sequence of l order constants of E, called the *order sources,*
4. E contains the atoms $\tau[1]<\tau[2]<\ldots<\tau[l]$
5. For each order constant u distinct from $\tau[l]$ the order relations in E entail $u < \tau[l]$.

We say that a left k,l-database is a *left k,l-graph* if the order constants are linearly ordered. Intuitively, this means that we may interpret G as a model rather than as a set of facts. Similarly, a *right k,l-graph* is a tuple $\langle E, \sigma, \tau \rangle$ satisfying the above conditions, except that now we replace constraint 5 by the constraint that $\tau[1]$ be a *minimal* element of the linear order on the order constants.

Constants not equal to any source will be called *internal* constants. Say that a right k,l-graph H is *proper* if every internal order constant is greater than $\tau[l]$, and no two sources are equal. For example, a graph with order constants $u < v < w$ and sources $\tau = uw$ is not proper because the constant $v < \tau[2]$ is internal. (The constraint on the equality of sources is related to the restriction that variables not occur twice in the heads of rules, and would be removed in the general case.)

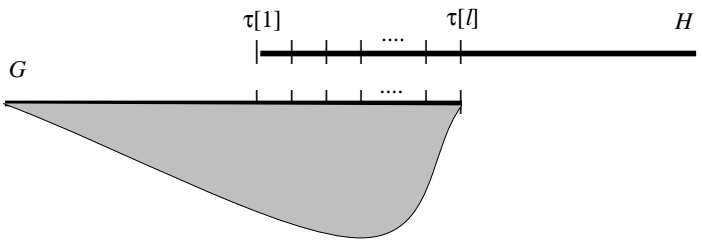

Figure 6.4: Gluing a left database to a right graph

Given a left k,l-database G and a *proper* right k,l-graph H we define the result $G \circ H$ of *gluing* G to H to be the model obtained from the union of the sets of basic and order atoms of G and H after performing the following identifications:

1. For $i = 1 \ldots k$ identify $\sigma_G[i]$ with $\sigma_H[i]$.

2. For $i = 1 \ldots l$ identify $\tau_G[i]$ with $\tau_H[i]$.

We require that the constants of G and H be standardized apart before gluing, so that the internal constants do not interact. Figure 6.4 illustrates this operation. Note that if G is a left graph then after the identification of sources the order atoms of G and H induce a linear order on all the order constants. (This is not necessarily true if H is not proper.)

Example 6.3.2: We will write k,l-graphs and k,l-databases using the notation $\{\sigma : \tau \mid E\}$ in which the first component lists the object sources, the second component lists the order sources and the last component lists the atoms. Consider the left 1,2-graph

$$G = \{a : t_2 t_4 \mid A(a,t_1,t_2), B(t_1,t_3), A(b,t_2,t_4), t_1{<}t_2{<}t_3{<}t_4\}$$

and the proper right 1,2-graph

$$H = \{c : u_1 u_2 \mid A(c,u_1,u_3), B(u_2,u_3), u_1{<}u_2{<}u_3\}$$

Then the result of gluing H to G is the model with atoms

$$\{A(a,t_1,t_2), B(t_1,t_3), A(b,t_2,t_4), A(a,t_2,u_3), B(t_4,u_3), t_1{<}t_2{<}t_3{<}t_4{<}u_3\}$$

after performing the identifications $a = c$, $u_1 = t_2$ and $u_2 = t_4$. □

Glue types again underlie the decision procedures of this chapter. In the case of databases not containing order relations, models were essentially sets of facts, and we were able to use graphs as representatives of the glue types of graphs. This appears not to be possible in the order case. Instead, glue types will now be represented by k,l-databases, which do not correspond directly to models because they need not be linearly ordered. This necessitates introducing a new consequence relation $\models_{po}$. Formally, given a set D of basic and order atoms, we have $D \models_{po} \Phi$ when Φ holds in the structure obtained from D by interpreting $<$ as the *partial order* derived from the order atoms of D. Equivalently, we take the consequence relation obtained by considering all models of D in which the relation $<$ is interpreted as a partial order.

For each query Φ we define an equivalence relation on left k,l-databases by $G_1 \sim_\Phi G_2$ when for all proper right k,l-graphs H, we have $G_1 \circ H \models_{po} \Phi$ if and only if $G_2 \circ H \models_{po} \Phi$. Because the result of gluing a left graph to a proper right graph is linearly ordered, the consequence relation in this definition is just evaluation of Φ in a model in the case that G is a left graph. As before, we call the equivalence classes with respect to this order the *glue types* of Φ.

We now derive a succinct representation for glue types of left databases. Suppose that Φ is a conjunctive query, written in the form $\Phi = \exists \mathbf{xt}[B(\mathbf{xt}) \wedge O(\mathbf{t})]$ where B contains all the basic atoms and O consists of the order atoms. As in Chapter 5, we view O as a dag, and write $<$ for the associated order on the variables $\mathbf{t}$. A *partition* will be a decomposition of Φ of the form

$$B_L(\mathbf{x}_L\mathbf{x}_M\mathbf{t}_L\mathbf{t}_M) \wedge B_R(\mathbf{x}_M\mathbf{x}_R\mathbf{t}_M\mathbf{t}_R) \wedge O_L(\mathbf{t}_L\mathbf{t}_M) \wedge O_R(\mathbf{t_L}\mathbf{t}_M\mathbf{t}_R) \tag{6.1}$$

satisfying the following constraints:

1. The object variables $\mathbf{x}$ are partitioned into three sets $\mathbf{x}_L, \mathbf{x}_M, \mathbf{x}_R$.

2. The order variables $\mathbf{t}$ are partitioned into three sets $\mathbf{t}_L, \mathbf{t}_M, \mathbf{t}_R$.

3. If $v \in \mathbf{t}_L \cup \mathbf{t}_M$ and $u < v$ then $u \in \mathbf{t}_L \cup \mathbf{t}_M$.

4. If $v \in \mathbf{t}_R$ and $v < u$ then $u \in \mathbf{t}_R$.

5. If an order atom has all its arguments in $\mathbf{t}_L \cup \mathbf{t}_M$ then it is contained in the component O_L.

With respect to the object variables this new notion of partition behaves exactly as the notion of partition of Section 3.4. Note that the form of Equation 6.1 implies that Φ contains no atoms involving both a variable in $\mathbf{x}_L$ and a variable in $\mathbf{x}_R$. However, we may have atoms containing variables from both sides of the pairs $\langle \mathbf{x}_L, \mathbf{x}_M \rangle$ and $\langle \mathbf{x}_M, \mathbf{x}_R \rangle$. With respect to the order variables and basic atoms the structure of partitions is similar. Order variables in order atoms behave slightly differently. The query Φ may contain order atoms $u < v$ with $u \in \mathbf{t}_L$ and $v \in \mathbf{t}_R$: these must appear in O_R. However, by condition 5, no order atoms involving only variables in $\mathbf{t}_L \cup \mathbf{t}_M$ are in O_R. That is, O_L contains all the constraints involving variables in $\mathbf{t}_L \cup \mathbf{t}_M$.

Example 6.3.3: Figure 6.5 illustrates the structure of a partition of the query

$$A(r,s) \wedge A(v,w) \wedge C(x,y) \wedge C(t,u) \wedge r{<}s{<}v{<}w{<}y \wedge r{<}t{<}u{<}x{<}y$$

in which all variables are order variables. Here we have $\mathbf{t}_L = ru$, $\mathbf{t}_M = stv$ and $\mathbf{t}_R = wxy$. The heavy dotted line separates $\mathbf{t}_L$ from $\mathbf{t}_R$. Observe that no atom contains variables from both $\mathbf{t}_L$ and $\mathbf{t}_R$, although the atom $A(v,w)$ "crosses" the line. The components of the partition are $B_L = \{A(r,s), C(t,u)\}$, $O_L = \{r{<}s{<}v, r{<}t{<}u\}$, $B_R = \{A(v,w), C(x,y)\}$ and $O_R = \{v{<}w{<}y, u{<}x{<}y\}$. Note that the atom which crosses the line is contained in B_R. □

Intuitively, the variables $\mathbf{x}_M, \mathbf{t}_M$ will correspond to the interface between the left and the right portions of models given as $G \circ H$, that is, to sources of glue types, whereas the remaining variables will correspond to internal constants. Given a partition ψ of Φ, a *ψ-contribution* will be any left k,l-database D obtained from the atoms

$$B_L(\mathbf{x}_L \mathbf{x}_M \mathbf{t}_L \mathbf{t}_M) \wedge O_L(\mathbf{t}_L \mathbf{t}_M)$$

by the following operations. First, identify some of the variables, in such a way as to preserve the partition of variables (that is, identify variables in $\mathbf{x}_L$ only with other variables in $\mathbf{x}_L$, etc.). Next, make each of the variables $\mathbf{x}_M$ one of the object sources σ. Also, make each of the variables $\mathbf{t}_M$ one of the order sources τ, in such a way that

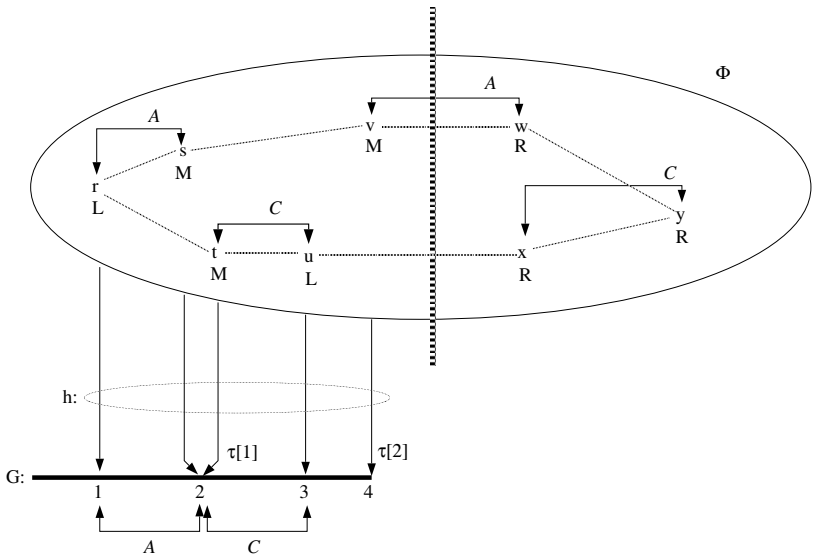

Figure 6.5: A partition of a query

O_L is consistent with the order $\tau[1] < \tau[2] < \ldots < \tau[l]$ on the order sources. Finally, for any variable v for which $v \leq \tau[l]$ does not now follow, add the atom $v < \tau[l]$. (The effect of this last step is to ensure that $\tau[l]$ is the unique maximal element, so that the result satisfies constraint 5 of definition 6.3.1)

Now say that D is *supported* by a left k,l-database G if there exists a mapping h of the variables of D to the constants of G such that

1. For each i, we have $h(\tau_D[i]) = \tau_G[i]$ and $h(\sigma_D[i]) = \sigma_G[i]$.
2. For every object and order atom $A(\mathbf{x})$ of D we have $G \models_{po} A(h(\mathbf{x}))$

Intuitively this holds just when D, interpreted as a query, is satisfied in G.

Example 6.3.4: We may construct a ψ-contribution for the partition of Example 6.3.3 as follows. First, we identify the variables s and t by substituting s for t. Notice these variables are both in $\mathbf{x}_M$. Next we assign sources by setting $\sigma = \emptyset$ and $\tau = sv$. Observe that now $\tau[2] = v$, but $u \leq v$ does not follow from the result. Hence we add the atom $u < v$. We obtain the 0,2-database

$$D = \{ : sv \mid A(r,s), C(t,u), r{<}s{<}v, r{<}s{<}u{<}v\}$$

Suppose that G is the graph $\{ : 2,4 \mid A(1,2), C(2,4), 1 < 2 < 3 < 4\}$. Figure 6.5 shows a mapping h witnessing that D is supported by G. □

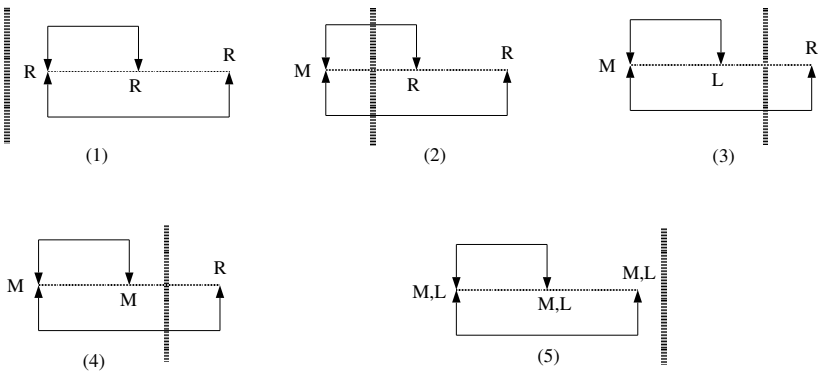

Figure 6.6: The partitions of a query

We let $Rep(G, \Phi)$ be the left graph obtained by gluing together all the ψ-contributions which are supported by G, as ψ varies over all partitions of Φ. That is, we identify the corresponding sources, but the remaining variables of distinct ψ-contributions D are renamed apart.

Example 6.3.5: We calculate the glue type of the 0,1-graph $G = \{: b \mid A(a,b), a<b\}$ with respect to the query $\Phi = \exists xyz[A(x,y) \wedge A(x,z) \wedge x<y<z]$. Figure 6.6 shows the partitions of the query. None of these, it turns out, yield a non-trivial contribution. In the case of (1) and (2), this is because there are no atoms in the component B_L. Note that in (2), (3) and (4) the node x must be labelled M, since it is contained in an atom that crosses the line. In case of (3) and (4), we have $B_L = \{A(x,y)\}$, but this cannot be supported by G because a is internal, whereas x must map to a source, because it is labelled M. Finally, (5) represents a total of 8 partitions, none of which is supported. Thus we obtain the trivial representative $Rep(G, \Phi) = \{: b \mid\}$. □

The following lemma states that $Rep(G, \Phi)$ records sufficient information about G to determine whether $G \circ H$ entails a query Φ, and generalizes Proposition 3.4.6 to order databases.

Lemma 6.3.6: For all left k,l-databases G and proper right k,l-graphs H we have $G \circ H \models_{po} \Phi$ if and only if $Rep(G, \Phi) \circ H \models_{po} \Phi$. The left database $Rep(G, \Phi)$ may be represented in space $(kl)^{2|\Phi|}$.

Proof: The functions h mapping the ψ-contributions to G may be combined into a homomorphism from $Rep(G, \Phi)$ to G, so it is straightforward to show that $Rep(G, \Phi) \circ H \models_{po} \Phi$ implies $G \circ H \models \Phi$. For the converse, suppose that θ is a satisfying mapping from the variables of Φ to the constants of $G \circ H$. We construct a partition of Φ, by means of the following definitions. First, take $\mathbf{x}_M$ ($\mathbf{t}_M$) to be the set of object variables x (respectively, order variables t) for which $\theta(x)$ (respectively, $\theta(t)$) is a source. Take $\mathbf{x}_L$ and $\mathbf{t}_L$ to be the set of variables of the appropriate sort which map to an internal constant of G. Similarly, take $\mathbf{x}_R$ and $\mathbf{t}_R$ to be the variables which map to an internal constant of H. Let B_L be the set of basic atoms A of Φ for which $A\theta$ is supported in $G \circ H$ by an atom arising from G. Let B_R be the remaining basic atoms. Finally, take O_L to be the order atoms of Φ not containing the variables $\mathbf{t}_R$, and O_R to be the rest.

We now show that these choices in fact yield a partition. First, we show that the form of the conjunction 6.1 is satisfied. Notice that variables from $\mathbf{x}_R$ and $\mathbf{t}_R$ cannot appear in an atom A of B_L because in this case $A\theta$ contains an internal constant of H, so cannot hold in G. Similarly, any basic atom A that contains a variable from $\mathbf{x}_L$ or $\mathbf{t}_L$ must be in B_L because $A\theta$ contains an internal constant of G, so must be supported by an atom from G. Thus B_R does not contain any variables from $\mathbf{x}_L$ or $\mathbf{t}_L$. The order atoms O_L do not contain the variables $\mathbf{t}_R$ by construction. This shows that the form of 6.1 is satisfied. If $v \in \mathbf{t}_L \cup \mathbf{t}_M$ and $u < v$ follows from the order atoms of Φ then $\theta(u) < \theta(v)$ holds in $G \circ H$. This implies that $\theta(u)$ is less than the maximal element of G, so cannot be an internal constant of H, which is proper. Thus $u \in \mathbf{t}_L \cup \mathbf{t}_R$ also, so constraint 3 of the definition of partition is satisfied. Constraint 4 of the definition of a partition follows because if H is proper then any constant greater than an internal constant is also an internal constant. Any order atoms not containing variables $\mathbf{t}_R$ are in O_L so constraint 5 is satisfied by construction. This shows that we do in fact have a partition ψ.

We now construct an assignment θ' such that $Rep(G, \Phi) \circ H \models \Phi\theta'$. Notice first that $G \models_{po} (B_L \wedge O_L)\theta$. This is because by construction of the partition θ maps all the variables occurring in $B_L \wedge O_L$ to constants of G, and because the fact that H is a proper right graph implies that an order atom or basic atom holds over these constants

in $G \circ H$ just in case it holds in G. It follows that the ψ-contribution obtained from B_L, O_L and θ is part of the representative $Rep(G, \Phi)$. On the variables $\mathbf{x}_L, \mathbf{t}_L, \mathbf{x}_M, \mathbf{t}_M$ we define θ' to map a variable to its image in this ψ-contribution in $Rep(G, \Phi)$. It is then immediate that $Rep(G, \Phi) \models_{po} (B_L \wedge O_L)\theta'$ For the remaining variables $\mathbf{x}_R, \mathbf{t}_R$ we define θ' to be equal to θ. Consider the atoms B_R, O_R. Each basic atom A in B_R contains a variable in $\mathbf{x}_R$ or $\mathbf{t}_R$, and consequently $A\theta$ must hold in H. Atoms $u < v$ in O_R can be of two forms: either we have $u \in \mathbf{t}_L$ and $v \in \mathbf{t}_R$ or both u and v are in $\mathbf{t}_M \cup \mathbf{t}_R$. In the former case we have $Rep(G, \Phi) \circ H \models \theta'(u) < \theta'(v)$, because we have $\theta'(u) < \tau[l]$ in the ψ-contribution obtained from θ, and $\tau[l] < \theta'(v)$ because $\theta(v)$ is an internal constant of the proper right graph H. In the latter case we have $H \models \theta(u) < \theta(v)$. It follows from these considerations that $Rep(G, \Phi) \circ H \models \Phi\theta'$, as desired.

It remains only to establish the bound on the size of $Rep(G, \Phi)$. Notice that each ψ-contribution can be written in space $2|\Phi|.(log\ k + log\ l)$ where the factor $log\ k + log\ l$ arises from the need to represent the sources and the factor of 2 arises from the possible need for order atoms not in Φ. Since $Rep(G, \Phi)$ is the union of a set of such contributions we obtain a total size of $(kl)^{2|\Phi|}$. $\square$

The result shows that the glue type of a left database encodes enough information to compute the glue type of the result of extending by a proper right graph. This lemma permits us to work with glue types rather than the actual databases themselves, and is the crucial link in the decidability result for regular rules.

Lemma 6.3.7: Let G be a left k, l-graph and H a proper right k, l-graph. Suppose σ and τ are sequences of object and order constants of H, respectively, and let I be the left graph obtained from $G \circ H$ by using σ and τ as sources. Then, given $Rep(G, \Phi)$ and H, it is possible to construct $Rep(I, \Phi)$ in deterministic space $O\left(|Rep(G, \Phi)| + |H| + (kl)^{2|\Phi|}\right)$.

Proof: Let σ_G, τ_G be the sources of G and σ_I, τ_I the sources of $Rep(I, \Phi)$. We will show that

$$Rep(I, \Phi) \sim_\Phi Rep(\{\sigma_I : \tau_I \mid Rep(G, \Phi) \circ H\}, \Phi)$$

The result follows from this since the right hand side may be computed exactly as described above. Suppose we are given a proper right k,l-graph J. Then

$$\{\sigma_I : \tau_I \mid G \circ H\} \circ J = G \circ \{\sigma_G : \tau_G \mid \{\sigma_I : \tau_I \mid H\} \circ J\}$$

Thus the left hand side satisfies Φ if and only if the right hand side satisfies Φ. Since $\{\sigma_G : \tau_G \mid \{\sigma_I : \tau_I \mid H\} \circ J\}$ is a proper right graph, it follows from Lemma 6.3.6 that this holds just in case

$$Rep(G, \Phi) \circ \{\sigma_G : \tau_G \mid \{\sigma_I : \tau_I \mid H\} \circ J\} \models_{po} \Phi$$

Now the left hand side of this entailment relation is equal to $\{\sigma_I : \tau_I \mid Rep(G, \Phi) \circ H\} \circ J$ so again by Lemma 6.3.6 this is equivalent to

$$Rep(\{\sigma_I : \tau_I \mid Rep(G, \Phi) \circ H\}, \Phi) \circ J \models_{po} \Phi$$

which proves the required equivalence. □

6.4 Decidability of Regular Rules

We are now ready to show the decidability of basic queries in regular databases. As usual, the decision procedure works by attempting to construct a counter-model. This is done differently from the previous procedure, which operated by guessing a glue type for each defined atom and verifying that this atom can in fact be generated. Notice that expansions of regular rules contain order relations. If two regular facts are concurrent, for example, if we have defined facts $\mathbf{A}(u, v), \mathbf{B}(u, v)$, then their expansion will result in a database which is not linearly ordered, because it contains two disjoint linearly ordered sequences between u and v. Thus a further topological sorting step is necessary after choosing an expansion for each defined atom. As we have seen in Example 6.2.4 it is not possible for a finite number of glue types to represent the way an expansion interleaves with another. This means that the previous approach no longer works.

What we do instead is to construct the model by a left to right traversal of the database, alternating expansion of defined atoms with topological sort of the basic facts introduced by expansion. We begin by (non-deterministically) topologically sorting

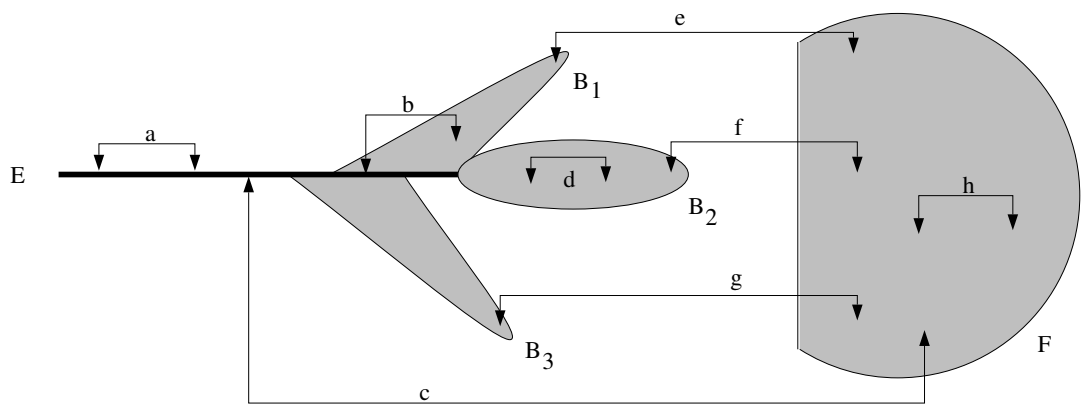

Figure 6.7: A stage of the expansion process

order constants from the left, by repeatedly choosing some set of minimal elements of the unsorted partial order for the next point of the linear order being constructed, and deleting these from the partial order. This continues until we reach a state in which there exists a defined atom $\mathbf{P}(\mathbf{x}, u, v)$ for which u is the greatest element in the linear order being constructed. That is, we stop sorting once we have selected the left point of a defined atom in the database. At this point it would be an error to continue the topological sort, because some constant internal to the expansion of the defined atom could be next in the linear order. Thus, we expand all the defined atoms with left order constant u by replacing each with the body of an applicable rule. This 'exposes' the minimal internal constants. We may now resume the topological sort, until we again encounter a defined atom, at which stage we must once more expand before proceeding. This alternation between sorting and expansion continues until the partial order is empty. Of course, if we have recursive rules then this procedure may construct arbitrarily large linear expansions, because expansion of defined atoms may introduce new defined atoms. We will use left glue types below to show that it is possible to give compact representations of the intermediate states generated by this process, so that it suffices to consider a finite set of expansions.

Example 6.4.1: Consider the database and query of Example 6.2.6. We show one possible sequence of steps in the linearization process. We use $\{L \mid R\}$ to denote the stages of the process: L will be the portion of the database already

sorted, R will be the remainder. The initial configuration is

$$\{ \,!\, \mathbf{R}(a,b), \mathbf{R}(a,c), A(b,d), a{<}b{<}c{<}d\}$$

The first step is to start the topological sort, so we move the unique minimal element a to the left, obtaining

$$\{a \,!\, \mathbf{R}(a,b), \mathbf{R}(a,c), A(b,d), b{<}c{<}d\}$$

We now have defined atoms containing the maximal left element a, so we must switch to the expansion phase. Choosing the recursive rule for both defined atoms yields the configuration

$$\{a \,!\, A(a,b'), \mathbf{R}(b',b), A(a,c'), \mathbf{R}(c',c), A(b,d), b'{<}b{<}c{<}d, c'{<}c\}$$

(Note that the right hand side is now no longer linearly ordered.) We now return to the sorting phase. Suppose we choose to map the two minimal elements b', c' to the next point. We identify b' and c' by substituting b' for every occurrence of c' and get the configuration

$$\{A(a,b'), a{<}b' \,!\, \mathbf{R}(b',b), \mathbf{R}(b',c), A(b,d), b{<}c{<}d\}$$

We now once more have defined atoms involving the maximal left element b', so we switch to expansion again. If we choose the nonrecursive rule for both defined atoms we obtain

$$\{A(a,b'), a{<}b' \,!\, A(b',b''), A(b',c''), A(b,d), b''{<}b{<}c{<}d, c''{<}c\}$$

From this configuration it is possible to continue the topological sort to obtain the sequence of configurations

$$\begin{gathered}
\{A(a,b'), A(b',b''), a{<}b'{<}b'' \,!\, A(b',c''), A(b,d), b{<}c{<}d, c''{<}c\}\rightarrow \\
\{A(a,b'), A(b',b''), A(b',c''), a{<}b'{<}b''{<}c'' \,!\, A(b,d), b{<}c{<}d\}\rightarrow \\
\vdots \\
\{A(a,b'), A(b',b''), A(b',c''), A(b,d), a{<}b'{<}b''{<}c''{<}b{<}c{<}d \,!\, \}
\end{gathered}$$

in which two intermediate configurations have been omitted. Note that the final configuration satisfies the query $\exists xyz[A(x,y) \wedge A(x,z) \wedge x{<}y{<}z]$ with $x = b', y = b''$ and $z = c''$. $\square$

The possible structure of the intermediate configurations is illustrated in Figure 6.7. Arrow-headed arcs in this diagram represent atoms of various sorts, where the arrows point to the left and right order constants of the atom. Suppose U is the set of order constants that have been linearly ordered at some stage. Let E be the set of atoms of the partial expansion all of whose order constants are in the set U, depicted in the diagram by arcs of type a. All of these atoms must be basic, since we expand any defined atom we encounter. Of the remaining atoms, some arise from expanding a defined atom, while others were in the original database D.

Consider first the former, depicted as the rule bodies B_i. For example, B_2 is a rule body arising from an atom that has just been expanded, whereas B_1 and B_3 are from earlier expansions, and already have some of their constants in the sorted set U. The basic atoms in the rule bodies are depicted as arcs b, which has its left constant already sorted, and d both of whose order constants remain to be sorted. The defined atoms introduced in expanding are depicted as the arcs e, f and g. Note that by the regularity of rules, each body has at most one defined atom, whose left order constant is the maximal order constant introduced by the body. Thus, by the time this defined atom is encountered in the topological sort, all the constants introduced by the rule have been sorted. This has two consequences. First, once the maximal order constant introduced by a rule has been sorted, all the basic atoms introduced by the rule are in the set E. This is because no basic atom in a rule body involves the rightmost order variable. Secondly, it follows that the number of bodies B_i in the intermediate states never exceeds the number of defined atoms in the original database D, because expansion of each body introduces at most one defined atom and the remaining atoms from the body are in E by the time this is expanded.

Finally, we have atoms which were in the original database but not all of whose order constants have been sorted. These are depicted as arcs c (in which some of the constants are sorted) and h (for which none of the constants have been sorted.) Note that if an atom of type c is defined then its left constant must be the maximal sorted constant. Atoms of type h may be either basic or defined. We write F for the set of atoms of type c and h.

The idea of the succinct representation is now to work with left glue types of the set E. We first turn this set of atoms into a graph G by taking for sources σ (τ) all the object (order) constants of E which also occur in an atom of type $b{-}h$. We take the maximal order source to be the last sorted constant in U. Then instead of the graph G, the bodies B_i and the set F, we may work with $Rep(G,\Phi)$, the bodies B_i and the set F. For, consider the ways in which continuing the process of expansion extends the left graph G. The new atoms generated by this process form a proper right graph H. This is because, firstly, no such atom contains an object constant in E which is not a source. Secondly, every order constant in H falls either to the right of G or is a source of G. Thus, the final model constructed is of the form $G \circ H$. By Lemma 6.3.6, $G \circ H \models \Phi$ if and only if $Rep(G,\Phi) \circ H \models_{po} \Phi$. This means that $Rep(G,\Phi)$ contains all the information required to check whether an intermediate (or final) state satisfies Φ.

We may now describe the procedure for finding a countermodel. We nondeterministically traverse the database from left to right, alternating between sorting and expanding, as before. However, now we compress the sorted portion G as we proceed, storing $Rep(G,\Phi)$ instead. Note that by Lemma 6.3.7, this information is also sufficient to compute the new representative each time a constant is added to the sort. If at any stage we obtain a representative in which Φ holds, we reject. If we complete the expansion without Φ holding in the final representative, we accept.

Example 6.4.2: Let us consider what happens to the run of the linearization process in Example 6.4.1 when we compress the left side as we go, by computing glue types of the query $\exists xyz[A(x,y) \wedge A(x,z) \wedge x{<}y{<}z]$. After the first sorting step we have the configuration

$$\{a \mid \mathbf{R}(a,b), \mathbf{R}(a,c), A(b,d), b{<}c{<}d\} \tag{6.2}$$

There are no atoms on the left, so compression is unnecessary. Expanding and continuing the sort, we obtain the configuration

$$\{A(a,b'), a{<}b' \mid \mathbf{R}(b',b), \mathbf{R}(b',c), A(b,d), b{<}c{<}d\}$$

The only constant shared between the left and right sides is b'. Hence we take the graph representing the left side to be $G = \{: b' \mid A(a,b'), a < b'\}$. The glue type

computation yields the representative $\{: b' \mid \}$ containing no atoms: intuitively, the atom $A(a, b')$ cannot contribute to the satisfaction of the query because right glue types extending G cannot refer to the constant a. For details, we refer the reader back to Example 6.3.5. Thus we may simplify the configuration to

$$\{b' \,!\, \mathbf{R}(b', b), \mathbf{R}(b', c), A(b, d), b{<}c{<}d\}$$

Notice that this is isomorphic to the configuration 6.2, so that the procedure has looped. (While one could in principle detect such loops, our decision procedure will not explicitly do so, but will simply rely on the fact that they must occur, by terminating branches of the search space after a certain number of steps.) The run now continues by expanding both defined atoms using the basic rule, yielding

$$\{b' \,!\, A(b', b''), A(b', c''), A(b, d), b''{<}b{<}c{<}d, c''{<}c\}$$

We now choose the constant b'' as the next point in the linear order, obtaining

$$\{A(b', b''), b'{<}b'' \,!\, A(b', c''), A(b, d), b{<}c{<}d, c''{<}c\}$$

Notice that in this configuration the constant b' occurs on both sides of the configuration, so is one of the order sources of the corresponding graph (the other is b'', since the maximal constant on the left is always a source.) The left hand side of this configuration is in fact a Φ-contribution, so in this case compressing the configuration by replacing the left hand side by its glue type produces exactly the same configuration. Choosing c'' as the next constant in the sort produces the configuration

$$\{A(b', b''), A(b', c''), b'{<}b''{<}c'' \,!\, A(b, d), b{<}c{<}d\}$$

in which the left hand side $\models_{po}$-entails the query, so this "branch" of the computation has failed to produce a countermodel to the query, and there is no need to pursue it further. We leave it to the reader to verify that all other branches also eventually entail the query.

The reason the algorithm described above yields a decision procedure is that the computation may be performed in bounded space. Let us calculate the amount of

space required. The contribution from E is minor, since this set is a subset of the database D. We have already mentioned that there will be at most $|D|$ bodies B_i, so the total contribution from these is of size $|D|.|\Sigma|$. The dominant contribution to space complexity comes from the representatives $Rep(G, \Phi)$. The total number of sources of either type is bounded by the size of F and the B_i, so is $O(|D|.|\Sigma|)$. It now follows from Lemma 6.3.6 that nondeterministic space $O\left((|D|.|\Sigma|)^{2|\Phi|}\right)$ suffices for the computation.

Theorem 6.4.3: If Σ, D is a regular database and Φ is a basic $<$-query then $\Sigma, D \models \Phi$ is decidable in deterministic space $O\left((|D|.|\Sigma|)^{4|\Phi|}\right)$. In particular, basic queries have PSPACE data complexity on regular databases, and EXPSPACE combined and expression complexity.

Since regular programs are linear, and linear programs even without order relations may have EXPSPACE complete combined and expression complexity, we see that the bounds on these forms of complexity stated in Theorem 6.4.3 are tight. This result also shows that no increase in combined and expression complexity is involved in the move from linear programs to regular programs. The following result shows that the upper bound for data complexity of regular programs is tight.

Theorem 6.4.4: There exists a regular program Σ and a $\{<\}$-query Φ such that the complexity of the problem $\Sigma, D \models \Phi$ as D varies is PSPACE complete.

Proof: We show how to encode arrays representing PSPACE computations using regular rules. The remaining details of the proof should by now be familiar from previous reductions, and are left to the reader. Suppose we have k tape symbols, and we wish to represent a computation using n cells. Then we use a defined predicate **seq** of arity $1 + k + 2$ whose last two arguments are order arguments. The first argument represents the cell number and the next k are used to carry the tape symbols. For each $i \leq k$ we have a recursive rule

$$\mathbf{seq}(x, \mathbf{y}, u, w) : -next\text{-}time(u, v), contains(x, v, y_i), \mathbf{seq}(x, \mathbf{y}, v, w), u{<}v{<}w$$

where y_i is the i-th variable in $\mathbf{y}$. These rules are used to generate a sequence of time points between u and w and select at each intermediate time point v one of the cell

symbols y_i as the contents of cell x at time v. We have one basic rule

$$\mathbf{seq}(x, \mathbf{y}, u, w) : -last(u), u{<}w$$

which terminates construction of the sequence of time points for cell x. Given a starting configuration $s_1 \ldots s_n$ we initialize the computation by letting the database contain basic facts

$$contains(c_1, t_0, s_1), contains(c_2, t_0, s_2), \ldots, contains(c_n, t_0, s_n)$$
$$next{-}cell(c_1, c_2), next{-}cell(c_2, c_3), \ldots, next{-}cell(c_{n-1}, c_n)$$

in which the constant c_i represents the i-th cell of the configuration. For each $i \leq n$ the database also contains the defined fact $\mathbf{seq}(c_i, \mathbf{d}, t_0, t_\infty)$ in which $\mathbf{d}$ is a vector of constants representing the possible cell contents. Inspection of the rules shows that these facts generate a linear sequence of facts for the relation $next{-}time$ representing the contents of the cell c_i. Besides these facts the database also contains a number of basic facts encoding the transition relation of the Turing machine, as usual. Notice now that the sequences generated for the cells c_i may be synchronized by including in the query a disjunct

$$\exists tuv[next{-}time(t, u) \wedge next{-}time(t, v) \wedge u{<}v]$$

exactly along the lines of Example 6.2.4. This sets up the array required by the computation. Additional disjuncts then verify the cell contents and check for acceptance of the computation, as usual. □

Recall that data complexity is in co-NP with respect to fixed programs not involving linear order (Theorem 3.5.3.) Thus we see once again that the addition of linear order leads to a sharp increase in complexity.

6.5 Regular Databases with Bounded Concurrency

An examination of the proof of Theorem 6.4.4 shows that in order to simulate a computation using space n we require n defined atoms $\mathbf{seq}(c_i, \mathbf{d}, t_0, t_\infty)$. Informally, we require an unbounded number of atoms which are *concurrent* over the interval (t_0, t_∞). Recall that in the previous chapter the width of an indefinite order database referred

to the number of points potentially mapping to the same point in models. It was found that a bound on the width of the database helps to reduce complexity. In this section we introduce the notion of concurrency of intervals and show that under certain restrictions the data complexity of regular databases drops from PSPACE to PTIME . This result is of interest even in the case that the database contains no defined atoms, since we have seen in Theorem 5.3.2 that order databases containing binary predicates have intractable data complexity. As we mentioned in the discussion after the proof, this is true even for databases with a width bound of two.

We confine ourselves in this section to what might be called "regular interval databases." These are regular databases containing only binary basic and defined predicates of the form $P(u, v)$ where both u and v are order arguments. That is, we now have no object arguments, so that all constants in the database are order constants. Intuitively, the atom $P(u, v)$ asserts that an event of type P occurs during the interval starting at u and ending at v. We will make the assumption for each predicate that the first argument denotes the interval's initial point and the second argument denotes its final point. That is, we assume that whenever a database contains the atom $P(u, v)$ it also contains the atom $u < v$. (But we will generally omit these order atoms when writing databases, leaving their presence implicit.) Besides these atoms, the database may also contain order constraints of the form $x < y$. As usual, we work with the minimal model semantics.

We first introduce the notion of concurrency for databases consisting only of a set of basic and order atoms D. It is convenient to introduce first the notion of an extended minimal model. This is simply a model which may be obtained from a minimal model by extending the linear order, introducing new points, possibly between points in the minimal model being extended. (It is not permitted to introduce new atoms.) We then say that an interval database D has *k-bounded concurrency* if for every extended minimal model M of D and every point t in M there are no more than k atoms $P(u, v)$ in M with $u < t < v$. The *concurrency* of a database will be the minimal k such that the database has k-bounded concurrency. Intuitively, the concurrency of a database is the maximal number of events which may occur in overlapping periods in its minimal

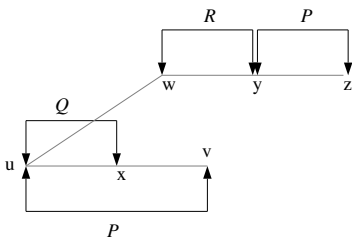

Figure 6.8: A database of concurrency three

models. Note that this is a semantic definition; we will shortly show how to calculate the degree of concurrency of a database without having to examine every minimal model.

Example 6.5.1: The following is a database with concurrency equal to three: (we omit the obvious order atoms)

$$P(u,v), Q(u,x), R(w,y), P(y,z), u<w, x<v$$

This degree of concurrency is attained in the minimal model corresponding to the topological sort $u < w < x < v < y < z$, which has the property that a point t inserted between w and x lies within the intervals $P(u,v)$, $Q(u,x)$ and $R(w,y)$. See Figure 6.8. No minimal model can have all four intervals concurrent, because the interval $R(w,y)$ must precede the interval $P(y,z)$. □

The degree of concurrency of a database D may be calculated by the following procedure. Suppose that G is the set of order atoms of D. For every interval $P(u,v)$ in D we introduce a new order constant x and add the order atoms $u < x < v$ to G. Let X be the set of new constants added, and let G' be the resulting set of order atoms. We then normalize this set of atoms, adding all derived relations between order constants in G' (i.e., we compute the transitive closure). Finally, we take G'' to be the restriction of the normalized set of atoms to the constants in X.

Example 6.5.2: For the database of Example 6.5.1 we add four new points $t_1, \ldots, t_4$ corresponding to the intervals in their order above. We obtain the graph

$$G' = \{u<t_1<v, u<t_2<x<v, u<w<t_3<y<t_4<z\}$$

and when we normalize and restrict to $t_1, \ldots, t_4$ we obtain the graph $G'' = \{t_1, t_2, t_3<t_4\}$. □

Notice that in this example the width of G'' is equal to the degree of concurrency of the original database D. This is in fact a general phenomenon, as the following result shows:

Lemma 6.5.3: The degree of concurrency of D equals the width of the dag G''.

Proof: Suppose first that the concurrency of D is less than or equal to k. We show that the width of G'' is also less than or equal to k. For suppose that this width is greater than k. Choose $k+1$ independent points in G'' and topologically sort G'' so as to map these points to the same point t in the linear order. Arguing exactly as in Lemma 5.2.5, this sort of the points X can be extended to an extended minimal model of D. But then we have the point t contained in $k+1$ intervals, a contradiction.

Conversely, suppose that the concurrency of D is greater than k, and let M be an extended minimal model of D containing a point t in more than k intervals. It is straightforward to see that this model may be extended to a model of $D \cup G'$ by mapping the new "internal constant" x for each of these intervals $P(u,v)$ to the point t, since x is constrained only by the relations $u < x < v$. But this means that we have at least $k+1$ independent points in G''. (Since G'' cannot contain any atom of the form $x < y$ for these k+1 points.) Thus the width of G'' is greater than k also. □

To generalize the notion of degree of concurrency to regular interval databases, notice that each defined fact expands to yield a linear sequence which may itself have contain several concurrent intervals. Thus we need to also take into consideration the degree of concurrency of the bodies of rules. Say that a regular rule

$$\mathbf{R}(t_1, t_k) :- B(t_1, \ldots, t_{k-1}) \wedge t_1 < \ldots < t_k \wedge \mathbf{S}(t_{k-1}, t_k)$$

has concurrency k if the set of atoms $B(t_1, \ldots, t_{k-1}) \wedge t_1 < \ldots < t_k$ has concurrency k. We may then say that a regular interval database Σ, D has k-bounded concurrency if the set D has k-bounded concurrency and each rule of Σ has k-bounded concurrency. Note that this does not mean that minimal models of expansions of k-bounded regular interval databases can have at most k concurrent intervals. However, it remains the case that such models will have a bounded number of concurrent intervals: the appropriate bound is k^2, since each defined atom may contribute k concurrent intervals. With these

definitions, we may establish the following:

Theorem 6.5.4: Basic queries have PTIME data complexity with respect to k-bounded regular interval databases.

Proof: We associate with each k-bounded regular interval database Σ, D the transition graph whose states correspond to configurations of the decision procedure of the previous section, and whose transitions correspond to the moves of this procedure. As before, a query is entailed by the database just in case every terminal configuration accessible from the initial configuration has its left glue type component entail the query. Thus we will be done once we have established that the transition graph has size polynomial in the size of the database.

To see this, recall that the configurations consist of three types of components: a left glue type, a collection of partially sorted rule bodies, and a residue F of the original set of facts D containing those facts which have not yet been sorted. Let us consider first the residue F. Because the set D is k-bounded and we are sorting from left to right, we obtain $O\left(|D|^k\right)$ possible instances of this component. Also by the k-boundedness constraint, at most k atoms of F in any configuration contain a constant from the left side of the configuration, so k sources suffice to connect the left portion to F.

Next, consider the rule bodies. By regularity and by the k-boundedness of D, we never require more than k rule bodies in any configuration, so we have $O\left(|\Sigma|^k\right)$ possible combinations of rule bodies in any configuration. Each body is partially sorted; since the bodies are linear this gives another factor of $O\left(|\Sigma|^k\right)$. Since the rules are k-bounded, we require at most k^2 sources to represent the interaction between the left portion of the configuration and the rule bodies. In fact k^2 sources suffice for all the interaction between the left part of the configuration and the right, since we have either up to k concurrent intervals from a partially expanded defined atom or just one interval from a basic atom. But now since the query is fixed and we require a bounded number of sources, the left portion is represented by a fixed number of possible glue types. This shows that we need only consider $O((|D|.|\Sigma|)^c)$ possible states in the transition diagram, for some constant c. This completes the proof. □

The result may be slightly generalized to databases admitting predicates with object

variables, provided they contain no more than a fixed number of object constants. Notice, however, that the proof above fails if there may be an arbitrary number object constants, since these may require us to maintain an unbounded number of object sources.

6.6 Discussion

As in previous chapters, there is a relationship between the problems we have studied and query optimization problems in deductive databases. The combined complexity problem of this chapter corresponds to the containment problem for Datalog queries containing inequalities. Thus, the main result of this chapter may be interpreted as stating that containment of a regular Datalog query (in which all recursions are regular) in a nonrecurive query is decidable. From this it follows that equivalence of a regular query and a nonrecursive query is decidable.

There have recently been some related studies of the optimization of Datalog programs containing constraints [5, 93, 118], in which the emphasis has been on pushing constraint selections from the defined predicates towards the basic predicates, by means of a combination of rule unfolding, goal reordering and magic sets optimization for bottom up computation. The problem of detecting 'unreachable rules' has also been studied in the context of Datalog with constraints [73], by means of an analysis of the derivation trees generated by such programs. The optimizations performed by these methods are either orthogonal to or weaker than the optimizations corresponding to recursion elimination by detection of an equivalent non-recursive query. On the other hand, these approaches do not suffer from the high degree of complexity of the stronger type of optimization, and are more readily implemented.

We have not been concerned in this chapter with non-recursive definitions, but it is interesting to note that these permit us to express all the relations in Allen's interval calculus. We have already seen that all of Allen's primitive relations may be expressed using conjunctions of inequalities between the endpoints. The complex relations in the interval calculus are all possible disjunctions of the primitive relations. Suppose $B_i(u_1, v_1, u_2, v_2)$ for $i = 1 \ldots 13$ is the conjunction representing that the i-th interval

relation holds between the intervals $[u_1, v_1]$ and $[u_2, v_2]$. Then the interval relation consisting of a disjunction

$$r_{i_1} \vee r_{i_2} \vee \ldots \vee r_{i_n}$$

of some set of primitive interval relations may be represented by the predicate $\mathbf{R}$ defined by the rules

$$\begin{aligned}
\mathbf{R}(u_1, v_1, u_2, v_2) &: -B_{i_1}(u_1, v_1, u_2, v_2) \\
\mathbf{R}(u_1, v_1, u_2, v_2) &: -B_{i_2}(u_1, v_1, u_2, v_2) \\
&\vdots \\
\mathbf{R}(u_1, v_1, u_2, v_2) &: -B_{i_n}(u_1, v_1, u_2, v_2)
\end{aligned}$$

This is simply the representation of disjunction we discussed earlier.

One of the most developed logical frameworks for the representation of action is *dynamic logic* introduced in Pratt [104]. Dynamic logic was originally developed as a formalism for reasoning about programs, but has also been used in the AI literature for the representation of actions [112]. See [51, 67, 52] for surveys of the theory that has been developed in this area. Dynamic Logic associates with each action (program) α a modal operator $\langle\alpha\rangle$, such that if φ is a formula then $\langle\alpha\rangle\varphi$ holds at a state s just when there is some way of performing the action (some execution of the program) α so as to produce a state in which the formula φ holds. Various ways of forming complex actions from the primitive actions have been considered. Generally, dynamic logic is formulated with three such operators:

1. non-deterministic choice: performing $\alpha \cup \beta$ means performing either α or performing β,

2. sequential composition: performing $\alpha;\beta$ means performing α, then performing β, and

3. iteration: performing α^* means performing α some finite number of times.

It is interesting to note that regular rules are able to express the class of regular events that form the basis of most formulations of dynamic logic. The connection between regular rules and the regular expressions follows simply from the fact that

regular events may be represented by left-linear regular grammars [55]. These are grammars in which every production is of the form $R \rightarrow wS$ or $R \rightarrow w$ where R and S are nonterminals, and w is a string of terminals. The connection between these and our regular rules is clear. Suppose that we interpret an atom of the form $A(u, v)$ to mean that the action A occurred over the interval starting at u and ending at v. Then to represent the production $R \rightarrow wS$ with $w = A_1 A_2 \ldots A_k$ we use the regular rule

$$\mathbf{R}(t_0, t_{k+1}) : -A_1(t_0, t_1), A_2(t_1, t_2), \ldots, A(t_{k-1}, t_k), t_0 < t_1 < \ldots < t_{k+1}, \mathbf{S}(t_k, t_{k+1})$$

Regular events have also been considered in the context of temporal logics by Wolper [130], who argues that temporal logics are insufficiently expressive because they are unable to represent regular properties of sequences, and suggests augmenting temporal logic by adding right-linear grammar operators. It is therefore very interesting that the class of definitions for which we are able to prove decidability of basic queries seems to have the minimal expressive power generally considered acceptable in modal logics of time and action.

Dynamic logic, in its original formulation, is founded on a state-based "before-after" semantics which associates to each action a binary relation. A pair of states (s_1, s_2) is in the semantics of the action just when performing the action starting in state s_1 may result in the state s_2. This semantics is inadequate for the representation of concurrent activity, and there is only a weak relation between it and the model of time as linear order with actions as interval predicates suggested in the present work. However, this deficiency has been recognized by some dynamic logicians, and has motivated the development of a number of frameworks for the representation of concurrency in dynamic logic. Dynamic logics have been proposed which model alternating computations [100] and which are built on the shuffle model of concurrency [96]. Most closely related to our work in this dissertation, however, is the family of *process logics* proposed by Pratt [105]. The semantics of process logic associates to an action a set of *sequences* representing the executions of the action. This makes it possible to introduce operators which refer to intermediate states of the computation.

There has been extensive research on process logics [53, 54, 95, 97, 126], the focus

of which has been to find maximally expressive versions of process logic that retain decidability in the propositional case. It is not immediately clear what connection there may be between what is known about decidability for process logics and our results, for a number of reasons. First, we are not concerned with modality in our work, and we study consequence for restricted cases of formulae and data only, rather than the consequence relation for the full modal languages studied in process logic. Secondly, work on process logics has dealt exclusively with the propositional case, whereas we work with a restricted first order syntax. However, we feel that the development of connections of our work to process and dynamic logic would make for an interesting project, but we will not attempt this here.

We have already described in Section 5.6 a connection between non-linear planning and indefinite order databases. The combination of defined relations and linear order is particularly interesting because it is able to represent certain aspects of *hierarchical* non-linear planning [113, 120]. Hierarchical planning is based on the assumption that there is an advantage to planning at higher levels of abstraction first. It is claimed that this helps the planner to avoid being overwhelmed by an explosion of insignificant details, by resolving conflicts at an abstract level before moving on to lower level aspects of the planning problem being solved. Thus, hierarchical planners work with abstract actions which are later expanded into more concrete actions.

There is a clear relationship between expansion of abstract actions and the process of expansion of defined atoms in our indefinite databases, as reflected in Example 6.1.1. In this regard, it is interesting to consider again Allen and Koomens' formulation of planning discussed in Section 5.6, since we found this account of planning be most closely related to our work. The axioms for the actions in this formulation were intended to be used in a hierarchical fashion: Allen and Koomen describe the behaviour of their procedure as first inserting the action **Move**, "expanding" this using the axiom 5.3 (page 152), deriving the interval consequences that follow from the additional interval relations asserted by this expansion step, and then continuing by expanding the actions **Pickup** and **Puton** in the "decomposition" part of the axiom.

Formally, this does not quite correspond to the sort of expansion process evident in

Example 6.1.1. The difference is that the axiom 5.3 models expansion as a deductive inference, whereas, as we have discussed at length above, expansion of defined atoms is more like an abductive process. However, it is clear that it is possible to reformulate Allen and Koomens' account in an abductive fashion simply by reversing the direction of the arrows. It is interesting to note that doing this gives an axiom which has precisely the form of a Horn rule.

By treating expansion as an abductive rather than a deductive process, we are better able to model abstract actions which may be performed in a variety of different ways. The deductive rule 5.3 does not allow, for example, that the move operation might be performed by pushing the block rather than picking it up and then putting it down. Further, we note that the abductive formulation allows us to model recursively decomposable actions correctly. Our arguments against the Clark Completion in Section 3.1 can be invoked once more to show that a first order logic formulation would be inadequate.

There has in fact been an attempt to give an abductive account of planning, but it proceeds along somewhat different lines than we have been suggesting. This is the work of Eshghi [33] who formulates planning as abduction using Horn rules, based on the Event Calculus of Kowalski and Sergot [66]. Eshghi writes Horn rules which formalize the effects of actions, and then uses abduction to implement a regression style planner. This approach has subsequently been simplified and followed up by others [115, 92, 30]. We refer the reader to [86] for further discussion on the relation between the results of the present chapter and AI planning problems.

Chapter 7

Conclusion

While we have stated a large number of results, there are many obvious questions we have left unanswered. In this concluding chapter we enumerate some of these, and suggest some directions for further research.

On the technical side, there are a number of problems we have not attempted to address in this work. The following is a list of questions, answers to which would round out the results of this dissertation, and provide a more complete picture of the complexity of processing the sorts of indefinite information we have studied.

1. We have concentrated throughout on the case of defined facts stated in terms of recursive rules. This has been primarily because we have been interested in identifying those cases in which recursively indefinite information may be decidably queried. However, one expects that defined facts stated in terms of non-recursive rules are likely to occur more frequently in practice. It is therefore desirable to obtain complexity characterizations in the non-recursive case.

2. Related to the previous point, we have not carefully studied here the case of nonrecursive intensional predicates in queries, although we know that these may be added without loss of decidability. As we have already remarked, we have presented a complete characterization of the data complexity of queries containing nonrecursive intensional predicates, since these expand out into basic queries. However, it would be interesting to complete the picture by studying the effect of such intensional predicates on combined and expression complexity. We have conducted a cursory investigation of this issue, and it appears that nonrecursive definitions may result in exponential improvements in query succinctness, resulting in 2–EXPTIME combined complexity even when querying indefinite data

expressed using only non-recursive rules.

3. Another class of queries we know to be decidable in recursively indefinite databases, but for which we have not provided any complexity characterization, is the queries containing transitive closure. This class is of particular interest, since transitive closure is one of the most common forms of recursive rule.

4. When dealing with indefinite order databases, we confined ourselves to databases and queries containing only the relation $<$. It is desirable to complete our characterizations to include the relations $\leq$ and $\neq$. Of particular interest is to determine to what extent the PTIME cases we have identified generalize.

5. The same remark applies to the regular databases. Further, it is not clear that the class of rules for which we have established decidability is the best possible. Notice that basic atoms in regular rules may not contain the "maximal" order variable t_k. This was a technical condition we needed to impose for our proof to work. It seems that it should be possible to relax this, but we do not presently know how.

6. As we noted in Section 5.5, the proof we have given for the PTIME data complexity of monadic $\{<\}$-queries is nonconstructive. Possibly the most difficult problem we have left open is the following: is it possible to compute for each monadic $\{<\}$-query the set of minimal elements of $S(\Phi)$? Without an affirmative answer to this question the algorithm we presented is of little computational value.

Our results have revealed worst-case complexities which are in general very high. It would be interesting to determine to what extent these complexities manifest themselves in a variety of examples. Our decision procedures have been designed to yield optimal worst-case complexity results. Since these algorithms are stated in terms of non-deterministic and alternating computations, it is unclear whether they can be implemented in such a way as to operate efficiently on even simple examples. For the class of problems studied in Chapter 3, the bottom-up fixpoint computation of glue types

described in Section 3.5 seems the most likely candidate for practical implementation. There is probably some scope for exploitation of known optimizations of bottom up computations with this approach.

Also of interest would be to develop incomplete approximation techniques, which may in practice exhibit better behaviour while still solving a variety of problems of interest. A variety of approximations suggest themselves:

1. **Inductive proofs:** It would be interesting to compare the performance of implementations of the decision procedure with that of theorem provers using heuristic techniques to construct inductive proofs [10, 11].

2. **Completion:** The semantics we have adopted for recursively indefinite databases is stronger than the completion semantics, as we have already remarked. That is, $\Sigma, D \models Comp(\Sigma, \mathcal{Q}) \cup D$, where $\mathcal{Q}$ is the set of defined predicates. This means that the completion can be used as a first order approximation. We note however, that it is not immediate clear whether this will result in complexity improvements. It may well be that the completion is equally (or more) complex to query, in spite of the fact that it is first order.

3. **Partial Order:** When dealing with indefinite order databases, the linear order semantics may be approximated by interpreting the database over all partially ordered models instead of linearly ordered models. In the context of indefinite order databases not containing defined facts, this reduces the data complexity from co-NP to PTIME. It is also interesting to note that basic queries containing $<$ are decidable in *all* recursively indefinite databases, not just the regular ones, provided we interpret these with respect to partially ordered structures rather than linearly ordered structures. This follows directly from the decidability of queries containing transitive closure. This provides further motivation to develop complexity results for transitive closure queries.

Inductive proofs are interesting in that they are able to handle certain examples, involving intensional queries and rules with function symbols, that are beyond the

scope of our decision procedures. We note, however, that the undecidability of intensional queries, together with the recursive enumerability of the complement of the query problem, implies that there cannot be a complete inductive proof theory. On the other hand, the decidability results suggest that one may obtain weak completeness results that state that a proof theory is complete for the inference of a certain class of formulae. This raises the following interesting question: What is the simplest form of inductive proof theory that is complete for, say, the basic queries? See [84, 85] for a very restricted, logic programming style formulation of inductive proofs, capable of handling a variety of examples of interest. It would be very interesting if some such proof theory could be shown to be complete.

We mentioned in the previous chapter that regular rules are related to the regular events commonly used in dynamic and process logics. It is very interesting to note in the present context that one of the axioms of the Segerberg axiomatization of dynamic logic is in fact a sort of induction axiom. We would like to better understand the relation between the decidabilty of dynamic logic and our results, if any.

Finally, we have already suggested in Chapter 6 that recursive definitions using linear order may provide an attractive framework in which to formalize AI theories of planning and reasoning about actions. Getting this right is likely to be a very complex task, but we believe that there will be something to be learned from investigating the application of our formalisms to the planning domain.

References

[1] S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78:159–187, 1991.

[2] J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:510–521, 1983.

[3] J. F. Allen and J. A. Koomen. Planning using a temporal world model. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 741–747, 1983.

[4] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufman, 1989.

[5] I. Balbin, D.B. Kemp, K. Meenakshi, and K. Ramamohanarao. Propagating constraints in recursive deductive databases. In *Proceedings of the North American Conference on Logic Programming*, pages 981–1000, 1989.

[6] F. Bancilhon and N. Spyratos. Update semantics and relational views. *ACM Transactions on Databases*, 6(4):557–575, 1981.

[7] M. Baudinet, M. Niezette, and P. Wolper. On the representation of infinite temporal data and queries (extended abstract). In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 280–290, 1991.

[8] P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6(3):132–144, 1990.

[9] T. Bench-Capon, editor. *Knowledge-Based Systems and Legal Applications*. Academic Press, 1991.

[10] R. Boyer and J. Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.

[11] A. Bundy, F. van Harmelen, and J. Hesketh. Extensions to the the rippling-out tactic for guiding inductive proofs. In *Proceedings CADE-10*. Springer LNCS No. 449, 1990.

[12] T. Bylander, D. Allemang, M.C. Tanner, and J.R. Josephson. The computational complexity of abduction. *Artificial Intelligence*, 49:25–60, 1991.

[13] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[14] A.K. Chandra and P.K. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 77–90. Association for Computing Machinery, 1976.

[15] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1985.

[16] S. Chaudhuri and M. Vardi. On the equivalence of recursive and non-recursive programs. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 55–66, 1992.

[17] J. Chomicki. Polynomial time query processing in temporal deductive databases. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 379–389, 1990.

[18] J. Chomicki and T. Imielinski. Temporal deductive databases and infinite objects. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 61–73, 1988.

[19] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[20] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):379–434, 1979.

[21] S. Cosmadakis, H.Gaifman, P. Kanellakis, and M. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 477–490, 1988.

[22] B. Courcelle. An axiomatic definition of context-free rewriting and its application to NLC graph grammars. *Theoretical Computer Science*, 55:141–181, 1987.

[23] B. Courcelle. On using context-free graph grammars for analyzing recursive definitions. In K. Fuchi and L. Kott, editors, *Programming of Future Generation Computers II Proceedings of the Second Franco-Japanese Symposium on Programming of Future Generation Computers*, pages 83–122. North Holland, 1988.

[24] B. Courcelle. Recursive queries and context-free graph grammars. *Theoretical Computer Science*, 78:217–244, 1989.

[25] B. Courcelle. The monadic second order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[26] P.T. Cox and T. Pietrzykowski. Causes for events: their computation and application. In J.H. Siekmann, editor, *Proceedings of CADE-86*, pages 608–621. Springer LNCS No. 230, 1986.

[27] T. Dean and M. Boddy. Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–387, 1988.

[28] T. Dean and D. McDermott. Temporal database management. *Artificial Intelligence*, 32:1–55, 1987.

[29] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[30] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In *Proceedings of the European Conference on Artificial Intelligence*, 1992. to appear.

[31] C. Elkan and D. McAllester. Automated inductive reasoning about logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 876–892. The MIT Press, 1988.

[32] M.H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[33] K. Eshghi. Abductive planning with event calculus. In R.A. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 562–579. The MIT Press, 1988.

[34] K. Eshghi and R. Kowalski. Abduction compared with negation as failure. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the Sixth International Conference*, pages 234–254, Boston, MA, 1989. The MIT Press.

[35] M.R. Fellows and M.A. Langston. Nonconstructive advances in polynomial time complexity. *Information Processing Letters*, 26:157–162, 1987.

[36] M.R. Fellows and M.A. Langston. Nonconstructive tools for proving polynomial time decidability. *Journal of the ACM*, 35:727–739, 1988.

[37] J.A. Fernandez and J. Minker. Bottom-up evaluation of hierarchical disjunctive deductive databases. In K. Furukawa, editor, *Logic Programming: Proceedings of the Eighth International Conference*, pages 660–675. The MIT Press, 1991.

[38] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[39] H. Gaifman, H. Mairson, Y. Sagiv, and M. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 106–115, 1987.

[40] A. Galton, editor. *Temporal Logics and their Applications.* Academic Press, 1987.

[41] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230, 1988.

[42] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R.A. Kowalski and K.A. Bowen, editors, *Logic Programming: Proc. Fifth International Conference and Symposium*, pages 1070–1080. The MIT Press, 1988.

[43] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[44] M.C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: A graph-theoretic approach. Technical Report RRR No. 22-91, RUTCOR: Rutgers Center for Operations Research, New Brunswick NJ, May 1991.

[45] G. Grahne. Horn-tables - an efficient tool for handling incomplete information in databases. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 75–82, 1989.

[46] Y. Gurevich and H. Lewis. The inference problem for template dependencies. In *Proceedings of the First ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–229, 1982.

[47] A. Habel and H.-J. Kreowski. May we introduce to you: Hyperedge replacement. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and their Applications to Computer Science 3rd International Workshop*, pages 15–26. Springer LNCS No. 291, 1987.

[48] A. Habel and H.-J. Kreowski. Some structural aspects of hypergraph languages generated by hyperedge replacement. In F.J. Brandenburg, G. Vidal-Vaquet, and M. Wirsig, editors, *STACS'87 4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 207–219. Springer LNCS No. 247, 1987.

[49] J. Halpern and Y. Shoham. A propositional modal logic of time intervals. In *Proceedings of the Symposium on Logic in Computer Science*, pages 279–292, 1986.

[50] G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers.* The Clarendon Press, Oxford, 1938.

[51] D. Harel. *First Order Dynamic Logic.* LNCS No. 68. Springer-Verlag, 1979.

[52] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, II: Extensions of Classical Logic*, pages 497–604. Reidel, Boston, MA, 1984.

[53] D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability and completeness. *Journal of Computer and System Sciences*, 25, 1982.

[54] D. Harel and D. Peleg. Process logic with regular formulas. *Theoretical Computer Science*, 38:307–322, 1985.

[55] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[56] E. Horowitz and S. Sahni. *Data Structures in Pascal.* Computer Science Press, Rockville, MD, 1984.

[57] T. Imielinski. Incomplete deductive databases. *Annals of AI and Mathematics*, 3(II-IV), 1991.

[58] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[59] T. Imielinski and K. Vadaparty. Complexity of query processing in databases with or-objects. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 51–65, 1989.

[60] F. Kabanza, J-M. Stevenne, and P. Wolper. Handling infinite temporal data. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 392–403, 1990.

[61] T. Kanamori and H. Fujita. Formulation of induction formulas in verification of PROLOG programs. In J.H. Siekmann, editor, *Eighth International Conference on Automated Induction*, pages 281–299. Springer-Verlag, 1986.

[62] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, 1990.

[63] D.G. Kendall. Some methods and problems in statistical archeology. *World Archeology*, pages 68–76, 1969.

[64] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[65] P. K. Kolaitis and C. H. Papadimitriou. Some computational aspects of circumscription. *Journal of the ACM*, 37(1):1–14, January 1990.

[66] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[67] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. B: Formal Models and Semantics*, pages 789–840. Elsevier, Amsterdam, 1990.

[68] J.B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory (Ser. A)*, 13:297–305, 1972.

[69] P.B. Ladkin. Metric constraint satisfaction. Technical Report TR-89-038, International Computer Science Institute, 1989.

[70] J.L. Lassez. Querying constraints. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 288–298, 1990.

[71] T. Lengauer and E. Wanke. Efficient analysis of graph properties on context-free graph languages (extended abstract). In T. Lepisto and A. Salomaa, editors, *Proceedings of ICALP'88*, pages 379–393. Springer LNCS No. 317, 1988.

[72] H. Levesque. A knowledge level account of abduction. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 1060–1067, Detroit, MI, 1989.

[73] A. Levy and Y. Sagiv. Constraints and redundancy in Datalog. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 67–80, 1992.

[74] V. Lifschitz. Computing circumscription. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 121–127, 1985.

[75] V. Lifschitz. Pointwise circumscription. In M.L. Ginsburg, editor, *Readings in Non-Monotonic Reasoning*, pages 179–193. Morgan Kaufman, 1988.

[76] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, second edition, 1987.

[77] D.W. Loveland. Near-Horn Prolog. In J-L. Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference*, pages 456–469. The MIT Press, 1987.

[78] D.W. Loveland. Near-horn prolog and beyond. *Journal of Automated Reasoning*, 7:1–26, 1991.

[79] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[80] J.A. Makowsky. Why Horn formulas matter in computer science: Initial structures and generic examples. *Journal of Computer and System Sciences*, 34:266–292, 1987.

[81] J. Malik and T.O. Binford. Reasoning in time and space. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 343–345, 1983.

[82] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[83] J. McCarthy. Applications of circumscription to non-monotonic reasoning. *Artificial Intelligence*, 28:89–116, 1986.

[84] L.T. McCarty. Computing with prototypes. Technical Report LRP-TR-22, Computer Science Department, Rutgers University, 1990. A preliminary version of this paper was presented at the *Bar Ilan Symposium on the Foundations of Artificial Intelligence*, Ramat Gan, Israel, June 1989.

[85] L.T. McCarty and R. van der Meyden. Indefinite reasoning with definite rules. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 364–378, 1991.

[86] L.T. McCarty and R. van der Meyden. Reasoning about indefinite actions. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 1992. to appear.

[87] R. van der Meyden. Reasoning with recursive relations: negation, inequality and linear order. In *Proceedings of the ILPS'91 Workshop on Deductive Databases (mimeo)*, pages 62–71, 1991.

[88] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 331–345, 1992.

[89] R. van der Meyden. Recursively indefinite databases. *Theoretical Computer Science*, to appear. an earlier version of this work appears in: ICDT'90: Proceedings of the International Conference on Database Theory, S. Abiteboul and P.C. Kanellakis (eds.) Springer LNCS No. 470, pp. 364-378, 1990.

[90] J. Minker. On indefinite databases and the Closed World Assumption. In D. Loveland, editor, *Proceedings of the Sixth Conference on Automated Deduction*, pages 292–308. Springer LNCS 138, 1982.

[91] J. Minker and A. Rajesekar. A fixpoint semantics for disjunctive logic programs. *Journal of Logic Programming*, 9(1):45–74, 1990.

[92] L. Missiaen. *Localized Abductive Planning with the Event Calculus*. PhD thesis, Katholieke Universiteit Leuven, 1991.

[93] I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 314–330, 1990.

[94] J. Naughton and Y. Sagiv. Minimizing expansions of recursions. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures I Algebraic Techniques*, pages 321–350. Academic Press, 1989.

[95] H. Nishimura. Descriptively complete process logic. *Acta Informatica*, 14:359–369, 1980.

[96] H. Nishimura. Arithmetical completeness in first order dynamic logic for concurrent programs. *Publ. RIMS, Kyoto Univ*, 17:297–309, 1981.

[97] R. Parikh. A decidability result for second order process logic. In *Proceedings of the IEEE Symposium on the Foundations of Computer Science*, pages 178–183, 1978.

[98] J. Pearl. Fusion, propagation and structuring in belief networks. *Artificial Intelligence*, 29:241–288, 1986.

[99] C.S. Peirce. Abduction and induction. In J. Buchler, editor, *Philosophical Writings of C.S. Peirce*, chapter 11, pages 150–156. Dover, New York, 1955.

[100] D. Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34(2):450–479, 1979.

[101] D. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[102] D. Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence*, 5:97–110, 1989.

[103] H.E. Pople. On mechanization of abductive logic. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 147–152, Stanford, CA, 1973.

[104] V. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, October 1976.

[105] V. Pratt. Process logic. In *Proceedings 6th ACM Symposium on Principles of Programming Languages*, pages 93–100, 1979.

[106] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufman, 1989.

[107] E.M. Reingold. On the optimality of some set algorithms. *Journal of the ACM*, 19:649–659, 1972.

[108] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, 1978.

[109] R. Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie, J. Mylopolous, and J.W. Schmidt, editors, *On Conceptual Modelling*, pages 163–189. Springer-Verlag, 1984.

[110] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[111] D.J. Rosenkrantz and H.B. Hunt. Processing conjunctive predicates and queries. In *Proceedings of the Sixth International Conference on Very Large Databases*, pages 64–72, 1980.

[112] S. Rosenschein. Plan synthesis, a logical perspective. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 331–337, 1981.

[113] E.D. Sacerdoti. *A Structure for Plans and Behaviour.* Elsevier, New York, 1977.

[114] W. Savitch. Relationships between non-deterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[115] M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1055–1060, 1989.

[116] O. Shmueli. Decidabilty and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 237–249, 1987.

[117] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of AI and Mathematics*, 1992. to appear.

[118] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 301–315, 1992.

[119] R.B. Stam and R. Snodgrass. A bibliography on temporal databases. *Database Engineering*, 7:231–239, 1988.

[120] A. Tate. Generating project networks. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 888–839, 1977.

[121] J.D. Ullman. *Principles of Database and Knowledge Base Systems, volume II: The New Technologies*. Computer Science Press, 1989.

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

[123] M. Vardi. Querying logical databases. *Journal of Computer and System Sciences*, 33:142–160, 1986.

[124] M. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 341–350, 1988.

[125] M. Vardi. Automata theory for database theoreticians. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 83–92, 1989.

[126] M. Vardi and P. Wolper. Yet another process logic. In *Proceedings of the Workshop on Logics of Programs*, pages 501–512. Springer LNCS No. 164, 1983.

[127] M.Y. Vardi. The implication and finite implication problem for typed template dependencies. In *Proceedings of the First ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–238, 1982.

[128] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI: Proceedings of the National Conference in Artificial Intelligence*, pages 377–382. Morgan Kaufman, 1986.

[129] E. Wanke. The complexity of connectivity problems on context-free graph languages. In J. Csirik, J. Demetrovics, and F. Gecseg, editors, *Fundamentals of Computation Theory, Proceedings International Conference FCT'89*, pages 470–479. Springer LNCS No. 380, 1989.

[130] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1,2):72–93, 1983.

[131] A. Yahya and L.J. Henschen. Deduction in non-horn databases. *Journal of Automated Reasoning*, 1(2):141–160, 1985.

Vita

Ronald van der Meyden

1983 B.A. with Honours in Pure Mathematics, Sydney University, Sydney, New South Wales, Australia.

1985 M.A. with Honours in Pure Mathematics, Sydney University, Sydney, New South Wales, Australia.

1986-89 Teaching Assistant, Department of Computer Science, Rutgers, the State University, New Brunswick, New Jersey.

1990 R. van der Meyden. The Dynamic Logic of Permission. In *Proceedings of the IEEE Symposium on Logic in Computer Science,* pages 72-78, Philadelphia, Pennsylvania.

1990 R. van der Meyden. Recursively Indefinite Databases. In *Proceedings of the International Conference on Database Theory,* pages 364-378, Paris, France.

1991 Visiting Part-time Lecturer, Department of Computer Science, Rutgers, the State University, New Brunswick, New Jersey.

1991 L. T. McCarty and R. van der Meyden. Indefinite Reasoning with Definite Rules. In *Proceedings of the International Joint Conference on Artificial Intelligence,* pages 890-896, Sydney, Australia.

1991 R. van der Meyden. A Clausal Logic for Deontic Action Specification. In *Proceedings of the International Logic Programming Symposium,* pages 221-238, San Diego, California.

1992 Visiting Fellow, University of New South Wales, Kensington, New South Wales, Australia.

1992 R. van der Meyden. The Complexity of Querying Indefinite Information about Linearly Ordered Domains. In *Proceedings of the ACM Symposium on Principles of Database Systems,* San Diego, California.

1992 L.T. McCarty and R. van der Meyden. Reasoning about Indefinite Actions. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning,* Boston, Massachusets.

1992 Ph.D. in Computer Science, Rutgers, the State University.