Recursively Indefinite Databases

Ron van der Meyden
meyden@paul.rutgers.edu

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903, USA

To appear in: *Theoretical Computer Science*

Abstract: We define *recursively indefinite databases*, a new type of logical database in which indefinite information arises from partial knowledge of the fixpoint of a Datalog program. Although, in general, query answering is undecidable, we show that queries containing only basic predicates and monadic defined predicates are decidable. The main contribution of the paper is an analysis of the complexity of query answering for this class of queries. We demonstrate a class of databases which generalizes disjunctive databases, but without increasing data complexity. We also establish connections with the theory of hypergraph edge replacement graph grammars.

1 Introduction

Logical databases permit the representation of various types of indefinite information. If in addition to ground atomic facts we allow existentially quantified conjunctions of atoms we obtain a class of logical databases which can represent the null values familiar from relational databases [28]. Another extension is obtained by allowing various sorts of disjunctive formulae in the database [12, 13]. In this paper we introduce the class of recursively indefinite databases, which generalize both of these sorts of databases. Recursively indefinite databases are able to express indefinite information that is intrinsically second order. This increased expressive power comes at a cost: in general, query processing is undecidable. Nevertheless, interesting classes of queries remain decidable. Further, we will show that under reasonable constraints on the type of recursively indefinite information allowed in the database, these queries have data complexity no higher than their data complexity with respect to databases containing null values or disjunction.

The intuition underlying recursively indefinite databases is that information frequently comes expressed in terms of predicates defined over the actual relations of interest. In order to interpret the meaning of the facts stated using defined predicates, it is necessary to "unpack the definition". This results in indefiniteness when the definition consists of several clauses, each of which provides an alternative explanation for the defined fact. If the definitions are stated using recursive rules, then the "unpacking" process results in an infinite number of possible explanations of the defined fact. Nevertheless, it may be possible to infer conclusions that hold under all of these alternatives.

Example 1.1: Let G be a directed graph some of whose nodes are coloured red or green. A 'red-green path' is any path through G each of whose nodes is coloured 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':

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

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 coloured red and b is coloured 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. That is, all the defined fact $\mathbf{rgpath}(a, b)$ tells us is that one of an infinite number of possibilities holds: for some $n \geq 0$ there exist nodes $x_1, x_2, \ldots, x_n$ such that $a, x_1, x_2, \ldots, x_n, b$ is a path in G, where each of these nodes is coloured either red or green. Nevertheless, this is sufficient information for us to conclude that the graph contains an edge from a red node to a green node: since a is red and b is green, there must be a transition from red to green somewhere along the path. (A simple inductive proof formalizes this argument.) Thus, on the basis of the information $red(a), green(b), \mathbf{rgpath}(a, b)$, 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 $red, green, edge$. □

The sort of indefinite information we are contemplating in this example consists of the following. First, we have a set of rules Σ which define certain predicates in terms of other 'basic' predicates. Second, we have a collection of facts stated using both the defined and the basic predicates. The basic facts B are interpreted under the 'open world assumption'. That is, we do not assume that the basic relations are completely known; rather, the basic facts B tell us only some of what is true in the actual state of the world. The defined facts D are interpreted according to the definitions Σ. The program provides us with a way to determine, given a state of the world, expressed using the basic predicates, which defined facts hold. In each state of the world, a defined fact holds only if it follows, using the definitions, from the basic facts holding in that state. In other words, the defined predicates denote in each state the relations obtained by computing the least fixpoint of the Datalog query over the basic facts. Thus, knowledge of a defined fact may provide additional information about the basic facts: these must be such as to make the definition of the defined fact true. However, this extra information may be indefinite: there may be many configurations of the basic facts which satisfy the definition, corresponding to different ways to expand out the definition. Because the definitions may involve recursive rules, expanding the definitions leads to the recurrence of indefiniteness in the expansion process, so we call the combination of the progam Σ and the facts $[B, D]$ a *recursively indefinite database*. The topic of this paper is: how complex is it to answer queries on the basis of this sort of indefinite information?

Example 1.2: 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

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

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

Because we interpret a defined fact to hold in a state of the world only if it may be derived, using the program, from the basic facts true in the state, if a defined fact holds then there must be some way to derive it. That is, there must be a rule that was used to derive the fact. This means that according to our semantics, the database $\Sigma, [B, D]$ satisfies $Comp(\Sigma)$, the "Clark completion" [3] of the program Σ. This is the first order theory obtained by turning the rules, which provide *sufficient* conditions for a defined fact, into *necessary and sufficient* conditions. For example, the completion of the program of Example 1.1 is the formula

$$\begin{array}{l}\forall x[\mathbf{rg}(x) \equiv \{red(x) \vee green(x)\}] \wedge\\ \forall xy[\mathbf{rgpath}(xy) \equiv \{(\mathbf{rg}(x) \wedge edge(x,y) \wedge \mathbf{rg}(y)) \vee\\ \qquad \exists z(\mathbf{rg}(x) \wedge edge(x,z) \wedge \mathbf{rgpath}(z,y))\}]\end{array} \tag{1}$$

In the case that the program is non-recursive, the completion may be shown to provide the semantics of the database, in the sense that the information expressed by the database $\Sigma, [B, D]$ is precisely that expressed by the first order theory $Comp(\Sigma) \cup B \cup D$.

This means that recursively indefinite information generalizes both null values and disjunctive information in 'open world' databases. For, the atom $r(a, @)$ where @ is a null value can be represented by the defined fact $\mathbf{R}(a)$, given the definition $\mathbf{R}(x) : -r(x,y)$. 'Marked' null values, for which each occurrence is to be interpreted as denoting the same object, are handled similarly, by placing all atoms in which the null occurs on the right hand side of the rule. The disjunction $A_1 \vee \ldots \vee A_n$ may be represented by the defined fact $\mathbf{A}$, given the definition

$$\begin{array}{c}\mathbf{A} : -A_1\\ \vdots\\ \mathbf{A} : -A_n.\end{array}$$

However, when the program Σ is recursive, the completion is weaker than the semantics we have described. The query of Example 1.1 does not follow under the completion semantics. To see this, consider the model in which we have an infinite sequence

$a = x_0, x_1, \ldots x_n \ldots, x_\omega = b$ of vertices, where x_ω is a point at infinity. There is an $edge(x_i, x_{i+1})$ holds for all natural numbers i. The colours of the vertices are given by $green(b)$, and $red(x_i)$ for all natural numbers i. Finally, $\mathbf{rgpath}(x_i, x_j)$ holds for all $i < j$, and in particular, when i is a natural number and $j = \omega$. This model satisfies the formula (1) and the basic and defined facts of Example 1.1, but not the query of that example.

In fact, it is possible to show that there is no first order theory that captures precisely the semantics we have described. The information expressed by recursively indefinite databases is, in general, intrinsically second order. Since second order logic is known to suffer from logical incompleteness, this raises the question of whether it is possible to provide even a recursively enumerable proof theory for query processing in recursively indefinite databases. We will provide both positive and negative answers to this question.

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, and the set of intensional queries entailed by a database may not be recursively enumerable. However, if we restrict to basic queries, the query problem becomes decidable. One can also show decidability for a slightly larger class of queries, the *monadic* queries, which may contain monadic defined predicates. All of these decidability results can be shown to follow from a general result of Courcelle [7] concerning context-free graph grammars. Our main contribution in this paper is to determine the exact complexity of various decision problems.

Closely related to recursively indefinite databases are the logical databases with 'skolem rules' of Imielinski [12]. These are logical databases containing rules of the form $\varphi \Rightarrow \exists x(\psi)$ where φ, ψ are conjunctions of atoms, and allowing recursion. Imielinski shows the undecidability of queries in logical databases containing skolem rules, but also provides some conditions under which query processing is decidable. Our work in this paper is not immediately applicable to skolem rules, because, like the Clark completion, skolem rules are satisfied in models containing 'infinite chains', whereas defined atoms in recursively indefinite databases must be supported in each model by a 'finite derivation'. However, we suspect that the techniques of the present paper could be applied to obtain new results about databases with skolem rules.

Recursively indefinite databases are able to express some of the informational states associated with updates to views defined over relational databases using recursive Datalog queries. In this interpretation the basic predicates correspond to the relations of the physical database and the defined predicates correspond to the views over this database, where the rules provide the view definition. If updates are assumed to add information to the database only, i.e., only updates completing the information in the database are permitted, not revisions or deletions, then insertions into the views result in exactly the sort of indefinite states of information expressed by recursively indefinite databases.

Recursively indefinite databases are also related to problems which have been studied in the artificial intelligence literature. The constraint that the intensional predicates are to be interpreted as being computed from the base predicates using the program may be stated in terms of McCarthy's notion of circumscription [19, 20] 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. Recursively indefinite databases result from taking the theory Σ to be a logic program with defined relations $\mathcal{Q}$. Circumscription of logic programs has been previously studied by Kolaitis and Papadimitriou [15]. 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 recursively indefinite database $\Sigma, [B, D]$ is equivalent to the theory $\mathit{Circum}(\Sigma, \mathcal{Q}) \cup B \cup D$.

Because the circumscription is a second order formula, there is no immediately applicable proof theory. Most of the research on proof theories for circumscription has concentrated on finding conditions under which the circumscription of a theory is equivalent to a first order theory [17, 18, 15], so that first order theorem proving technology may be applied. Kolaitis and Papadimitriou [15] show that the circumscription of a Datalog program is first order just in case the program is bounded. Since boundedness of programs is undecidable [9], 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 established below show that a different approach to reasoning in circumscribed theories is possible in certain cases. We provide a broad class of queries which are decidable in *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 recursively enumerable proof theory.

Example 1.3: Queries in recursively indefinite databases are related to verification problems for Datalog programs. For example, suppose that transitive closure has been defined using the following linear Datalog program.

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

One might wish to verify that the relation computed is in fact transitively closed, i.e., that it satisfies the formula

$$\forall xyz[(\mathbf{R}(x, y) \wedge \mathbf{R}(y, z)) \Rightarrow \mathbf{R}(x, z)]$$

It is possible to show that this can be done by asking the query $\mathbf{R}(a, c)$ of the database with an empty set B of basic atoms and defined atoms $D = \{\mathbf{R}(a, b), \mathbf{R}(b, c)\}$. $\square$

A number of authors have considered using inductive proofs to do such reasoning about logic programs [8, 14]. In certain cases inductive proofs can be automated using a version of second order intuitionistic logic programming [21, 22]. We will show in Section 7 that this program is in a class of programs for which such properties may be verified automatically. (However, we do not provide any complexity results for this case.)

We now summarize the complexity results we present for our decidable classes of queries. One of the measures of complexity we consider is data complexity, the complexity of evaluating a fixed query as a function of the size of the database. The definition of data complexity we work with is slightly unusual, in that we consider the program Σ to be a part of the data. However, this perspective is natural once one takes the point of view that the program is

Program Type	Complexity Type		
	Data	Expression	Combined
arity(k)	co-NP	2-EXPTIME	2-EXPTIME
Linear	PSPACE	EXPSPACE	EXPSPACE
All Datalog	EXPTIME	2-EXPTIME	2-EXPTIME

Table 1: Complexity of Monadic Queries

part of the mechanism used to express indefinite information. In any case, we also study the complexity of queries with respect to a fixed program.

It is known that the data complexity of positive existential (basic) queries in databases containing even very simple sorts of disjunctive information is co-NP complete [1, 12, 28]. 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 interesting result shows that if one is prepared to live with the complexity of dealing with indefinite information, there is no extra cost in admitting the extra expressiveness of some types of recursively indefinite information.

We also analyze the data complexity for databases whose defined predicates have unbounded arity. In addition we consider other measures of query answering complexity: expression complexity (the complexity of answering queries in a fixed database) and combined complexity (in which all parameters vary). Table 1 summarizes our results. Each entry indicates a class for which the corresponding problem is complete under logspace reductions. These characterizations apply to both basic and monadic queries. Note 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.

Section 2 formally defines recursively indefinite databases and the classes of queries we study. In Section 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 4 we introduce the ideas underlying the decidability of monadic queries. Section 5 establishes upper bounds on the complexity of monadic queries and Section 6 discusses lower bounds. In Section 7 we return to intensional queries and show that there exist other interesting classes of intensional queries which are decidable.

2 Recursively Indefinite Databases

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 1.1–1.3 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 [27] 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

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

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 2.1: Let $\mathbf{D}$ be the recursively indefinite database with program Σ equal to the program for transitive closure of Example 1.3, 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 [30]. For the unique names semantics, things are a little more subtle.

Example 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 $\mathbf{D} \models \exists x[A(x) \wedge B(x)]$. $\Box$

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

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 [10, 11], are generalizations of context free grammars, generating sets of hypergraphs instead of sets of strings. We present Courcelle's [5] slightly more general version of these grammars. Let $\mathcal{A}$ be an alphabet with a rank function $\tau : \mathcal{A} \rightarrow \mathbf{N}$. 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', $lab : E \rightarrow \mathcal{A}$ associates a label from $\mathcal{A}$ with each hyperedge, $vert : E \rightarrow V^*$ 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 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 3.1: Figure 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.

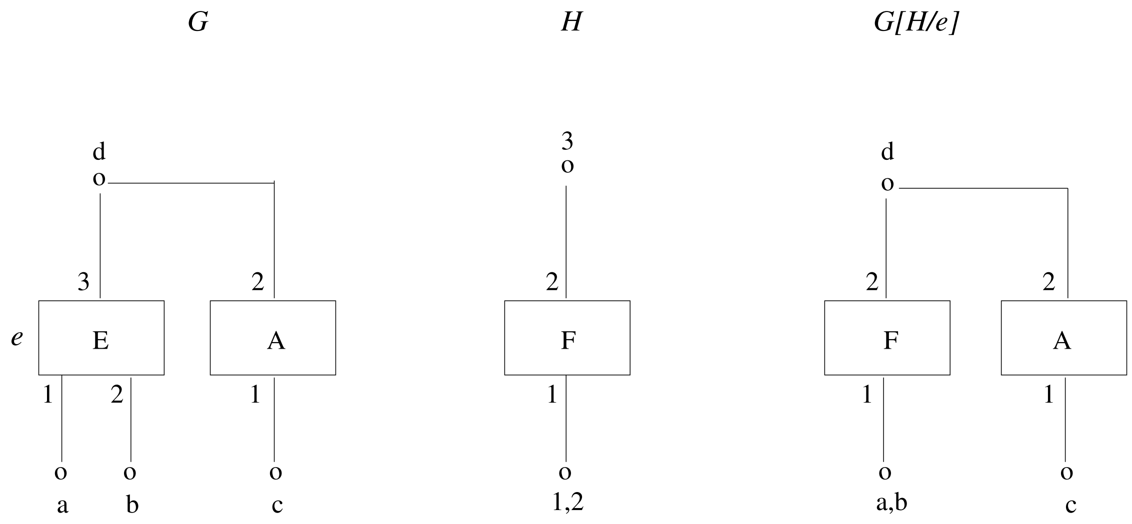

Figure 1: Substituting a graph for an edge

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 3.2: Let Σ be the program of Example 1.1. 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 $L(\Sigma, \mathbf{rgpath})$. □

Hypergraphs and structures are closely related. Any 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 $j = 1 \dots n$. If A is a basic predicate the atom $A(u_1 \dots u_l) \in I$ just when there exists an edge e with $lab(e) = A$ and $vert(e) = u_1 \dots u_l$. If $\mathbf{R}$ is a defined predicate the atom $\mathbf{R}(u_1 \dots u_l) \in J$ when there exists an edge e with $lab(e) = \mathbf{R}$ and $vert(e) = u_1 \dots u_l$. 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 $j = 1 \dots n$ 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 $vert(e) = u_1 \dots u_k$ for each basic or defined atom $A(u_1 \dots u_k)$ in $I \cup J$. The sources are given by $src = i(c_1) \dots i(c_n)$. 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 $vert(e) = u_1 \dots u_n$ as the atom $A(u_1 \dots u_n)$. 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 $c_1 \dots c_m$. 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 $z_1 \dots z_m$. First, replace each occurrence of a constant c_j in r by the variable z_j. Then, replace each occurrence of a defined atom $\mathbf{R}(u_1 \dots u_n)$, in either the head or the body of the rule, by the atom $\mathbf{R}'(u_1 \dots u_n, z_1 \dots z_m)$. 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 $\mathbf{R}(u_1 \dots u_n)$ by $\mathbf{R}'(u_1 \dots u_n, c_1 \dots c_m)$. (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 $\mathbf{R}'(u_1 \dots u_n, c_1 \dots c_m)$ for each defined atom $\mathbf{R}'(u_1 \dots u_n)$ in D. Then it is straightforward to show the following:

Lemma 3.1: $\Sigma, [B, D] \models \varphi$ 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: Consider the program of example 2.2. This contains one constant b. The transformed program Σ' is

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

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

$$\begin{aligned} \{xz_1 : \mathbf{R}'(x, z_1)\} &\longrightarrow \{xz_1 : \mathbf{S}'(x, z_1, z_1)\} \\ &\longrightarrow \{z_1 z_1 : B(z_1)\} = G \end{aligned}$$

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 3.2: 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: 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 2.1 and Lemmas 3.2 and 3.3. 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^0_1.

Lemma 3.4: 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 [25] stating that containment of binary Datalog queries is undecidable. Gaifman et al. [9] developed techniques to prove undecidability of boundedness of Datalog programs which may be used to show that containment remains undecidable for linear programs. Vardi [29] 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 3.5: 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

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

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

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

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.

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 3.4 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 1.3 expresses the disjunction

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

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 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 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 $G_2 \circ H' \models \varphi$. $\square$

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 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

$$\delta = \exists \mathbf{xyz}[\delta_1(\mathbf{x}, \mathbf{y}) \wedge \delta_2(\mathbf{y}, \mathbf{z})] \tag{2}$$

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 4.1: 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 4.4: 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 4.5: 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 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 2.1. 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 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 $2^{|\varphi|(|\Pi|+log\, k)} + |\Pi| \cdot 2^{|\Pi|(|\Pi|+log\, k)} \leq k^{|\varphi|\cdot|\Pi|^2}$. □

The following is an immediate consequence of Lemma 4.2 and Proposition 4.5

Corollary 4.6: It is possible to determine if $G \sim^{\Pi}_{\varphi} H$ for k-graphs G, H by a deterministic computation using additional space $O\left(k^{|\varphi|\cdot|\Pi|^2}\right)$.

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 [28] 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 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 $G \sim^{\Pi}_{\varphi} H^a$. We write $Types(\Sigma, \mathbf{R})$ for a complete set of adorned representatives of $L(\Sigma, \mathbf{R})$.

Example 5.1: Consider the program of Example 1.1. The expansions depicted in Figure 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 $\{x_1 x_2 : \mathbf{rgpath}(x_1, x_2)\}$ 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 $G = \langle g, u_1, \ldots, u_n, r\rangle$, 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

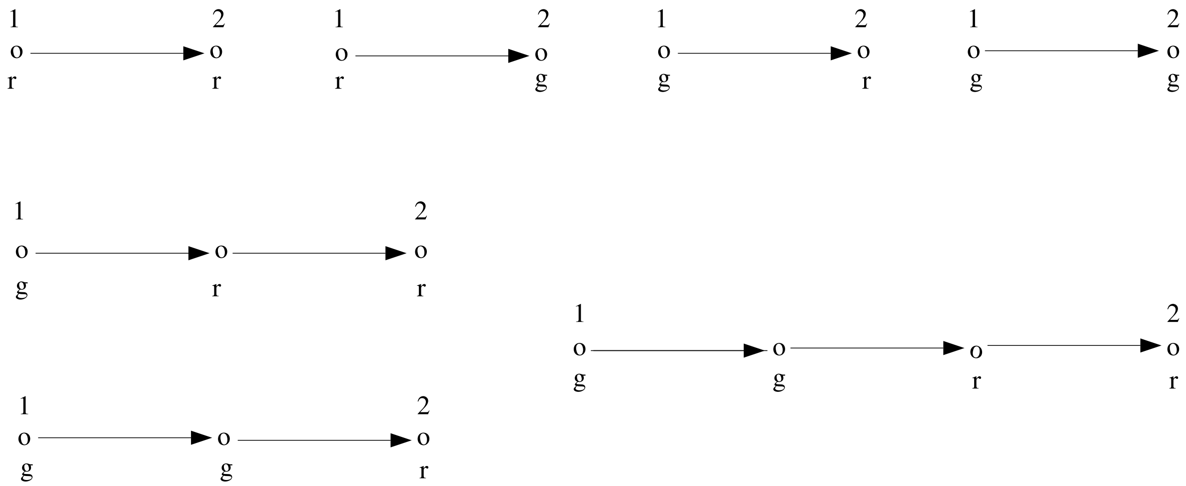

Figure 2: A complete set of expansions

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. $\Box$

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 5.1: $\Sigma, [B, D] \not\models (\Pi, \varphi)$ 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 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 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 we have $\theta(e) \sim_\varphi^\Pi \theta^a(e)$ for each defined atom e, it follows using Lemma 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 2.1. Lemma 3.2 now yields that $\Sigma, [B, D] \not\models (\Pi, \varphi)$. $\Box$

Lemma 5.1 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 $G_e \in Types(\Sigma, \mathbf{R}_e)$ 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 $Types(\Sigma, \mathbf{R})$ 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 $Types(\Sigma, \mathbf{R})$ 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 5.2: 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 4.5 there exists a finite set S of size

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

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 5.3: For every monadic query (Π, φ) and number k, the set $AS_{Arity(k)}(\Pi, \varphi)$ is in co-NP.

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 [2] 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 following recursive alternating procedure.

$ELT(G, R)$:

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.

The following lemma establishes correctness of this routine.

Lemma 5.4: The procedure call $ELT(G, \mathbf{R})$ accepts if and only if G is an adorned representative of $L(\Sigma, \mathbf{R})$.

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 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

$$H(n) = B(n)\theta' \tag{3}$$

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 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) implies that

$$H^a(n) = B^a(n)\theta^a \tag{4}$$

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 (4) by Lemma 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 $ELT(G(n), \mathbf{R}(n))$. $\square$

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 4.5 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 4.6, 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 [2], the class

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

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 [24]. This establishes the following upper bounds for data complexity when programs have unbounded arity:

Theorem 5.5: 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 4.5 and Corollary 4.6, 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 5.6: 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 5.7: 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 3.2 and Lemma 3.3 the condition that substitutions not identify vertices which the unique names condition requires are distinct. This results in the following equivalent of Lemma 5.1. 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 5.8: $\Sigma, [B, D] \not\models^{un} (\Pi, \varphi)$ 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.

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 [12]. 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 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. $\square$

It follows from Theorem 6.1 and Theorem 5.3 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.

$$\begin{aligned} &\mathbf{bit}(x) : -zero(x) \\ &\mathbf{bit}(x) : -one(x) \\ &\mathbf{quant}(x) : -all(x) \\ &\mathbf{quant}(x) : -some(x) \\ &\mathbf{node}(x) : -leaf(x), \mathbf{bit}(x), \mathbf{quant}(x), \mathbf{id}(x) \\ &\mathbf{node}(x) : -\mathbf{bit}(x), \mathbf{quant}(x), \mathbf{id}(x), \\ &\qquad\qquad left(x, y_1), right(x, y_2), \mathbf{node}(y_1), \mathbf{node}(y_2). \end{aligned} \tag{5}$$

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 over-generation 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

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

where

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

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 (5), plus the rules

$$\begin{aligned}
&\mathbf{id}(x) : -\mathbf{cell}(x, c_1), \ldots, \mathbf{cell}(x, c_m) \\
&\mathbf{cell}(x, y) : -contains(x, y, a_1) \\
&\qquad\vdots \\
&\mathbf{cell}(x, y) : -contains(x, y, a_k).
\end{aligned} \tag{6}$$

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

$$\begin{aligned}&\exists xyz_1t_1z_2t_2z_3t_3t_2'\{\\ &\quad left(x,y)\wedge next(z_1,z_2)\wedge next(z_2,z_3)\wedge\\ &\quad\quad \bigwedge_{i=1}^{3} contains(x,z_i,t_i)\wedge\\ &\quad contains(y,z_2,t_2')\wedge lerror(t_1,t_2,t_3,t_2')\}.\end{aligned}$$

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 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 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, $|\Sigma'| \leq |\Sigma|^c . |\Sigma_1| = |\Sigma|^{cN}$. $\square$

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 (5) we use

$$\begin{array}{l} \mathbf{node}(x) : -leaf(x), \mathbf{id}(x) \\ \mathbf{node}(x) : -\mathbf{id}(x), nextnode(x,y), \mathbf{node}(y). \end{array} \tag{7}$$

to generate a sequence of Turing machine configurations representing the computation. We reuse the rules (6) to generate the configurations. Note that this program is not linear. However, it is weakly linear, the ranking of defined predicates being **cell**, **id**, **node**. Thus, we may use Lemma 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 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 (5) 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.

$$
\begin{aligned}
&\mathbf{id}(x) : -firstcell(x,y), \mathbf{cellseq}(x,y)\\
&\mathbf{cellseq}(x,y) : -lastcell(x,y), \mathbf{cell}(x,y)\\
&\mathbf{cellseq}(x,y) : -\mathbf{cell}(x,y), nextcell(y,z), \mathbf{cellseq}(x,z)\\
&\mathbf{cell}(x,y) : -cellof(x,y), state(y,s), firstbit(s,u), \mathbf{bitseq}(s,u),\\
&\qquad\qquad cellnum(y,t), firstbit(t,v), \mathbf{bitseq}(s,v)\\
&\mathbf{bitseq}(x,y) : -lastbit(x,y), \mathbf{bit}(y)\\
&\mathbf{bitseq}(x,y) : -\mathbf{bit}(y), nextbit(y,z), \mathbf{bitseq}(x,z).
\end{aligned}
\tag{8}
$$

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 (5) and (8). 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

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

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

$$
\begin{aligned}
\exists xyt\{ & firstcell(x,y) \wedge cellnum(y,t) \wedge\\
& \exists t_1 \dots t_m[firstbit(t,t_1) \wedge nextbit(t_1,t_2) \wedge \dots\\
& \qquad \dots \wedge nextbit(t_{m-1},t_m) \wedge \bigvee_{1\le i\le m} one(t_i)]\}
\end{aligned}
$$

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

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

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

$$
\begin{array}{rl}
a(x) = & \exists y_1 y_2 y_3[state(x,u) \wedge firstbit(u,y_1) \wedge nextbit(y_1,y_2) \wedge \\
& \quad nextbit(y_2,y_3) \wedge zero(y_1) \wedge one(y_2) \wedge zero(y_3)]
\end{array}
$$

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.

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

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

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

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 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 (7) and (8), linearized using Lemma 6.3.

Theorem 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 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.

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 [7] 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 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 paper. (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

$$\begin{array}{l}\mathbf{R}(x_1, x_2) : -edge(x_1, x_2)\\ \mathbf{R}(x_1, x_2) : -\mathbf{R}(x_1, y), edge(y, x_2)\end{array}$$

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

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

We note that the query $Q_{\mathbf{R}}$ contains only basic predicates, and is preserved under homomorphism. Thus, it follows from Lemma 3.4 and Theorem 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 [31] 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. [9] 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.

8 Conclusion

Courcelle [6] has previously used hypergraph grammars to show the decidability of a problem related to a notion of optimization of Datalog programs due to Sagiv and Naughton [23]. We expect that these techniques will find other applications to Datalog.

The notion of adorned glue type is a modification of the notion of glue type due to Lenguaer and Wanke [16], who study the complexity of a number of specific queries in the languages generated by graph grammars. For monadic queries, their equivalence relation would correspond to the relation defined on hypergraphs 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. Cosmadakis et al. [4] have used ideas similar to adornment 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 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 7.1 yields the decidability of this class.

In view of the high complexities uncovered in this paper it is desirable to find either approximations or more tractable subproblems. One apparently promising direction is to consider conjunctive queries, i.e., queries not containing any disjunctions. We have already noted that all our simulations required disjunction. It is possible to classify conjunctive queries in databases with OR-objects into two syntactic classes [13]. All the queries in one class have polynomial time data complexity. The queries in the other class have co-NP complete data complexity. It would be interesting to have a similar result for a more general class of databases.

In further work [26], we investigate the effect on the complexity of basic queries of a number of generalizations of the class of permitted definitions in recursively indefinite databases. Specifically, we consider Datalog rules using negation-as-failure, as well as rules and queries containing inequality ($\neq$) and the order relation $<$, interpreted over linearly ordered domains. These generalizations result in increases in query complexity.

Acknowledgements: This work has benefitted from the comments of Eric Allender, Alex Borgida, Tomasz Imielinski, Thorne McCarty, Alberto Mendelzon and two anonymous referees. Thanks also to Bruno Courcelle for his explanation of the complexity of his decision procedure. This work was begun while the author was supported by a Rutgers University Graduate School Excellence Fellowship.

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] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[3] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[4] 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.

[5] B. Courcelle. An axiomatic definition of context-free rewriting and its application to NLC graph grammars. *Theoretical Computer Science*, 55:141–181, 1987.

[6] 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.

[7] B. Courcelle. The monadic second order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] T. Imielinski. Incomplete deductive databases. *Annals of AI and Mathematics*, 3(II-IV), 1991.

[13] T. Imielinski and K. Vadaparty. Complexity of query processing in databases with or-objects. In *Proceedings of the Eighth annual SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 51–65, 1989.

[14] 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.

[15] P. K. Kolaitis and C. H. Papadimitriou. Some computational aspects of circumscription. *Journal of the ACM*, 37(1):1–14, January 1990.

[16] 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.

[17] V. Lifschitz. Computing circumscription. In *Proceedings of the International Joint Conference in Artificial Intelligence*, pages 121–127, 1985.

[18] V. Lifschitz. Pointwise circumscription. In M.L. Ginsburg, editor, *Readings in Non-Monotonic Reasoning*, pages 179–193. Morgan Kaufman, 1988.

[19] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[20] J. McCarthy. Applications of circumscription to non-monotonic reasoning. *Artificial Intelligence*, 28:89–116, 1986.

[21] 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.

[22] 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.

[23] 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.

[24] W. Savitch. Relationships between non-deterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[25] O. Shmueli. Decidabilty and expressiveness aspects of logic queries. In *Proceedings of the Sixth annual SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 237–249, 1987.

[26] R. van der Meyden. *The Complexity of Querying Indefinite Information: Defined Relations, Recursion and Linear Order.* PhD thesis, Rutgers University, 1992.

[27] 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.

[28] M. Vardi. Querying logical databases. *Journal of Computer and System Sciences*, 33:142–160, 1986.

[29] M. Vardi. Decidability and undecidability results for boundedness of linear recursive queries. In *Proceedings of the Seventh annual SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 341–350, 1988.

[30] M. Vardi. Automata theory for database theoreticians. In *Proceedings of the Eighth annual SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 83–92, 1989.

[31] 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.