Reprinted from JOURNAL OF COMPUTER AND SYSTEM SCIENCES

Vol. 52, No. 3, June 1996
Printed in Belgium

Completeness Results for Recursive Data Bases

TIRZA HIRST AND DAVID HAREL*

Department of Applied Mathematics & Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel

Received November 20, 1995

We consider infinite recursive (i.e., computable) relational data bases. Since the set of computable queries on such data bases is not closed under even simple relational operations, one must either make do with a very modest class of queries or considerably restrict the class of allowed data bases. We define two query languages, one for each of these possibilities, and prove their completeness. The first is the language of quantifier-free first-order logic, which is shown to be complete for the non-restricted case. The second is an appropriately modified version of Chandra and Harel's language QL, which is proved complete for the case of "highly symmetric" data bases, i.e., ones whose set of automorphisms is of finite index for each tuple width. We also address the related notion of BP-completeness.

1. INTRODUCTION

Computer scientists are interested predominantly in finite objects. In so far as they are interested in *infinite* objects, these must be countable and computable, i.e., recursive thus admitting an effective finite representation. Recursive graphs have been studied quite extensively in the past. They can be viewed simply as recursive binary relations over the natural numbers (here $\mathcal{N}$ serves, without loss of generality, as the set of nodes). Many properties of recursive graphs have been investigated and have been shown to be undecidable; see, e.g., [Be1, Be2, BG]. We are also gaining some understanding as to their precise level of undecidability; see [BG, H, HH1].

Viewing graphs as binary relations immediately suggests a generalization. We may define a recursive model, or a recursive relational data base, simply as a finite tuple of recursive relations (not necessarily binary) over some countable domain. We thus obtain a natural generalization of the notion of a finite relational data base.[1] Recursive models have been the subject of much work in classical model theory (see, e.g., the survey [NR]), but none, it seems, from the perspective of data bases. The idea of considering infinite, recursive data bases is not without justification: Values for the trigonometric functions, for example, can be viewed as a recursive data base, since we might be interested in the sines or cosines of infinitely many angles. Instead of keeping them all in a table, which is impossible, we keep rules for computing the values from the angles, and vice versa. This really just means that we have an effective way of telling whether an edge is present between nodes i and j in an infinite graph—which is precisely the notion of a recursive graph.

Recursive data bases appear to constitute a fertile area for research, raising theoretical and practical questions concerning the computability and complexity of queries and update operations, and the power and flexibility of appropriate query languages. In this paper, we propose to initiate such research by addressing the problem of capturing the class of computable queries over recursive data bases, the motivation being borrowed from [CH].

It is easy to see that recursive relations are not closed under some of the simplest accepted relational operators. For example, if we define the primitive recursive relation R, such that $R(x, y, z)$ holds for a 3-tuple of natural numbers iff the y th Turing machine halts on input z after x steps, then $R\downarrow$, the projection of R on the second and third columns, is the nonrecursive halting predicate. Thus, even very simple queries do not preserve computability when applied to general recursive relations. This difficulty can be overcome in essentially two ways. The first is to accept the situation as is; that is, to resign ourselves to the fact that on recursive data bases the class of computable queries will necessarily be very modest, and then to try to capture that class in a (correspondingly modest) complete query language. The second is to restrict the data bases, so that the standard kind of queries *will* preserve computability, and then to try to establish a reasonable completeness result for these restricted inputs. The first case will give rise to a rich class of data bases but a poor class of queries, and in the second these will be reversed. In both cases, of course, in addition to being Turing computable, the queries will also have to

* Part of this author's work was carried out during a sabbatical at the Dept. of Computer Science, Cornell University, Ithaca, NY, and was partially supported by Grants AF F49620-94-1-0198 (to F. Schneider), NFS CCR-9223183 (to B. Bloom), NSF CDA-9024600 (to K. Birman), and ARO DAAL03-91-C-0027 (to A. Nerode). E-mail addresses: {tirza, harel} @ wisdom.weizmann.ac.IL.

[1] Clearly this idea can also be applied to more elaborate versions of the relational model, including, e.g., attributes and multiple domains, as well to non-relational models of data.

0022-0000/96 $18.00

satisfy the consistency criterion of [CH], more recently termed *genericity* [HY], whereby queries must preserve isomorphisms.

In the interest of pursuing both of these approaches, we prove two results. In Section 2 we consider recursive data bases and computable queries, and prove that the language of quantifier-free first-order relational calculus is complete. This shows that the class of computable queries on general recursive data bases is indeed extremely restricted. At the heart of the proof is a notion we call *local isomorphism*, which is defined over pairs (B, u) where B is a data base and u is a tuple over B's domain. We show that for each tuple-width, the equivalence relation corresponding to this isomorphism is of finite index, and that the results of applying a computable query constitute a union of some of the equivalence classes.

In Section 3 we consider *highly symmetric* data bases, which have very rich sets of automorphisms. Technically, the restriction prescribes that for each tuple-width the equivalence relation that identifies tuples that are interchangeable by some automorphism of B is recursive and of finite index. We also require that a (finitely branching) tree of representatives of all equivalence classes for all tuple widths is recursive. For such data bases, represented by such a tree, we prove that a slightly modified version of the query language QL of [CH] is complete. The proof follows the general lines of the proof in [CH] but is rather more complicated.

The two results lie on the extremities of the spectrum of possibilities for interesting classes of computable queries over interesting classes of recursive data bases. The results could lead to work on other, less extreme possibilities. Moreover, it now appears to be feasible to investigate the complexity of queries on recursive data bases and their implementation.

Sections 4 and 5 discuss variations of the completeness result for highly symmetric data bases. In the former we consider the special case of finite–co-finite relations, proving completeness for corresponding variants of QL, and in the latter we prove completeness for an appropriately modified version of the generic machine language GM of [AV].

Finally, Section 6 addresses the notion of BP-completeness, based on the work of Bancilhon [B] and Paredaens [P]. Under this notion, which involves expressing relations, not queries (see [CH]), we show that no effective complete language for the full class of recursive data bases is possible. On the other hand, for highly symmetric data bases, first-order logic is complete, as in the finite case.

It is of interest to note that highly symmetric data bases give rise to the same complete languages as finite data bases, for both query completeness and BP-completeness. This is due to the fact that they can be effectively represented by finite relations that consist of representatives of the aforementioned equivalence classes.

2. COMPLETENESS FOR RECURSIVE DATA BASES

A *recursive relation* is a recursive set of tuples over a recursive countably infinite domain. For example, the following relation is recursive:

$$\{(x, y, z) \mid x, y, z \in \mathcal{N}, z = x * y\}.$$

A recursive relation R can be represented by a Turing machine, which on input u decides whether the tuple u is in R. We denote by $|u|$ the rank of a tuple u.

DEFINITION 2.1. Let D be a countably infinite recursive set, and let $R_1, ..., R_k$, $k > 0$, be relations, such that for all $1 \leqslant i \leqslant k$, $R_i \subseteq D^{a_i}$. We say that $B = (D, R_1, ..., R_k)$ is a *recursive relational data base* of type $a = (a_1, ..., a_k)$ (an *r-db* for short) if each R_i is a recursive relation. We often use $D(B)$ to denote D, the domain of B.

We actually think of an r-db as a sequence of Turing machines that accept the appropriate relations.

DEFINITION 2.2. Let $B_1 = (D_1, R_1, ..., R_k)$ and $B_2 = (D_2, R'_1, ..., R'_k)$ be two r-db's of type $a = (a_1, ..., a_k)$ and let $u \in D_1^n$ and $v \in D_2^n$. Then

1. B_1 and B_2 are *isomorphic* by isomorphism h, if $h: D_1 \to D_2$ is a bijection, and $h(R_i) = R'_i$ for each $1 \leqslant i \leqslant k$.

2. (B_1, u) and (B_2, v) are *isomorphic*, written $(B_1, u) \cong (B_2, v)$, if B_1 and B_2 are isomorphic by an isomorphism taking u to v.

3. (B_1, u) and (B_2, v) are *locally isomorphic*, written $(B_1, u) \cong_l (B_2, v)$, if the restriction of B_1 to the elements of u and the restriction of B_2 to the elements of v are isomorphic by an isomorphism taking u to v.

Isomorphism implies local isomorphism, but not vice versa. For example, let $R_1 = \{(a, a), (a, b)\}$ and $R_2 = \{(c, c)\}$. Here, $(R_1, (a)) \cong_l (R_2, (c))$, but $(R_1(a)) \ncong (R_2, (c))$.

An important difference between local isomorphism and isomorphism is that the former is decidable for r-db's, whereas the second is not. In fact, we have:

PROPOSITION 2.1 [M]. *The isomorphism problem for r-db's is Σ_1^1-complete.*

Note that deciding $(B_1, u) \cong (B_2, v)$ is also Σ_1^1-complete, since B_1 and B_2 are isomorphic iff $(B_1, (\,)) \cong (B_2, (\,))$, where $(\,)$ is the tuple of rank 0.

PROPOSITION 2.2. *The relation $\cong_l$ is recursive.*

Proof. Given r-db's $B_1 = (D_1, R_1, ..., R_k)$ and $B_2 = (D_2, R'_1, ..., R'_k)$ and tuples $u = (u_1, ..., u_n)$ and $v = (v_1, ..., v_m)$. In order to determine whether $(B_1, u) \cong_l (B_2, v)$, we have to verify the following, all of which are readily seen to be computable: (i) $|u| = |v|$; (ii) for every i and j,

$1 \leqslant i, j \leqslant n$, $u_i = u_j$ iff $v_i = v_j$; and (iii) for every i, $1 \leqslant i \leqslant k$, and for each choice of $j_1, ..., j_{a_i}$ between 1 and n, we have $(u_{j_1}, ..., u_{j_{a_i}}) \in R_i$ iff $(v_{j_1}, ..., v_{j_{a_i}}) \in R'_i$. ▮

It is easy to see that if we fix the type a, then for tuples of rank n, $\cong_l$ is an equivalence relation of finite index; its equivalence classes are pairs of the form (B, u), where B is an r-db of type a and u is a tuple of rank n. We denote these equivalence classes by $C^n = \{C^n_1, ..., C^n_m\}$.

EXAMPLE. For type $a = (2, 1)$, there are $2^2 + 2^4 \cdot 2^2 = 68$ equivalence classes of $\cong_l$ of rank 2. One of them is:

$$C^2_i = \{(B, (x, y)) \mid x \neq y \wedge (x, y) \notin R_1 \wedge (y, x) \in R_1 \wedge (x, x) \in R_1 \wedge (y, y) \notin R_1 \wedge x \notin R_2 \wedge y \in R_2\}.$$

DEFINITION 2.3. An *r-db query of type a* (or an *r-query* for short) is a partial function Q, which, for each r-db B of type a, yields an output (if any) that is a recursive relation over $D(B)$. We let $\hat{Q}$ denote the set $\{(B, u) \mid u \in Q(B)\}$.

We now want to define recursive queries over r-db's. There are various possibilities for this. One is to require the existence of a Turing machine, which, on an input containing (codes for) the Turing machines of the relations in the input data base, produces as output the (code of the) Turing machine for the output relation. We prefer the following oracle-based definition.

DEFINITION 2.4. An r-query Q is *recursive* if there is an oracle Turing machine which, given a tuple u, uses oracles for the relations of the input data base B to decide whether $u \in Q(B)$. If $Q(B)$ is undefined, the machine does not halt on any input tuple.

We denote a Turing machine M that uses oracles for the relations of B by M^B.

The two approaches differ in that, if the Turing machine computing Q can relate to the actual codes for the input Turing machines, it can violate genericity (see below). If the Turing machine is allowed to access the input machines only in order to ask questions of the form "is $u \in R$", then the two definitions can be shown to be equivalent.[2]

DEFINITION 2.5. An *r-query* Q is called *generic* if it preserves isomorphisms; i.e., for all B_1, B_2, u, and v, if $(B_1, u) \cong (B_2, v)$, then $Q(B_1)$ is defined iff $Q(B_2)$ is defined, and $u \in Q(B_1)$ iff $v \in Q(B_2)$. It is called *locally generic* if it preserves local isomorphisms; i.e., for all B_1, B_2, u, and v, if $(B_1, u) \cong_l (B_2, v)$, then $Q(B_1)$ is defined iff $Q(B_2)$ is defined, and $u \in Q(B_1)$ iff $v \in Q(B_2)$.

[2] The first approach has the additional problem of multiple codes for the output relation, and we would like the Turing machine computing an r-query to return a unique code, say, the smallest one. However, deciding equality of codes for Turing machines is non-computable. Hence, we prefer the second definition.

Unions, intersections, and complementations, for instance, are both generic and locally generic. The query that gives "the first tuple in R" or "all tuples containing the constant a" are neither generic nor locally generic. Indeed, every locally generic query is also generic, but the converse is not true. For example, the query Q defined by:

$$\{x \mid \exists y\, (x \neq y \wedge (x, y) \in R)\}$$

is clearly generic but is not locally generic, since if we let $R_1 = \{(a, a), (a, b)\}$ and $R_2 = \{(c, c)\}$, then $Q(R_1) = \{(a)\}$ and $Q(R_2) = \varnothing$ but $(R_1, (a)) \cong_l (R_2, (c))$.

PROPOSITION 2.3. *Any locally generic r-query Q satisfies the following*:

1. *either Q is defined for all B or Q is undefined for all B;*
2. *if $(B_1, u) \cong_l (B_2, v)$, then $(B_1, u) \in \hat{Q}$ iff $(B_2, v) \in \hat{Q}$;*
3. *if $(B_1, u) \in \hat{Q}$ and $(B_2, v) \in \hat{Q}$, then $|u| = |v|$.*

Proof. Parts 1 and 2 follow immediately from the definition. (For Part 1 note that for all B_1 and B_2, $(B_1, ( )) \cong_l (B_2, ( ))$.)

To prove Part 3, let $B_1 = (D_1, R_1, ..., R_k)$ and $B_2 = (D_2, R'_1, ..., R'_k)$ be two r-db's of type $a = (a_1, ..., a_k)$, and let $u = (u_1, ..., u_n)$ and $v = (v_1, ..., v_m)$ be tuples with $u \in Q(B_1)$ and $v \in Q(B_2)$. Assume, without loss of generality, that D_1 and D_2 are disjoint. We construct a new r-db, $B_3 = (D_3, S_1, ..., S_k)$, whose domain contains $u_1, ..., u_n$, $v_1, ..., v_m$ and additional new elements to make it infinite. For each i, S_i is defined as follows: $z \in S_i$ iff z is a tuple over $\{u_1, ..., u_n\}$ and $z \in R_i$, or z is a tuple over $\{v_1, ..., v_m\}$ and $z \in R'_i$. Now, $(B_1, u) \cong_l (B_3, u)$ and $(B_2, v) \cong_l (B_3, v)$. Since Q is a locally generic, and $u \in Q(B_1)$ and $v \in Q(B_2)$, we have $u \in Q(B_3)$ and $v \in Q(B_3)$. Since $Q(B_3)$ is a relation, u and v must have the same rank. ▮

Proposition 2.3 states that a locally generic r-query Q is either nowhere defined, or is everywhere-defined but does not split any equivalence class of $\cong_l$. Actually, $\hat{Q}$ is the union of some equivalence classes of $\cong_l$ of a common rank n. By the notation introduced above, if Q is a locally generic r-query, $\hat{Q} = \bigcup_{j=1}^{t} C^n_{i_j}$ for some $C^n_{i_1}, ..., C^n_{i_t} \in C^n$. Conversely, it is easy to see that each subset of C^n defines a locally generic r-query Q, such that $\hat{Q}$ is the union of its elements. Hence, we have the following characterization:

PROPOSITION 2.4. *Q is a locally generic r-query iff $\hat{Q} = \bigcup_{j=1}^{t} C^n_{i_j}$ for some $C^n_{i_1}, ..., C^n_{i_t} \in C^n$.*

So much for locally generic r-queries. Now, while generic r-queries are not the same in general, they are the same when the queries are restricted to being recursive:

PROPOSITION 2.5. *If Q is a recursive r-query, then Q is generic iff Q is locally generic.*

Proof. Let Q be a recursive r-query of type a.

Let $B_1=(D_1, R_1, ..., R_k)$ and $B_2=(D_2, R'_1, ..., R'_k)$ be two r-db's of type $a=(a_1, ..., a_k)$ and let u and v be n-tuples. If $(B_1, u)\cong(B_2, v)$, then $(B_1, u)\cong_l(B_2, v)$. Thus the "if" direction is clear. For the "only-if" direction, assume that Q is generic but not locally generic. Therefore, there are $B_1=(D_1, R_1, ..., R_k)$, $B_2=(D_2, R'_1, ..., R'_k)$, $u=(u_1, ..., u_n)$, and $v=(v_1, ..., v_n)$, such that $(B_1, u)\cong_l(B_2, v)$ but either $Q(B_1)$ is defined and $Q(B_2)$ is undefined or both are defined but $u\in Q(B_1)$ and $v\notin Q(B_2)$. Let M be an oracle Turing machine that computes Q, and let σ_1 and σ_2 be the computation paths of $M^{B_1}(u)$ and $M^{B_2}(v)$, respectively. Note that if $Q(B_2)$ is undefined σ_2 is infinite. Let $d_1, ..., d_m$ be all the elements from D_1 that do not appear in u but which, during the computation σ_1, appear in tuples participating in questions of M to the oracles for B_1. Similarly, let $E=\{e_1, ..., e_r, ...\}$ be the set of all the elements from D_2 that do not appear in v but which, during the computation σ_2, appear in tuples participating in questions of M to the oracles for B_2. E might be infinite if σ_2 is infinite.

We now construct new data bases $B_3=(D_3, S_1, ..., S_k)$ and $B_4=(D_4, S'_1, ..., S'_k)$, as follows. D_3 contains $u_1, ..., u_n$, $d_1, ..., d_m$, and new elements $e'_1, ..., e'_r, ...$ corresponding to $e_1, ..., e_r, ...$. If E is finite, we add elements in order to make D_3 infinite. The relations of B_3 are defined such that the answers to questions of M to the oracles of B_1 and to the oracles of B_3 will be identical. For each i, $1\leqslant i\leqslant k$, define S_i as follows: $x\in S_i$ iff x is over $\{u_1, ..., u_n, d_1, ..., d_m\}$ and $x\in R_i$, or x is over $\{u_1, ..., u_n, e'_1, ..., e'_r, ...\}$ and $x'\in R'_i$, where x' is x, except that each u_i is replaced by v_i, and each e'_i is replaced by e_i. B_4 is defined similarly, using $d'_1, ..., d'_m$ corresponding to $d_1, ..., d_m$.

Now, the following permutation:

$$\begin{pmatrix} u_1u_2\cdots u_nd_1d_2\cdots d_me'_1e'_2\cdots e'_r\cdots \\ v_1v_2\cdots v_nd'_1d'_2\cdots d'_me_1e_2\cdots e_r\cdots \end{pmatrix}$$

is an isomorphism between B_3 and B_4, taking u to v. Hence, $(B_3, u)\cong(B_4, v)$. On the other hand, the computation paths of $M^{B_3}(u)$ and $M^{B_4}(v)$ are identical to those of $M^{B_1}(u)$ and $M^{B_2}(v)$, respectively. This means that in the first case, when $Q(B_1)$ is defined but $Q(B_2)$ is undefined, $Q(B_3)$ will be defined but $Q(B_4)$ will be undefined, and in the second case, when $u\in Q(B_1)$ and $v\notin Q(B_2)$, then also $u\in Q(B_3)$ and $v\notin Q(B_4)$. This contradicts the genericity of Q. ∎

DEFINITION 2.6. An r-query is called *computable* if it is recursive and generic. A query language is *r-complete* if it expresses precisely the computable r-queries.

Denote by $\mathscr{L}^-$ the language of first-order logic without quantifiers, considered as a query language with queries of the form:

$$\{(x_1, ..., x_n)\mid \phi(x_1, ..., x_n, R_1, ..., R_k)\}.$$

Here, ϕ is a quantifier free formula, $R_1, ..., R_k$ denote the relations of the input r-db, and $x_1, ..., x_n$ are ϕ's free variables. The allowed atomic formulas are $x_i=x_j$, for $1\leqslant i, j\leqslant n$, and $(x_{j_1}, ..., x_{j_{a_i}})\in R_i$, for $1\leqslant j_1, ..., j_{a_i}\leqslant n$ and $1\leqslant i\leqslant k$, where a_i is the rank of R_i. In particular, if R is of rank 0, then $(\ )\in R$ is a legal atomic formula. In addition, $\mathscr{L}^-$ contains a special expression, *undefined*, in order to express everywhere-undefined queries.

THEOREM 2.1. *$\mathscr{L}^-$ is r-complete.*

Proof. Clearly, every query expressible in $\mathscr{L}^-$ is recursive and generic when applied to r-db's. For the other direction, let Q be a computable r-query. By Proposition 2.5, Q is locally generic. If Q is undefined for all B, then it is expressed by the special expression *undefined*. Otherwise, Q is always defined, and by Proposition 2.4, for the appropriate n, $\hat{Q}=\bigcup_{j=1}^{l} C^n_{i_j}$, for some choice of i_j's. In addition, every equivalence class C^n_i is expressible by some ϕ_i, in such a way that $(B, u)\models\phi_i$ iff $(B, u)\in C^n_i$. The formula ϕ_i describes the containment or non-containment of the projections on u in the relations of B. Thus, $\phi_{i_1}\vee\cdots\vee\phi_{i_l}$ is a formula in $\mathscr{L}^-$ that expresses Q. ∎

To illustrate this, the ϕ_i corresponding to C^2_i (see the example given earlier) is:

$$x\neq y\wedge(x, y)\notin R_1\wedge(y, x)\in R_1\wedge(x, x)\in R_1$$
$$\wedge(y, y)\notin R_1\wedge x\notin R_2\wedge y\in R_2.$$

It is not to hard to show that $\mathscr{L}^-$ (or an appropriate generalization) is complete for certain special cases of r-db's. Here are some examples.

PROPOSITION 2.6. *$\mathscr{L}^-$ is complete for unary r-db's.*

One way of viewing this, without loss of generality, is that quantifier-free first order predicate logic with equality, when applied to recursive sets over $\mathscr{N}$ (i.e., apart from equality the predicates are unary) expresses precisely the generic computable functions that yield recursive relations over $\mathscr{N}$.

Denote by $\mathscr{L}^-_n$ the language $\mathscr{L}^-$ applied to data bases with domain $\mathscr{N}$ and restricted to have results only in $\{1, ..., n\}$. Specifically, we may assume that it allows only expressions of the form: $\{(x_1, ..., x_m)\mid \phi(x_1, ..., x_m, B)\wedge x_1, ..., x_m\in\{1, ..., n\}\}$, where ϕ is quantifier free.

$\mathscr{L}^-_n$ yields non-generic queries. For example, assume that $Q(B)$ is non-empty. Let B' be isomorphic to B, such that $1, ..., n$ are replaced, respectively, with $n+1, ..., 2n$. Then $Q(B')=\varnothing$, and thus $Q(B')$ is not isomorphic to $Q(B)$. Here we require genericity only for tuples over $\{1, ..., n\}$; i.e., if B_1 and B_2 are isomorphic, then for every $u, v\in\{1, ..., n\}^m$, $u\in Q(B_1)$ iff $v\in Q(B_2)$.

PROPOSITION 2.7. *For any n, $\mathscr{L}^-_n$ expresses precisely the recursive functions that yield relations over $\{1, ..., n\}$ whose isomorphisms are preserved only for tuples over $\{1, ..., n\}$.*

Proof. $\mathscr{L}_n^-$ is clearly recursive and preserves isomorphisms for tuples over $\{1, ..., n\}$. Let Q be a recursive query yielding, for each r-db over $\mathscr{N}$ of type $a = (a_1, ..., a_k)$ and a given integer n, a relation over $\{1, ..., n\}$ whose isomorphisms are preserved for tuples over $\{1, ..., n\}$. Q is locally generic for tuples over $\{1, ..., n\}$. Otherwise, as in the proof of Proposition 2.5, we could construct isomorphic r-db's B_3 and B_4 with tuples u and v over $\{1, ..., n\}$, such that $(B_3, u) \cong (B_4, v)$ but $\hat{Q}$ contains only (B_3, u). In addition, as in the general case, Q yields for all r-db's relations of some common rank. Since there are finitely many equivalence classes of $\cong_l$ for each rank that contain only tuples over $\{1, ..., n\}$, we can express Q in $\mathscr{L}^-$ restricted to $\{1, ..., n\}$. ∎

3. COMPLETENESS IN THE HIGHLY SYMMETRIC CASE

3.1. *Highly Symmetric Data Bases*

DEFINITION 3.1. Let B be a fixed r-db. For each $u, v \in D(B)^n$, $n > 0$, we say that u and v are *B-equivalent* (or just *equivalent* if B is understood), written $u \cong_B v$, if $(B, u) \cong (B, v)$.

In words, u and v are B-equivalent if there is an automorphism of B taking u to v.

DEFINITION 3.2. B is *highly symmetric* if, for each $n > 0$, the relation $\cong_B$ induces only a finite number of equivalence classes of rank n.

Let $B = (D, R_1, ..., R_k)$. Term an r-db of the form $(D, R_1, ..., R_k, \{(d_1)\}, ..., \{(d_m)\})$, for some $m \geq 0$ and $d_1, ..., d_m \in D$, a *stretching* of B by $d_1, ..., d_m$.

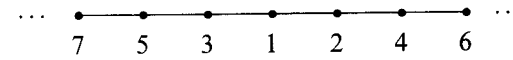
PROPOSITION 3.1. *B is highly symmetric iff for each stretching B' of B the relation $\cong_{B'}$ induces only a finite number of equivalence classes of rank* 1.

Proof. If a stretching B' of B by $d_1, ..., d_m$ contains infinitely many elements $a_1, a_2, ...$ that are pairwise non-equivalent in B', then the infinitely many tuples $\{(d_1, ..., d_m, a_1), (d_1, ..., d_m, a_2), ...\}$ are pairwise non-equivalent in B.

For the converse, assume that B is not highly symmetric. Let m be the maximal rank for which there is only a finite number of equivalence classes of $\cong_B$. Let U be a set of representatives of all of the equivalence classes of rank m, and let S be a set containing infinitely many tuples of rank $m+1$ that are pairwise non-equivalent. Since U is finite and S is infinite, there exists some $u = (d_1, ..., d_m) \in U$ and an infinite subset $S' \subset S$, such that each $v = (v_1, ..., v_m, v_{m+1}) \in S'$ satisfies $(v_1, ..., v_m) \cong_B (d_1, ..., d_m)$.

Now, let B' be the stretching of B by $d_1, ..., d_m$. If $\cong_{B'}$ contains only a finite number of equivalence classes of rank 1, there must be infinitely many elements from among the $(m+1)$th components of the tuples of S' (i.e., from among the v_{m+1}'s above) that are in the same equivalence class of $\cong_{B'}$. But this means that infinitely many tuples in S' are actually equivalent in B, which is a contradiction. ∎

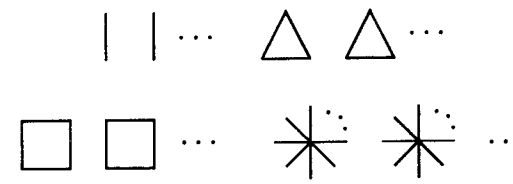
This characterization of highly symmetric r-db's yields a simple technique for showing that an r-db is not highly symmetric. It suffices to mark (or color) some elements of the domain, and then show that infinitely many elements are pairwise non-equivalent, where the coloring is taken to be part of the structure. For example, the following graph is not highly symmetric, since after coloring some node, there would be infinitely many nodes at different distances from the colored one:[3]



Without the coloring, the $\cong_B$ of this example induces a single equivalence class of rank 1 tuples, since all the nodes are equivalent. For $n = 2$, however, there are infinitely many non-equivalent pairs, since for each pair of distinct nodes i and j we have $(1, 2i) \not\cong_B (1, 2j)$.

For the case of rank 2 relations, i.e., graphs, high symmetricity is somewhat easier to elucidate. A highly symmetric graph consists of a finite or infinite number of connected components, where each component is, inductively, highly symmetric, and there are only finitely many pairwise non-isomorphic components. In a highly symmetric graph, the finite degrees, the distances between points, and the lengths of the induced paths are all bounded. A grid, for instance, is not highly symmetric, since it has an infinite path as an induced subgraph, and hence it would contain infinitely many pairs that are pairwise non-equivalent.[4] On the other hand, the full infinite clique is highly symmetric.

Here is an example of another highly symmetric graph:



A particularly interesting example of highly symmetric data bases are the (not necessarily recursive) countable random structures. These constitute a natural generalization of the Rado graph (cf. [Ra, Fa]). They are characterized by an infinite set of *extension axioms*, which say that for each finite set X of points in the domain, and for each possible way that a point not in X can be related to X in terms of atomic

[3] In the figure, a line between i and j represents the two edges (i, j) and (j, i).

[4] An induced subgraph of G is a subgraph obtained by restricting G to some of its nodes, omitting only edges connected to deleted nodes.

formulas, there is indeed such a point y. The axioms are indexed by the size of the set X. For example, the following is a 2-extension axiom for type $a=(2)$, i.e., for data bases consisting of a single binary relation symbol R:

$$\forall x_1 \forall x_2 (x_1 \neq x_2 \Rightarrow \exists y\,(y \neq x_1 \wedge y \neq x_2 \wedge (y, x_1) \in R \wedge (x_1, y) \notin R \wedge (y, x_2) \notin R \wedge (x_2, y) \in R)).$$

PROPOSITION 3.2. *For each type a, any countable random data base over a is highly symmetric.*

Proof. Let A be a countable random data base over a. We show that tuples are equivalent in A iff they are locally isomorphic. Since for each n, $\cong_l$ induces only a finite number of equivalence classes of rank n, the same is true for $\cong_A$, which means that A is highly symmetric.

Let $u=(u_1, ..., u_n)$ and $v=(v_1, ..., v_n)$ be tuples that are locally isomorphic in A. In order to show that $u \cong_A v$, we construct an automorphism on $D(A)$, taking u to v. For the purpose of the proof, let us assume that $D(A)$ is ordered (an order always exists, though not necessarily a recursive one).

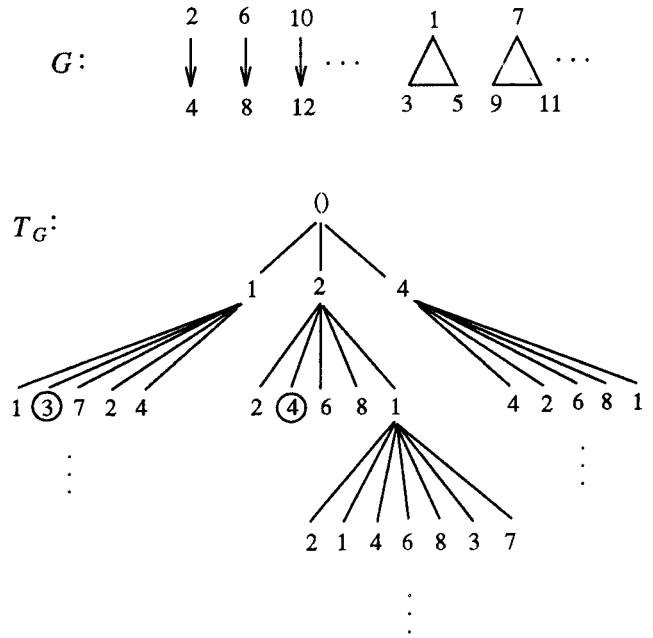
Let a_1 be the first element in $D(A)$ not appearing in u. By some n-extension axiom, there must be an element b_1 not appearing in v, such that $(A, ua_1) \cong_l (A, vb_1)$.[5] Now, let b_2 be the first element in $D(A)$ not appearing in vb_1. By some $(n+1)$-extension axiom, there is an element $a_2 \notin \{u_1, ..., u_n, a_1\}$ such that $(A, ua_1a_2) \cong_l (A, vb_1b_2)$. Continuing this back and forth argument yields the required automorphism. ∎

We now discuss the way we require highly symmetric data bases to be represented.

DEFINITION 3.3. A *characteristic tree* T_B for a relational data base $B=(D, R_1, ..., R_k)$ is any tree that satisfies the following. T_B's vertices are labeled with elements of D (with the exception of the root, which has no label), such that the tuple of labels along any path from the root to a node constitutes a representative of some equivalence class of $\cong_B$. A node in T_B is identified with the tuple leading to it. The tree covers representatives of all equivalence classes, and no two paths form representatives of the same class. T^n denotes the set of paths of length n from the root of T_B, and $T_B(x)$ denotes the set of labels of the immediate offspring of node x.

Clearly, B is highly symmetric iff T_B is finitely branching. As an example, the following figure describes a highly symmetric graph G, and a characteristic tree for it. Arrows denote directed edges and lines denote pairs of directed edges. The marked nodes, (1, 3) and (2, 4), are the representatives of the equivalence classes constituting G; that is, (i, j) is an edge in G iff it is equivalent to (1, 3) or (2, 4).

[5] We write ua_1 as shorthand for $(u_1, ..., u_n, a_1)$, and similarly for other tuple extensions.



3.2. *On Tuple Equivalence*

Before we define the kinds of recursive data bases we will be dealing with here, and then proceed to exhibit a complete language, we need to establish some preliminary results. In the following, let B be a fixed infinitely countable data base. We will often write $u \cong_l v$ instead of $(B, u) \cong_l (B, v)$, and D instead of $D(B)$.

PROPOSITION 3.3. *For every* $u, v \in D^n$,

$$u \cong_B v \quad \textit{iff} \quad \forall a_1 \exists b_1 \forall b_2 \exists a_2 \cdots (ua_1a_2 \cdots \cong_l vb_1b_2 \cdots).$$

(*See* [CK pp. 114–115].)

Proof. The "only-if" direction follows from the existence of an automorphism on D taking u to v. For the "if" direction, assume that

$$\forall a_1 \exists b_1 \forall b_2 \exists a_2 \cdots (ua_1a_2 \cdots \cong_l vb_1b_2 \cdots).$$

Let a_1 be the first element of D not appearing in u. By the assumption, there is some element b_1, such that $ua_1 \cong_l vb_1$. Now, let b_2 be the first element from D not appearing in vb_1. By the assumption, there is some a_2, such that $ua_1a_2 \cong_l vb_1b_2$. Continuing this back and forth construction for all the elements in D yields a full permutation taking u to v, thus implying $u \cong_B v$. ∎

DEFINITION 3.4. For $u, v \in D^n$, we write $u \equiv_0 v$ whenever $(B, u) \cong_l (B, v)$, and $u \equiv_{r+1} v$ whenever both $\forall a\, \exists b.ua \equiv_r vb$ and $\forall b\, \exists a.ua \equiv_r vb$.

Another way of stating the requirement for $u \equiv_r v$ in this definition is that, in the structure B, the tuples $u=(u_1, ..., u_n)$ and $v=(v_1, ..., v_n)$ must satisfy precisely the same

first-order formulas with up to r quantifiers and n free variables. An additional well known characterization is that $u \equiv_r v$ iff the duplicator has a winning strategy in the r-round first-order Ehrenfeucht–Fraisse Game (r-game, for short) played on (B, u) and (B, v) [E, Fr].

It is easy to see that it suffices to have the quantifiers range over nodes in T_B, since T_B contains representatives of all the equivalence classes:

PROPOSITION 3.4. *Let $u, v \in D^n$ and $u', v' \in T^n$, such that $u \cong_B u'$ and $v \cong_B v'$. Then, for all r,*

$$u \equiv_{r+1} v \quad \text{iff} \quad \forall a \in T(u')\ \exists b \in T(v')\ u'a \equiv_r v'b$$
$$\text{and} \quad \forall b \in T(v')\ \exists a \in T(u')\ u'a \equiv_r v'b.$$

In particular, if $u, v \in T^n$ then

$$u \equiv_{r+1} v \quad \text{iff} \quad \forall a \in T(u)\ \exists b \in T(v)\ ua \equiv_r vb$$
$$\text{and} \quad \forall b \in T(v)\ \exists a \in T(u)\ ua \equiv_r vb.$$

We now take advantage of the fact that highly symmetric data bases have finitely branching characteristic trees:

PROPOSITION 3.5. *If B is highly symmetric, then for every $u, v \in D^n$,*

$$u \cong_B v \quad \text{iff} \quad \forall r\ u \equiv_r v.$$

Proof. Let $u, v \in D^n$. The "only-if" direction is clear. For the "if" direction, assume that $\forall r\ u \equiv_r v$. Let $u', v' \in T^n$ such that $u \cong_B u'$ and $v \cong_B v'$. Hence, in order to establish $u \cong_B v$, it suffices to prove $u' \cong_B v'$.

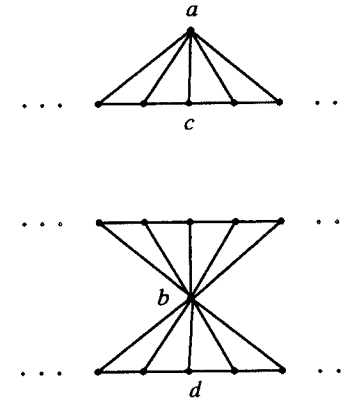
According to Proposition 3.4, $\forall r\ u \equiv_r v$ implies

$$\forall u\ \forall a_1\ \exists b_1\ \forall b_2\ \exists a_2 \cdots \forall c\ \exists d(u'a_1a_2 \cdots a_r \cong_l v'b_1b_2 \cdots b_r),$$

where, if r is odd, $c = a_r$ and $d = b_r$, and otherwise $c = b_r$ and $d = a_r$, and where, for all $1 \leqslant i \leqslant r$, $a_i \in T(u'a_1a_2 \cdots a_{i-1})$ and $b_i \in T(v'b_1b_2 \cdots b_{i-1})$.

In order to show $u' \cong_B v'$, we construct a permutation on D, which is an automorphism that takes u' to v', as follows: Let d_1 be the first element from D not appearing in u'. Let $a_1 \in T(u')$ such that $u'd_1 \cong_B u'a_1$. By the assumption for this a_1, for each r there is some $b_1 \in T(v')$ such that $\forall b_2\ \exists a_2 \cdots \forall c\ \exists d\ (u'a_1a_2 \cdots a_r \cong_l v'b_1b_2 \cdots b_r)$, and where $b_2, a_2, ..., c, d$ are as described above. In particular, $u'a_1 \cong_l v'b_1$. Since T_B's outdegree is finite, there are infinitely many r's for which the same b_1 can be used. Denote the set of all such r's by M_1. We have $u'd_1 \cong_B u'a_1 \cong_l v'b_1$. Now, let e_1 be the first element from D not appearing in $v'b_1$, and let $b_2 \in T(v'b_1)$, such that $v'b_1b_2 \cong_B v'b_1e_1$. By the assumption, for each $r \in M_1$ there exists $a_2 \in T(u'a_1)$ such that $\forall a_3\ \exists b_3 \cdots \forall c\ \exists d\,(u'a_1a_2 \cdots a_r \cong_l v'b_1b_2 \cdots b_r)$, where $a_3, b_3, ..., c, d$ are as described above. In particular, $u'a_1a_2 \cong_l v'b_1b_2$. Again, by the finiteness of $T(u'a_1)$, there is an infinite subset M_2 of M_1, such that a single common element $a_2 \in T(u'a_1)$ can be used for each $r \in M_2$. Let h be the automorphism on D taking $u'a_1$ to $u'd_1$, and let a_2' be $h(a_2)$. So far we have $u'd_1a_2' \cong_B u'a_1a_2 \cong_l v'b_1b_2 \cong_B v'b_1e_1$. Continuing this back and forth construction for all members of the domain D yields the desired automorphism on D that takes u' to v'. ∎

Proposition 3.5 is stronger than Proposition 3.3, as it requires only finite formulas to establish an isomorphism between u and v. Indeed, Proposition 3.3 holds for every countable data base, but Proposition 3.5 does not. For example, consider the following graph, G, which is not highly symmetric:



Here, $(a, c) \not\cong_G (b, d)$, but one can use Ehrenfeucht–Fraisse games to show that (a, c) and (b, d) cannot be distinguished by any first-order formula, so that $\forall r(a, c) \equiv_r (b, d)$. Actually, we can strengthen Proposition 3.5:

PROPOSITION 3.6. *If B is highly symmetric then there is a fixed r such that for every $u, v \in D^n$,*

$$u \cong_B v \quad \text{iff} \quad u \equiv_r v.$$

Proof. The "only-if" direction is clear. We have to show that

$$\exists r\ \forall u, v \in D^n\ (\text{if } u \equiv_r v \text{ then } u \cong_B v). \qquad (*)$$

First, take the contra-positive of one direction of Proposition 3.5, and restrict the u and v to come from T^n:

$$\forall u, v \in T^n\ (\text{if } u \not\cong_B v \text{ then } \exists r\ u \not\equiv_r v).$$

Now, T^n is finite. Hence, we may denote by r_0 the maximum of the r's that exist according to this fact, taken over the pairs u and v. But since $u \not\equiv_r v$ implies $u \not\equiv_{r'} v$ for all $r' \geqslant r$, we obtain:

$$\forall u, \quad v \in T^n \qquad (\text{if } u \not\cong_B v \text{ then } u \not\equiv_{r_0} v).$$

Actually, this r_0 works for all $u, v \in D^n$, since the same r can be used to distinguish both between u and v and between the elements of some non-equivalent pair $u', v' \in T^n$, where $u \cong_B u'$ and $v \cong_B v'$. We thus obtain (*). ▮

An interesting corollary of this concerns elementary equivalence. Recall that two structures are *elementarily equivalent* if they satisfy precisely the same first-order sentences. Since any finite structure is definable by a first-order sentence, finite structures are elementarily equivalent iff they are isomorphic. However, it is easy to see that there exist non-isomorphic recursive structures (i.e., r-db's) that are elementarily equivalent. For example:

and

However, in terms of elementary equivalence, highly symmetric structures are like finite ones:

COROLLARY 3.1. *Highly symmetric data bases of the same type are isomorphic iff they are elementarily equivalent.*

Proof. Let $B_1 = (D_1, R_1, ..., R_k)$ and $B_2 = (D_1, R'_1, ..., R'_k)$ be highly symmetric data bases. The "only-if" direction is clear. For the "if" direction, assume that B_1 and B_2 are elementarily equivalent, and without loss of generality assume that D_1 and D_2 are disjoint, and that $a, b \notin (D_1 \cup D_2)$. Define B to be $(D_3, S_1, ..., S_k, E)$, where (i) $D_3 = D_1 \cup D_2 \cup \{a, b\}$, (ii) $S_i = R_i \cup R'_i$, for all $1 \leqslant i \leqslant k$, and (iii) $E = \{(a, x) \mid x \in D_1\} \cup \{(b, x) \mid x \in D_2\}$. Note that B is also highly symmetric and that $a \cong_B b$ iff $B_1 \cong B_2$.

Now, according to Proposition 3.5, $a \cong_B b$ iff $a \equiv_r b$ for each r. Hence, in order to show that $B_1 \cong B_2$, it suffices to show that for each r the duplicator has a winning strategy in the r-game over (B, a) and (B, b).

Let $r > 0$. Since B_1 and B_2 are elementarily equivalent, the duplicator has a winning strategy in the r-game over B_1 and B_2. In each step of the r-game over (B, a), (B, b), if the spoiler chooses an element $d_1 \in D_1$ (respectively, $d_1 \in D_2$), the duplicator chooses that element from D_2 (resp. D_1) that he/she would have chosen in the r-game over B_1 and B_2. (If the spoiler chooses a then the duplicator chooses b, and vice versa.) Let $a_1, ..., a_r$ be the elements that were chosen from D_1, and let $b_1, ..., b_r$ be the elements that were chosen from D_2. Since the game is conducted in a way that simulates the r-game over B_1 and B_2 and the duplicator wins that game, B_1 restricted to $a_1, ..., a_r$ must be isomorphic to B_2 restricted to $b_1, ..., b_r$. But therefore also B restricted to $a, a_1, ..., a_r, b_1, ..., b_r$ is isomorphic to B restricted to $b, b_1, ..., b_r, a_1, ..., a_r$. This means that the duplicator wins the r-game over (B, a), (B, b). ▮

DEFINITION 3.5. Denote by V_r^n the partition of T^n into the equivalence classes of $\equiv_r$. Similarly, denote by V^n the partition of T^n into the equivalence classes of $\cong_B$.

Thus, V^n is a set of sets, and in fact each of its elements is a singleton. Now, by Proposition 3.6 we have

COROLLARY 3.2. *For highly symmetric B, there is an r such that $V^n = V_r^n$.*

DEFINITION 3.6. For $V \subseteq T^n$, we let $V\hat{\downarrow} = \{u \mid \exists a \in D. ua \in V\}$, and if $V_r^n = \{V_1, ..., V_k\}$, we let $V_r^n\hat{\downarrow} = \{V_1\hat{\downarrow}, .., V_k\hat{\downarrow}\}$.

Note that the $\hat{\downarrow}$ operator yields a set, so that $V_r^n\hat{\downarrow}$ might have less than k elements.

PROPOSITION 3.7. $V_r^{n+1}\hat{\downarrow} = V_{r+1}^n$.

Proof. Let $V_r^{n+1} = \{V_1, ..., V_l\}$. In order to prove $\{V_1\hat{\downarrow}, ..., V_l\hat{\downarrow}\} = V_{r+1}^n$, we show that for all $u, v \in T^n$, $u \not\equiv_{r+1} v$ iff for some $1 \leqslant i \leqslant l$, $u \in V_i\hat{\downarrow}$, but $v \notin V_i\hat{\downarrow}$.

Let $u, v \in T^n$, and let i be such that $u \in V_i\hat{\downarrow}$ but $v \notin V_i\hat{\downarrow}$. There must exist $a \in T(u)$ with $ua \in V_i$, but there is no $b \in T(v)$ with $vb \in V_i$. By Definition 3.5, $ua \not\equiv_r vb$. By Definition 3.4, it follows that $u \not\equiv_{r+1} v$.

For the converse, assume that $u \not\equiv_{r+1} v$. By Definition 3.4, and without loss of generality, there is $a \in T(u)$ such that for all $b \in T(v)$ we have $ua \not\equiv_r vb$. Let $V \in V_r^{n+1}$ be the set containing ua. By Definition 3.5, $vb \notin V$ for all $b \in T(v)$. Hence, $V\hat{\downarrow}$ contains u, but not v. ▮

If we take $\hat{\downarrow}^r$ to stand for $\hat{\downarrow} \cdots \hat{\downarrow}$ with r occurrences of $\hat{\downarrow}$, we have:

COROLLARY 3.3. $V_0^{n+r}\hat{\downarrow}^r = V_r^n$.

3.3. *Completeness of* QL_{hs}

In order to be able to discuss query languages for highly symmetric data bases, we have to decide on their representation. The convention will be to represent a data base B by a tuple of the form

$$C_B = (T_B, \cong_B, C_1, ..., C_k),$$

where T_B is some characteristic tree for B, $\cong_B$ is the tuple equivalence relation, and each C_i for $1 \leqslant i \leqslant k$, is the set of representatives of the equivalence classes constituting R_i

that appear in T_B. (Note that since $\cong_B$ takes into account the existence and non-existence of tuples in the relations of B, each R_i must indeed be a union of whole equivalence classes.)

For a general infinite data base, the C_i's might be infinite, and the very components of C_B might not be computable. We restrict ourselves here to highly symmetric r-db's for which C_B is computable, and this alleviates both problems.

DEFINITION 3.7. B is a *highly symmetric recursive data base* (or an *hs-r-db*, for short) if B can be represented by some $C_B = (T_B, \cong_B, C_1, ..., C_k)$, such that T_B is highly recursive and $\cong_B$ is a recursive predicate.[6]

Note that, given $C = (T, \cong, C_1, ..., C_k)$ satisfying the conditions in Definition 3.7, one can compute the data base B for which $C = C_B$, since for each i, $u \in R_i$ iff $u \cong v$ for some $v \in C_i$, and the C_i must be finite. On the other hand, given an r-db B that happens to be highly symmetric, C_B is not necessarily computable from B, since T_B contains additional information about B that might not be computable. It follows that each hs-r-db is a highly symmetric r-db, but not every r-db that is highly symmetric is indeed an hs-r-db.

EXAMPLE. We can show that for each a there is a countable random structure that is an hs-r-db of type a. Here is how: In [HH2], we show that there exists a *recursive* countable random structure A. By the proof of Proposition 3.2, tuples are equivalent in A iff they are locally isomorphic, which means that for A, $\cong_l$ and $\cong$ are the same. Since by Proposition 2.2, $\cong_l$ is recursive, so is $\cong_A$. In addition, T_A is highly recursive, since for any x appearing in T_A, the number of immediate offspring of x is the same as the number of equivalence classes of $\cong_l$ containing tuples of the form xa. This number is finite and can be computed. Hence, in order to compute $T_A(x)$ it suffices to find sufficiently many non-equivalent tuples of the form xa, using the recursive predicate $\cong_A$.

DEFINITION 3.8. An *hs-r-query of type a* (or an *hs-r-query* for short) is a partial function Q, which, for each hs-r-db B of type a, yields an output (if any) that is a relation over $D(B)$.

DEFINITION 3.9. An hs-r-query Q is *recursive* if there is a Turing machine with oracles for T_B and $\cong_B$,[7] which, on input $C_1, ..., C_k$, does not halt if $Q(B)$ is undefined and otherwise outputs the representatives from T_B of the equivalence classes constituting the relation $Q(B)$.

Genericity of hs-r-queries is defined as in Definition 2.5.

[6] A recursive tree T is called *highly recursive* if it is finitely branching and the function $T(x)$ that yields the finitely many immediate offspring of a given node is recursive too.

[7] The oracle for T_B yields $T_B(x)$ when applied to a node x, and the oracle for $\cong_B$ decides, given tuples u and v, whether $u \cong_B v$.

DEFINITION 3.10. An hs-r-query is *computable* if it recursive and generic.

DEFINITION 3.11. Let Q be an hs-r-query of type a. An expression E in a query language *expresses* Q if for every hs-r-db B of type a, and each C_B representing B, either $E(C_B)$ and $Q(C_B)$ are both undefined, or they are both defined and yield sets of representatives of the same equivalence classes. A query language is *hs-r-complete* if it expresses precisely the computable hs-r-queries.

We now define our query language for hs-r-db's, QL_{hs}. It is a slight variation of the QL language for finite data bases, proposed by Chandra and Harel [CH]. We provide a brief description of the syntax and semantics of QL_{hs} here, but the reader is referred to [CH] for a more gradual exposition of the finitary version. QL_{hs} is a programming language with relational variables of dynamically changing ranks. Queries are expressed by programs.

Syntax

$Y_1, Y_2, ...$ are variables in QL_{hs}. The set of *terms* of QL_{hs} is defined inductively as follows:

1. E is a term, and for $i \geq 1$, Rel_i and Y_i are terms.
2. For any terms e and f,

$$(e \cap f), (\neg e), (e\uparrow), (e\downarrow), \text{ and } (e^{\sim})$$

are terms.

The set of *programs* of QL_{hs} is defined inductively as follows:

1. $Y_i \leftarrow e$ is a program, for a term e and $i \geq 1$.
2. For programs P and P'

$$(P; P')$$

is a program.

3. For a variable Y_i and a program P,

$$\textbf{while } |Y_i| = 0 \textbf{ do } P$$

and

$$\textbf{while } |Y_i| = 1 \textbf{ do } P$$

are programs.[8]

[8] QL employed the emptiness test $|Y_i| = 0?$ only. Indeed, in the finite case, $|Y| = 1?$ is definable in QL, via the definable relation perm(D), which consists of all permutations on the n elements of D. In our case, however, perm(D) is of infinite rank, and we were not able to see how to define $|Y| = 1?$ using the rest of QL_{hs}, with an emptiness test. Hence we have added $|Y| = 1?$.

The *rank* of a term is defined in the usual way; see [CH].

Semantics

Let $B=(D, R_1, ..., R_k)$ be an hs-r-db of type $a=(a_1, ..., a_k)$, represented by $C_B=(T_B, \cong_B, C_1, ..., C_k)$.

1. Terms take on as values finite sets of representatives of the equivalence classes of $\cong_B$ of some common rank, but those labeling paths in T_B. In other words, at any point during the computation of a program each term contains the labels along some paths in T^n, for some n.

2. Variables are initialized to the empty set.

3. The fixed term E is $T^2 \cap \{(a, a) \mid a \in D\}$, and Rel_i, for $1 \leqslant i \leqslant k$, contains the input relation C_i.

4. Let e and f be terms of rank n.

- $e \cap f$ is defined in the usual way.
- $\neg e$ is $T^n - e$.
- $e\uparrow = \{ud \mid u \in E, \text{ and } ud \in T^{n+1}\}$.
- $e\downarrow$ contains those tuples from T^{n-1} that are equivalent by $\cong_B$ to tuples obtained by projecting out the first coordinate from the tuples in e.
- $e^{\sim}$ contains those tuples from T^n that are equivalent to tuples obtained by exchanging the two rightmost coordinates of the tuples in e.

5. The tests $|Y_i| = 0?$ and $|Y_i| = 1?$ used in the **while** statement are true iff Y_i is empty or contains a single representative, respectively.

The programs of QL_{hs} thus really act on the representation C_B. As in [CH], the result of applying a program P to C_B is undefined if P does not halt; otherwise it is taken to be the contents of some fixed variable, say Y_1.

THEOREM 3.1. *QL_{hs} is hs-r-complete.*

Proof. First, note that QL_{hs} programs are computable when applied to hs-r-db's. To see this, recall that B is represented by C_B, where T_B is highly recursive (in particular, it is finitely branching), $\cong_B$ is recursive, and the terms are finite sets of representatives that appear in T_B. The operations $\cap$, $\neg$, $\uparrow$, and E are carried out using the relevant terms and T_B. For example, to compute $e\uparrow$, we find the direct offspring in T_B of the tuples in e. To carry out $\downarrow$ and $\sim$, $\cong_B$ is also needed. For example, $e\downarrow$ is computed by finding all the paths in T_B of length $\mathrm{rank}(e)-1$ that are equivalent to some tuple in $\{(a_2, ..., a_n) \mid (a_1, a_2, ..., a_n) \in e\}$. Finally, the QL_{hs} test $|Y| = 1?$ is also recursive.

Second, QL_{hs} does not violate genericity, since its operations are a subset of the generic operations of the relational algebra (but are carried out directly on the representatives of equivalence classes instead of using the whole relations), and the test $|Y| = 1?$ cannot distinguish between non-isomorphic relations. Hence QL_{hs} expresses computable queries only.

For the other direction, we follow closely the completeness proof for QL from [CH]. First, note that QL_{hs} can be thought of as having counters: $E\downarrow\downarrow$ plays the role of 0, and if e plays the role of the natural number i, then $e\uparrow$ and $e\downarrow$ play the role of $i+1$ and $i-1$, respectively. Testing whether e is "equal" to 0 is accomplished by testing $e\downarrow$ for emptiness. This gives QL the power of general counter machines (and hence of Turing machines), with numbers represented by the ranks of the relations in the variables.

In addition, the conventional operators on relations appearing in [CH], such as **if** Y **then** P **else** P', rank(e), Cartesian product, etc., can be programmed in QL_{hs} precisely as is done in [CH] for QL. The reason for this is that the meanings of QL_{hs} operations (except for the new test $|Y_i| = 1?$) are just as in QL, the only difference being that here the resulting relations are represented by the equivalence classes. The test $|Y| = 1?$ is not needed for these operations; it is needed later, however.

Let Q be a recursive generic hs-r-query of type $a=(a_1, ..., a_k)$. We exhibit a program $P_Q \in \mathrm{QL}_{hs}$ that computes Q. Given C_B of type a, the computation of P_Q proceeds according to the following main steps, which are analogous to those given in the proof in [CH]:

1. Compute a tuple d, such that $C_1, ..., C_k$ can be obtained by projections on d; i.e., there are finite sets $X_1, ..., X_k$ of tuples over $\mathcal{N}$ such that for each j, $1 \leqslant j \leqslant k$, we have $\bigcup_{(i_1, ..., i_{a_j}) \in X_j} d_{[i_1, ..., i_{a_j}]} = C_j$.

2. Compute a particular $X=(X_1, ..., X_k)$ satisfying the condition appearing in step 1 using d, by guessing X_i's and checking equality to the C_i's. Note that this X can be thought of as representing an hs-r-db B_N over the domain $\mathcal{N}$, which is isomorphic to the input data base B.

3. Compute $Q(B_N)$ (which is actually Q applied to the $(X_1, ..., X_k)$ representation of B_n) using the recursiveness of Q.

4. Compute $Q(C_B)$ from $Q(X)$ using d:

$$Q(C_B) = \bigcup_{(i_1, ..., i_m) \in Q(X)} d_{[i_1, ..., i_m]}.$$

Steps 2 and 4 can be programmed in QL_{hs} just as in [CH], but using the computed d (from Step 1) instead of the entire equivalence class of d, which was called CG_d^B in [CH].

As for Step 3, its computation is based on cooperation between the Turing machine capability of QL_{hs} and oracles for T_B and $\cong_B$. Since Q is recursive, there is a Turing machine M that computes Q. To compute $Q(B_N)$, our program P_Q follows the computation of M on B_N. Whenever $M(B_N)$ asks for $T_{B_N}(x)$, P_Q projects d on x to yield $d_{[x]}$, computes $d_{[x]}\uparrow$, and then encodes back to the appropriate tuples over $\mathcal{N}$. If the tuple x contains constants that are larger than the length of d (and this could happen, because

d was computed in Step 1 only to encode $C_1, ..., C_k$), and thus d is not sufficient for expressing the necessary representatives, P_Q computes a larger d as it did for the original one in Step 1. Whenever $M(B_N)$ asks whether $x \cong_{B_N} y$, P_Q checks if $d_{[x]} = d_{[y]}$.

We now show how to compute d in Step 1, based on Corollaries 3.2 and 3.3. Our goal is to compute V^n, which contains one representative for each equivalence class of $\cong_B$ of rank n, and somehow isolate from it a single representative to be the required d, which is to be a tuple of distinct elements that satisfies the condition appearing in Step 1.

First, note that $\hat{\downarrow}$ can be easily programmed in QL_{hs}. Now, P_Q stores a partition $V = \{V_1, ..., V_l\}$ by the Cartesian product $V_1 \times \cdots \times V_l$. It can then isolate each V_i by projecting out the unnecessary coordinates. In order to find a suitable d, P_Q computes the V^n successively, for $n = 1, 2, ...$, as shown below, and checks if there is a $V \in V^n$ that represents a "good" d. There must be *some* V^n that contains a suitable d. (Actually, n will be the number of distinct elements from D appearing in $C_1, ..., C_l$.) Thus the process is always finite.

To compute V^n for a given n, P_Q computes, in order, V_0^n, V_1^n, V_2^n, ... as shown below, until it finds $V_r^n = \{V_1, ..., V_l\}$ for which $|V_i| = 1$ for each i. This V_r^n is an adequate V^n. While Corollary 3.2 asserts the existence of such a V_r^n, it does not provide the r. Hence our need to test $|V_i| = 1$ in order to verify that the partition V_r^n is fine enough to be a V^n.

To compute V_r^n for a given n and r, note that, by Corollary 3.3, $V_r^n = V_0^{n+r} \hat{\downarrow}^r$. Accordingly, P_Q computes V_0^{n+r} as shown below, and then performs $V \hat{\downarrow}^r$ for each $V \in V_0^{n+r}$.

Finally, V_0^{n+r} is computed as follows: P_Q starts with $V \leftarrow (E\downarrow\downarrow)\uparrow^{n+r}$, which is actually T^{n+r}. It then refines V by checking the containment or non-containment of all possible projections of the appropriate tuples in the relations of B. At each step, some component from the partition V is removed, and is split into two nonempty components, denoted A and B, one containing the tuples u that satisfy $u_{[i_1, ..., i_m]} \in R_j$, for some m, j, $i_1, ..., i_m$, and the other containing all the others. It then adds A and B to V by $V \leftarrow V' \times A \times B$ where V' is the previous V with the component $A \cup B$ removed. The refinement is completed when it is impossible to perform any more nontrivial splits. In this case, each component contains tuples that are locally equivalent, and different components contain non-locally-equivalent tuples; i.e., V is an adequate V_0^{n+r}. ∎

4. FINITE AND CO-FINITE RELATIONS

It turns out that QL_{hs}, or variants thereof, are complete in some additional cases. We present one here.

DEFINITION 4.1. B is a *finite/co-finite r-db* (or an *fcf-r-db* for short) if B is an r-db whose relations are finite or co-finite, such that the finite relations are represented by their finite set of tuples, and the co-finite ones are represented by their finite complement and a special indicator.

Note that an r-db that happens to contain only finite and co-finite relations is not necessarily an fcf-r-db. An fcf-r-db contains the additional indication of the finiteness of the relations, which is not recursive in general.

We denote by D_f the set of all constants from $D(B)$ appearing in the finite parts of the relations.

PROPOSITION 4.1; *B is an fcf-r-db if and only if B is an hs-r-db whose relations are finite or co-finite.*

Proof. If B is an fcf-r-db, it contains only finitely many equivalence classes of each rank, since the elements appearing only in the infinite parts of the relations (i.e., the elements in $D(B) - D_f$) are all equivalent. In addition, T_B is a highly recursive tree and $\cong_B$ is recursive, since they can be computed from the finite parts of the relations. Thus, B is actually an hs-r-db.

Let B be an hs-r-db whose relations are finite or co-finite, that is, represented by C_B. Note that it suffices to compute D_f, since the required representation of an fcf-r-db is easily computed from D_f and C_B.

Let $d = (d_1, ..., d_n)$ be the shortest tuple in T_B satisfying the following: (i) for every i and j with $1 \leqslant i \neq j \leqslant n$, we have $d_i \neq d_j$; (ii) $T(d)$ contains only one tuple $(d_1, ..., d_n, d_{n+1})$ such that $d_{n+1} \neq d_i$ for all $1 \leqslant i \leqslant n$.

We claim that $\{d_1, ..., d_n\}$ is indeed D_f. First, note that each tuple $x = (x_1, ..., x_m)$ in T_B not containing all the elements in D_f must have at least two immediate offspring in T_B satisfying condition (ii)—one for representing the addition of an element from D_f, and the other for representing the addition of an element from $D(B) - D_f$. Hence, a d that satisfies condition (ii) must contain all the elements in D_f. Also, d cannot contain elements from $D(B) - D_f$, otherwise there would be a shortest tuple satisfying the conditions, consisting of the elements in D_f only in some order. Since T_B is highly recursive, d is computable. ∎

COROLLARY 4.1. *QL_{hs} is complete for fcf-r-db's.*

What kind of output relations may we expect to obtain using programs in QL_{hs} applying to fcf-r-db's? Clearly, unions, complementations, and $\sim$'s preserve the property of being finite or co-finite. It is also clear that projections on finite relations yield finite relations. As for projections on co-finite relations, we have:

PROPOSITION 4.2. *If $R \subseteq D^n$, for $n \geqslant 1$, is co-finite, then $R\downarrow = D^{n-1}$, which is finite for $n = 1$ and is otherwise co-finite.*

Proof. Let $R \subset D^n$, for $n \geqslant 1$, be a co-finite relation, and let $\bar{R} = D^n - R$. We have:

$$R\downarrow = \{(x_2, ..., x_n) \mid \exists x_1 (x_1, ..., x_n) \in R\}$$
$$\overline{R\downarrow} = \{(x_2, ..., x_n) \mid \forall x_1 (x_1, ..., x_n) \in \bar{R}\}.$$

Note that $\overline{R\downarrow} = \varnothing$, since if there would be some $(x_2, ..., x_n)$ for which for all x_1, $(x_1, ..., x_n) \in \bar{R}$, then $\bar{R}$ would be infinite. But R is co-finite. Hence $R\downarrow = D^{n-1}$. If $n = 1$, then $R\downarrow$ is finite, since $D^0 = \{(\,)\}$. ∎

In contrast to this, $\uparrow$ does not preserve finiteness or co-finiteness: For example, if R is a finite relation, $R\uparrow$ is neither finite nor co-finite. Hence QL_{hs} yields output relations that need not necessarily be finite or co-finite.

To remedy this, we present a version of the finitary QL of [CH], denoted QL_{f+}, that expresses precisely the computable and generic queries that yield only finite or co-finite relations as output when applied to fcf-r-db's. The syntax of QL_{f+} is like that of QL, with the following addition: For a variable Y and a program P,

$$\textbf{while } |Y| < \infty \textbf{ do } P$$

is a program. The semantics of QL_{f+} is the same as that of QL [CH], except for the following:

1. A term takes on as value a finite set of tuples consisting of the relation itself in case of a finite relation, or the complement of the relation and a special indicator, in case of a co-finite one.

2. Two changes in the definition of the operations are:

- $e\uparrow = e \times D_f$, and is defined only if e is finite.
- $E = \{(a, a) \mid a \in D_f\}$.

The rest of the operations are defined in the usual way, but are carried out using only the finite parts of the relations and the special indicator. For example, $\neg e$ is computed by simply flipping the indicator from present to absent and vice versa. Also, if e is finite and f co-finite, then $e \cap f$ is computed as $e - (\neg f)$; i.e., by removing from e the finitely many tuples in $\neg f$.

3. $|Y| < \infty$ is true iff Y contains a finite relation.[9]

The result of applying a program P to B is undefined if P does not halt; otherwise, some fixed variable, say Y_1 contains the finite part of the answer, and some other variable, say Y_2, contains $\{(\,)\}$ if the answer is co-finite, and is otherwise empty.

PROPOSITION 4.3. QL_{f+} *expresses precisely the computable queries that yield only fcf-relations when applied to fcf-r-db's.*

Proof. Clearly, QL_{f+} expresses only recursive and generic queries, and these yield only fcf-relations.

[9] Note that here we do not need a special term for capturing the finitary elements of the domain, since we have "hard-wired" it into the semantics: D_f can be obtained by $E\downarrow$.

For the converse, let Q be such a query. We construct a program $P_Q \in \mathrm{QL}_{f+}$, based on the program constructed in [CH] for the finitary completeness of QL. Note that the isomorphisms of a fcf-r-db can be computed by using only the finite parts of the relations, since constants that appear in infinite parts only are all equivalent:

P_Q consists of the following main steps:

1. Prepare $Z = (D_f, Z_1, ..., Z_k)$ to be the data base consisting of the finite parts of $R_1, ..., R_k$.

2. Compute the set of automorphisms of Z.

3. Compute an internal "model" data base Z_N over $\mathcal{N}$, isomorphic to Z.

4. Compute $Y = (Y_1, ..., Y_k)$, where

$$Y_i = \begin{cases} \{(1)\} & \text{if } |R_i| < \infty \\ \{(0)\} & \text{if } |R_i| = \infty \end{cases}$$

5. Compute $Q(Z, Y)$ using the Turing machine capability.

6. Compute the finite part of $Q(B)$ from $Q(Z, Y)$ using the set of automorphisms,

7. Compute the special indicator that marks the finiteness of the answer, from $Q(Z, Y)$.

Step 1 can be carried out easily using the test for finiteness, and Z is then just a finite data base over D_f. This means that Step 2 can be carried out exactly as in [CH], but with the operations computed with respect to D_f (which is $E\downarrow$), instead of the whole D. Thus, $\neg X$ will be $\neg X \cap (E\downarrow)^n$, where $n = \mathrm{rank}(X)$, and $X\uparrow$ will be $X \times E\downarrow$. ∎

5. COMPLETENESS OF GENERIC MACHINES

In [AV], a somewhat different approach to designing a complete language for finite data bases was taken, resulting in *generic machines* (GMs). Here is a rephrasing of excerpts from [AV]:

A GM consists of a TM interacting with a relational store. The store contains relations over a finite set, which are represented as fixed-arity predicates. Some of these are designated as input or output predicates. GM allows transferring tuples between the relational store and the tape. Loading a relation with n tuples to the tape has the effect of spawning n copies of the machine, with one tuple appended to the tape of each copy. Each individual copy is called a unit-GM. The unit-GM's, which proliferate as a result of load operations, then compute synchronously in parallel, with each having its own finite state control, tape, and relational store. Initially, the GM consists of a single unit-GM in the start state, working in an empty tape, and with the input relations in the relational store. If several unit-GM's

simultaneously reach the same state and identical tape contents, they collapse automatically into a single unit-GM, whose relational store is the union of their relational stores. At the end of the computation, there must be a single machine in a halting state with an empty tape.

We turn GM's into an hs-r-complete language GM_{hs}, as follows:

Given an hs-r-db B represented by $C_B=(T_B, \cong_B, C_1, ..., C_k)$, the relational store of the GM_{hs} will contain $C_1, ..., C_k$ as finite relations, and the GM_{hs} will use the oracles for T_B and $\cong_B$ in its calculations.

The tape cells hold constants from two alphabets. The first is the Turing machine alphabet, which is used only by the finite state control. The second is the domain of B. Symbols denoting elements from $D(B)$ can be loaded on the tape from the store or from the oracle for T_B, and can also be placed in the store. The finite state control may refer to these symbols in testing for equality of symbols (see Test 3 below) and in checking the equivalence of tuples (Test 4 below). The GM_{hs} has two heads, and in the descriptions below when only one head is needed the first head is intended.

In each GM_{hs}, transitions depend on:

1. the current state;

2. the contents of the current tape cell, when it contains a symbol from the Turing machine alphabet;

3. equality or non-equality of the contents of the current tape cell and another tape cell pointed to by the second head, when the tape cells contain symbols from $D(B)$;

4. the answer to the question "Is $u \cong_B v$?", where u and v are pointed to by the two heads.

Transitions involve the following actions:

(i) move the heads right or left on the tape;

(ii) overwrite current tape cell with some tape symbol;

(iii) change state;

(iv) load the representatives of some relation from the relational store onto the tape;

(v) load the offspring of the current tuple from T_B onto the tape (the current tuple consists of the symbols between the two heads);

(vi) store a tuple from T_B that is equivalent to the current tuple in the relational store.

The loading in operations (iv) and (v) are carried out by spawning copies of the machine, with one tuple appended to the tape of each copy.

THEOREM 5.1. *GM_{hs} is hs-r-complete.*

Proof. The programs in GM_{hs}, when applied to hs-r-db's, are clearly recursive. Also, they compute generic queries, due to the machines "splitting" parallelism and the fact that the only operations allowed concerning elements from the domain are comparisons of type 3 and 4 above and storing representatives from T_B.

For the other direction, let Q be a computable hs-r-query. The program P_Q that computes Q first loads onto the tape the $C_1, ..., C_k$ relations and the finite tree T^n, where n is the maximum rank of the C's. This involves many single load operations, and results in many unit-GM_{hs}. Too many, in fact, since identical tuples appear in most of the spawned machines; P_Q then discontinues the ones that loaded identical tuples. After all this, each unit-GM_{hs} contains the tuples of all of the C_i and the three T^n in a different order.

Here is a somewhat more detailed description of the loading process. To load relation C_i, repeated "load C_i" operations are carried out. After each of them, every newly spawned unit-GM_{hs} has to decide whether the loaded tuple is identical to some former tuple that was already loaded or is new. If the loaded tuple is old, it erases the tape and enters the halting state. If the loaded tuple is new, the unit-GM_{hs} has to decide whether its tape already contains all the tuples of C_i or to keep loading additional tuples. This decision is made as follows: The unit-GM_{hs} carries out an additional "load C_i", resulting in new unit-GM_{hs}'s, each containing an additional tuple from C_i. Call these machines $U_1, ..., U_m$. Now, each U_j checks if the loaded tuple is new or old. If it is old, U_j erases this tuple from the tape, and enters a specific state, q. If the tuple is new, U_j stores this tuple in the relational store, eases it from the tape, and enters the state q. After all the U_j's do this, they collapse, since their tapes are identical and they are in the same state. The new relational store will contain the union of the previous stores. Now, if the appropriate store in the collapsed machine is empty, then the present unit-GM_{hs} already contains the whole of C_i, and hence it can stop its loading; otherwise, there are more tuples in C_i, so that additional "load C_i" operations are needed.

After the loading, each unit-GM_{hs} encodes $C_1, ..., C_k$ and T^n by tuples of integers, by guessing tuples of integers and checking if equal elements from $D(B)$ correspond to equal integers, and different elements correspond to different integers. Now, the Turing machine M that computes Q can be applied to the integer version of the input B. Whenever M requires a new integer as part of the data base, P_Q loads more levels from T_B onto the tape, and encodes them, until it obtains that integer. Whenever M asks "Is $u \cong_B v$?", P_Q decodes u and v into tuples over constants from the input, and performs a test of type 4.

At the end of M's computation, P_Q decodes the output into tuples over the input constants and stores them in the relational store. Finally, P_Q erases the tapes of all the unit-GM_{hs}'s, and enters the halting state. Now, all the unit-GM_{hs}'s collapse into a single unit-GM_{hs} whose relational store is the union of their stores. Since M is generic, the relational stores of all the unit-GM_{hs} are the same, and hence the computation is completed successfully. ∎

6. BP-COMPLETENESS

In this section, we discuss BP-completeness over recursive data bases. The term, taken from [CH], was first studied by Bancilhon [B] and Paredaens [P] for finite data bases. Instead of referring to the ability of a language to express functions, it refers to its ability to define the relations that preserve the automorphisms of B, for any fixed B.

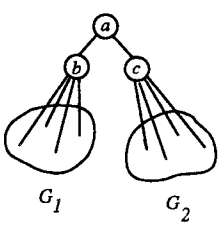
DEFINITION 6.1 A relation R *preserves the automorphisms of* B if, for all u, v, whenever $u \cong_B v$ we have $u \in R$ iff $v \in R$.

We first address the case of the full class of recursive data bases.

DEFINITION 6.2. A language is *BP-r-complete* if for each r-db B it expresses precisely the relations that are recursive and preserve the automorphisms of B.

THEOREM 6.1. *There is no effective BP-r-complete language.*[10]

Proof. By Proposition 2.1, the isomorphism problem for recursive graphs is Σ_1^1-hard. If an effective BP-r-complete language L would exist, this problem would be co-r.e., as follows.[11] Given two recursive graphs $G_1 = (D_1, E_1)$ and $G_2 = (D_2, E_2)$, consider the r-db $B = (D, R_1, R_2)$, where $D = D_1 \cup D_2 \cup \{a, b, c\}$, $R_1 = \{a\}$ and $R_2 = E_1 \cup E_2 \cup \{(a, b), (a, c)\} \cup \{(b, v) \mid v \in D_1\} \cup \{(c, u) \mid u \in D_2\}$. We assume that D_1 and D_2 are disjoint and $a, b, c \notin D_1 \cup D_2$. Actually, R_2 is the following graph:



where b and c are connected to every point in G_1 and G_2, respectively.

Now, b and c are equivalent in B (i.e., $b \cong_B c$) if and only-if G_1 and G_2 are isomorphic. In addition, there are no candidate elements for being equivalent to b besides c, due to the element a (which is the only element in R_1, and is connected only to b and c in R_2). Hence, $b \ncong_B c$ iff there exists a recursive relation that preserves the automorphisms of B and contains b but not c. For example, $\{b\}$ is such a relation. If a BP-complete language L were to exist, there would be an expression that expresses this relation over B. Thus, in order to determine whether G_1 and G_2 are not isomorphic, it would suffice to check that there exists an expression in L which, when applied to B, expresses some relation that contains b but not c. Since L is effective, this check is r.e. ∎

It is possible to modify this proof to show that there is no effective BP-complete language even for the case where the output relations are restricted to $\{1, ..., n\}$, for some n. Simply, take $a = 1$, $b = 2$, and $c = 3$, and make the other constants of B be the rest of the integers. Now consider expressions in L restricted to outputs over $\{1, 2, 3\}$ only.

We are better off in the case of unary data bases (in which case completeness captures computability over recursive sets):

PROPOSITION 6.1. *If B is an unary r-db, then $u \cong_B v$ iff $u \cong_l v$.*

Proof. The "only-if" direction is clear. For the "iff" direction take $u = (u_1, ..., u_n)$ and $v = (v_1, ..., v_n)$ to be such that $u \cong_l v$, and let $\{d_1, d_2, ...\}$ be the other constants of $D(B)$. Here is an automorphism that takes u to v:

$$\begin{pmatrix} u_1 \cdots u_n v_1 \cdots v_n d_1 d_2 \cdots \\ v_1 \cdots v_n u_1 \cdots u_n d_1 d_2 \cdots \end{pmatrix}. \quad \blacksquare$$

THEOREM 6.2. *$\mathcal{L}^-$ is BP-complete for unary r-db's.*

Proof. Let B be a unary r-db, and let R be a recursive relation that preserves the automorphisms of B. By Proposition 6.1, R also preserves local automorphisms. Hence it consists of the union of some equivalence classes of $\cong_l$. These can be expressed in $\mathcal{L}^-$, as we have shown in Theorem 2.1. ∎

Now to the case of highly recursive data bases.

DEFINITION 6.3. A language is *BP-hs-r-complete* if for each hs-r-db B it expresses precisely the relations that are recursive, highly symmetric, and preserve the automorphisms of B.

Actually, we need not explicitly say that the output relation is highly symmetric, since if it preserves automorphisms of B the number of equivalence classes of $\cong_R$ cannot be larger than that of $\cong_B$.

THEOREM 6.3. *The first-order relational calculus $\mathcal{L}$ is BP-hs-r-complete.*

Proof. For the first direction, note that the expressions in $\mathcal{L}$ clearly preserve the automorphisms of B. As noted, this also shows that the output relation is highly symmetric. We have to show that the relations expressible in $\mathcal{L}$ are

[10] A language is effective if its set of expressions is recursive, and evaluating an expression on a given recursive data base is computable.

[11] The Σ_1^1-hardness result of [M] is not needed here, and it suffices to use a weaker result, which happens to be simpler to prove, to the effect that the isomorphism problem for recursive graphs is Π_2^0-hard.

recursive. Consider, for example, determining whether $u \in R$, for a relation R defined in $\mathscr{L}$ by

$$R = \{(x_1, ..., x_n) \mid \exists y_1 \forall y_2 \cdots \Phi y_k \phi(x_1, ..., x_n, y_1, ..., y_k, B)\}.$$

It suffices to first find in T^n the representative v that is equivalent to u, and then to evaluate the quantifiers only over the finitely many elements from T^{n+k}. It is not necessary to evaluate the quantifiers over all of D, since each of the other elements is equivalent to one of the representatives, and hence would produce the same answers. The tuple v can be found using the given recursive $\cong_B$.

For the other direction, let B be an hs-r-db, and let R be some recursive relation of rank n that preserves the automorphisms of B. The equivalence classes contained in R are coarser than those of B, and they must therefore be represented by some elements in T^n. According to the discussion in Section 3, there representatives from T^n can be isolated by operating only on the representatives from T^{n+r}, for some $r \geqslant 0$ (see Proposition 3.6). Since B is fixed, r is a specific fixed constant, and hence the isolated representatives are expressible in $\mathscr{L}$. The required expression consists of a disjunction of the subexpressions corresponding to the appropriate isolated representatives. ∎

$\mathscr{L}$ is trivially BP-complete also for fcf-r-db's when the answers are not required to be finite or co-finite, since fcf-r-db's are special cases of hs-r-db's. When considering fcf-r-db's whose output relations are required to be finite or co-finite, it is possible to show that $\mathscr{L}$ restricted to D_f is BP-complete. The same is true for a relational algebra containing operations that yield only finite/co-finite relations, such as the operations of QL_{f+}. Any finite/co-finite relation that preserves the automorphisms of the fcf-r-db can be expressed by applying operations only to the finite parts of the relations with respect to D_f. At the end, if the output relation has to be co-finite, a complementation symbol is applied to the finite part of the answer.

ACKNOWLEDGMENTS

We thank the referees for their helpful comments.

REFERENCES

[AV] S. Abiteboul and V. Vianu, Generic computation and its complexity, *in* "Proc. 23rd ACM Symposium on Theory of Computing," pp. 209–219, ACM Press, New York, 1991.

[B] F. Bancilhon, On the completeness of query languages for relational data bases, *in* "Proc. 7th Symposium on Math. Foundations of Computer Science, Zakophane, Poland," Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York/Heidelberg, 1978.

[Be1] D. R. Bean, Effective coloration, *J. Symbolic Logic* **41** (1976), 469–480.

[Be2] D. R. Bean, Recursive Euler and Hamiltonian paths, *Proc. Amer. Math. Soc.* **55** (1976), 385–394.

[BG] R. Beigel and W. I. Gasarch, On the complexity of finding the chromatic number of a recursive graph, I & II, *Ann. Pure Appl. Logic* **45** (1989), 1–38, 227–247.

[CH] A. K. Candra and D. Harel, Computable queries for relational data bases, *J. Comput. Systems Sci.* **21** (1980), 156–178.

[CK] C. C. Chang and H. J. Keisler, "Model Theory," 3rd ed., North-Holland, Amsterdam, 1990.

[E] A. Ehrenfeucht, An application of games to the completeness problem for formalized theories, *Fund. Math.* **49** (1961), 129–141.

[Fa] R. Fagin, Probabilities on finite models, *J. Symbolic Logic* **41** (1976), 50–58.

[Fr] R. Fraisse, "Cours de Logique Mathématique," Gauthier-Villars/Nauwelaerts, 1967.

[H] D. Harel, Hamiltonian paths in infinite graphs, *Israel J. Math.* **76** (1991), 317–336; also, *in* "Proc. 23rd ACM Symposium on Theory of Computing, New Orleans," pp. 220–229, ACM, New York, 1991.

[HH1] T. Hirst and D. Harel, Taking it to the limit: On infinite variants of NP-complete problems, *J. Comput. System Sci.*, to appear; also *in* "Proc. 8th IEEE Conference on Structure in Complexity Theory," IEEE Press, New York, 1993.

[HH2] T. Hirst and D. Harel, More about recursive structures: Descriptive complexity and zero-one laws, *in* "Proceedings, 11 th Sympos. on Logic in Computer Science," IEEE Press, New York, to appear.

[HY] R. Hull and C. K. Yap, The format model: A theory of database organization, *J. Assoc. Comput. Machin.* **31** (1984), 518–537.

[M] A. S. Morozov, Functional trees and automorphisms of models, *Algebra and Logic* **32** (1993), 28–38.

[NR] A Nerode and J. Remmel, A survey of lattices of R. E. substructures, *in* "Recursion Theory" (A. Nerode and R. A. Shore, Eds.), Proc. Symposa in Pure Math., Vol. 42, pp. 323–375, Amer. Math. Soc., Providence, RI, 1985.

[P] J. Paredaens, On the expressive power of the relational algebra, *Inform. Process. Lett.* **7** (1978), 107–111.

[Ra] R. Rado, Universal graphs and universal functions, *Acta Arith.* **9** (1964), 331–340.

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium