Domain-Independent Queries on Databases with External Functions

Dan Suciu*
AT&T Labs — Research

Abstract

We study queries over databases with external functions, from a language-independent perspective. The input and output types of the external functions can be atomic values, flat relations, nested relations, etc. We propose a new notion of data-independence for queries on databases with external functions, which extends naturally the notion of generic queries on relational databases without external functions. In contrast to previous such notions, ours can also be applied to queries expressed in query languages with iterations. Next, we propose two natural notions of computability for queries over databases with external functions, and prove that they are equivalent, under reasonable assumptions. Thus, our definition of computability is robust. Finally, based on this equivalence result, we give examples of complete query languages with external functions. A byproduct of the equivalence result is the fact that Relational Machines [AV91, AVV92] are complete on nested relations: they are known not to be complete on flat relations.

Keywords: generic query, domain-independent query, external function, computable query, active domain, accessible domain.

1 Introduction

Database functionalities are important both in the context of traditional relational databases, and in that of object-oriented databases. The object-oriented database management system O_2 [Deu90] allows the query language to invoke any method written in the programming language C. An O_2 query can be parsed, compiled, optimized, and executed, without knowing what the external function does. At execution time, the external function is *called*, as an ordinary library function. In an object-oriented system like O_2, the set of user-defined methods can vary for a given database, as new methods are inserted, or old methods are deleted from the database. SQL3 [Dig93] offers a more limited approach, in which the number of user-written functions is fixed a priori, and is independent of the database[1]: we call these user-written functions *external functions*. While the practical aspects

*This work was done while the author was at the University of Pennsylvania and was partially supported by NSF grant CCR-90-57570, and by an IRCS fellowship.

[1]See 4.29: "A <routine> is either a SQL <routine> or an external <routine>".

of integrating external functions in query languages is well understood, a language-independent study of queries in the presence of external functions has received less attention. In this paper we make such a study, in the framework of a fixed number of external functions. We believe that most of our techniques will carry over smoothly to the object-oriented setting.

Chandra and Harel [CH80] study queries (without external functions) from a language-independent perspective. They give a precise characterization of the "reasonable" queries, namely as those which map isomorphic databases into isomorphic results. This definition emphasizes that queries are data-independent, i.e. the only observation they can make about the individual values in the database is their equality or inequality. Chandra and Harel called this condition the *consistency criterion*, but we will follow terminology which later became standard and call such queries *generic*. They also discuss the case of databases with *interpreted* functions: unlike our external functions, these are fixed functions over a fixed domain.

Giving a precise, language-independent characterization of the "reasonable" queries is important, because it offers us an upper bound on what a query language may express: namely each query expressed in a reasonable query language will be generic and Turing-computable. Chandra and Harel define a query language to be *complete* if it can express all generic and Turing-computable queries. Moreover, they give an example of a complete query language, QL.

In this paper we investigate how the language-independent characterization of generic queries carries over to databases with external functions. An example of a "reasonable" (i.e. data-independent) query in this context may be: "return the set of all employees of a company, with their salary x replaced with $P(x)$"; here P is an external function, implementing the company's salary raise policy. P should be thought of as a library function, written in a general-purpose programming language like C. This query is no longer generic as described above. Indeed one can prove that generic queries do not "invent new values", that is all atomic values in the query's result must have been in the database too. Obviously our query does invent new values, as it computes new salaries. But it is arguably still "reasonable", in the sense that the only observations it makes about the individual values in the database are their equality, inequality, and applications of the external function P.

We propose a definition for "reasonable" queries in the presence of external functions (Definition 3.2), and call such queries *externalq-function domain independent.* The inputs and outputs of the external functions can be scalar values, relations, nested relations, etc. We show that all queries expressed in reasonable query languages with external functions are external-function domain independent.

Next we address the issue of computability of queries with external functions. We propose two natural definitions for *computable queries with external functions.* The first definition assumes that external functions are computable themselves, and, hence, can be finitely encoded: then we say that a query is computable if there exists some Turing Machine which, when given an encoding of the input database and encodings of the external functions, computes an encoding of the output of the query. The second definition relies on a model of computation which is closer to the way external functions are handled in real database systems, namely as library functions. To do that, we essentially extend the Turing Machine with *oracles*, i.e. devices which can compute $P(x)$, for some external function

P and some input x. Then a query with external functions is computable if it can be computed by such an extended machine. We prove that, when the external functions are totally defined, then the two definitions of computability coincide (Theorem 6.2). We believe this result to be evidence for the robustness of the notion of computability.

Finally we define a query language with external functions to be *complete* if it can express exactly those queries which are both external-function domain independent and computable. We give examples of complete query languages with external functions.

Subtle differences separate our two notions of computability when the external functions are partial: the first definition allows *parallel* execution of the external functions, while the second definition restricts queries to *sequential* execution of the external functions, see Example 6.6 and Proposition 6.5.

Abiteboul and Beeri [AB88], and Escobar-Molano, Hull, and Jacobs [EMHJ93] propose two related language-independent definitions for "reasonable" queries with external functions. Their main purpose was to identify which of the queries expressible in certain logic-based query languages with external functions were "reasonable". Both definitions work well in conjunction with query languages without iterations: they fail however for languages with some forms of iterations, like fixpoints, or *while*, etc, as we show in Example 4.3. We explain here the connection between our notion of external-function domain independence and the definitions given in [AB88, EMHJ93], see Theorem 3.12. Computability of queries with external functions is not addressed in [AB88, EMHJ93], and no previous attempt has been made to define complete query languages with external functions.

Abiteboul and Vianu [AV91] introduce the notion of *loose Generic Machine*, later simplified to *Relational Machines* in [AVV92]. Their purpose was to define order-independent computations on databases. In order to compute a query with a Turing Machine, we first have to encode the input database; thus we introduce artificially an order on that database. By contrast, in a Relational Machine the input database is not encoded, but stored in *relational registers*. At each step, the machine can perform either a traditional Turing Machine move, affecting the tape and the state, or a first-order computation on the relational registers, affecting one of its relational registers: the latter computation are obviously order-independent. It turns out that Relational Machines are not complete: in particular they cannot compute the parity of an input set [AV91].

We extend the Relational Machines in a natural way to databases with nested relations and external functions, by replacing the relational registers with nested relational registers, and the first-order computations on the registers with computations in the Nested Relational Algebra with external functions. We call these machines *Relational Machines for Complex Objects*. In particular, a Relational Machine for Complex Objects may apply in one step an external function P on the value(s) in one of its registers: this can be viewed as an oracle inquire. Our second definition of computability is based on Relational Machines for Complex Objects. Our result (Theorem 6.2), stating that the two notions of computability coincide, implies that, by extending the Relational Machines from flat relations to nested relations, we obtain completeness.

Abiteboul, Papadimitriou, and Vianu [APV94] also achieve completeness by using a different extension of the Relational Machines, namely with *reflections*. A Reflective Relational Machine has the ability to dynamically create queries, and to answer them in constant time. This technique is orthogonal to our way of obtaining completeness, namely by extending flat relations to nested relations. Parallelism arises in Reflective Relational Machines from their ability to compute any first-order query in one parallel step: as a consequence, interesting connections to parallel complexity classes are proven in [APV94]. Here we are concerned with a different kind of parallelism which arises in conjunction with queries over databases with external functions: the ability of a query to initiate the computation of several external functions in parallel, and to wait until *one* of them terminates. Relational Machines for Complex Objects do not have this ability, however Turing Machines expecting a finite encoding of the external functions do. This observation allows us to prove that our two notions of computable queries over databases with external functions differ, when the external functions are partial.

The first description of a complete query language can be found in [CH80]: it is a dynamically typed language, in which an integer n can be encoded as some set of tuples of width n. Other complete query languages use different tools to achieve completeness: e.g. object inventions in [AK89], and untyped sets in [HS89]. Our examples of complete query languages with external functions are illustrations of how these known techniques can be extended to obtain completeness in the presence of external functions.

Our queries over databases with external functions are related to *higher type computability*. A number-theoretic function of type $\mathbb{N} \to \mathbb{N}$ is a *type 1 function*; a function $(\mathbb{N}^{\mathbb{N}})^k \times \mathbb{N}^l \to \mathbb{N}$ is a *type 2 function* [Con73, KC91]. Functions of type level n are those which expect as inputs functions of type level $\leq n-1$ [Bel94]. From this perspective, our queries on databases with external functions are of type 2, in that they take as inputs scalar functions. Our emphasis however is on "data-independence", an issue that is not present in the theory of higher-type computability, since the object of discourse there (the natural numbers) are fully interpreted.

Hirst and Harel [HH93] consider *recursive databases*, i.e. databases over a countable domain in which each relation, considered as a set of tuples, is recursive. We could view each external function of a database as infinite binary relations, by considering its graph. Still, our notions of external-function domain independence and computability are totally different from those of genericity and computability of [HH93]. The first reason is because the graphs of recursive functions are in general not recursive, but recursively enumerable [HR87]. Even if we restrict the external functions of a database to having recursive graphs, there is a deeper reason for which their framework differs from ours. Namely, Hirst and Harel observe (this is a consequence of Corollary 1 [HH93]) that, for a fixed type, there are only finitely many recursive generic queries of that type[2]. By contrast there are infinitely many computable, external-function domain independent queries for a given type[3].

[2]Thus, their notion of recursive generic query, when applied to finite relations, does not coincide with the traditional notion of computable, generic query. This is a significant departure from the case of finite databases.

[3]For example let P be an external function, and consider two constants a, b. Then for every $k \geq 0$, the boolean query $Q_k = (P^{(k)}(a) = b)$, with Q_k undefined when $\exists i \leq k.(P^{(i)}(a)$ is undefined), is computable and external-function domain independent. Obviously all queries $Q_0, Q_1, Q_2, \ldots$ are distinct.

Section 2 contains background an motivation: we review first some basic notions, like genericity, domain independence, computability, complex objects, etc., then define databases with external functions. We show, through an example, why the classical notions of genericity and domain independence fail in this case. Section 3 introduces the new concept of external-function domain independent query: Subsection 3.1 contains the definitions, Subsection 3.2 explains the intuition behind this concept, while Subsection 3.3 establishes the relationships between external-function domain independence and the embedded domain independence of [EMHJ93]. Section 4 reviews the Nested Relational Algebra with external functions, an algebraic query language for complex objects, and proves that each query expressed in this language is indeed external-function domain independent: this result remains true even for extension of the Nested Relation Algebra with various forms of iterations. Section 5 introduces the two concepts of computability: the computability notion based on Turing Machines and encodings of the external functions is presented in Subsection 5.1, while the computability notion based on extensions of Relational Machines is introduced in Subsection 5.2. We prove in Section 6 that they coincide, on databases with external functions which are totally defined. We give examples of complete query languages with external functions in Section 7.

2 Background and Motivation

Recall from [CH80] that a database instance is $\mathcal{D} = (D; R_1, \ldots, R_k)$, where $R_i \subseteq D^{a_i}$. Here D is called *the domain* of $\mathcal{D}$, while $R_1, \ldots, R_k$ are its *relations*, and are required to be finite. The numbers $a_1, \ldots, a_k$ are called the *arities*[4] of $R_1, \ldots, R_k$. We say that $(a_1, \ldots, a_k)$ is the *type* of the database $\mathcal{D}$. A *database query* of type $(a_1, \ldots, a_k) \to b$ is a partial function F mapping any database instance $\mathcal{D}$ of type $(a_1, \ldots, a_k)$ to some finite relation $F(\mathcal{D}) \subseteq D^b$.

More, it is usually required that the query be be *generic*, *domain independent* and *computable.*

Definition 2.1 *[CH80, HS89] A database query F is* **generic** *iff for any database instance $\mathcal{D} = (D; R_1, \ldots, R_k)$ and any isomorphism $\lambda : D \to D'$, $F(D'; \lambda(R_1), \ldots, \lambda(R_k)) = \lambda(F(D; R_1, \ldots, R_k))$*[5]*.*

A database query F is **domain independent** *iff for any database instance $\mathcal{D} = (D; R_1, \ldots, R_k)$ and for any set $D' \supseteq D$, $F(D; R_1, \ldots, R_k) = F(D'; R_1, \ldots, R_k)$.*

A database query F is **computable** *iff there exists a Turing Machine which, when given an encoding of a database instance $\mathcal{D} = (D; R_1, \ldots, R_k)$ on its tape, halts iff $F(\mathcal{D})$ is defined, and in this case will leave an encoding of $F(\mathcal{D})$ on its tape.*

Genericity and domain independence are independent conditions. They can be combined into a single condition:

[4] Called *ranks* in [CH80].

[5] Throughout the paper $E = E'$ means that either both expressions E, E' are undefined, or both are defined and equal.

Fact 2.2 *A query F is generic and domain independent iff for any two database instances $\mathcal{D} = (D; R_1, \ldots, R_k)$ and $\mathcal{D}' = (D'; R'_1, \ldots, R'_k)$ and for any injective function $\lambda : D \to D'$ for which $\lambda(R_1) = R'_1, \ldots, \lambda(R_k) = R'_k$, we have $F(\mathcal{D}') = \lambda(F(\mathcal{D}))$.*

The use of external functions in databases leads us naturally to structures whose domain is no longer the active domain of the database but an infinite one containing the active domain. Indeed, most of the external functions we will consider in this paper, like $+$, *succ*, *make_object*, etc. have infinite domains and codomains. Therefore we consider database instances with an *infinite* domain D (but still with finite relations $R_1, \ldots, R_k$), which is contrary to the traditional view that database instances have *finite* domains. However, because database queries are required to be domain independent, this is not a significant departure from the case with finite domains.

We want to allow external functions with arbitrary types of inputs and outputs, i.e. scalars or relations. For example an external function with one input could have (1) a scalar input and a scalar output, or (2) a scalar input and a relation as an output, or (3) a relation as an input and a scalar as an output, or (4) relations both as inputs and outputs. Instead of considering several cases we adopt a uniform approach by working in the context of *complex objects*, and replacing first order structures with higher order structures. The definitions below are consistent with [HS89, GV91, PG92, BBW92, BNTW95].

Definition 2.3 *We define* **complex object types** *by the grammar $t ::= d \mid t \times \ldots \times t \mid \{t\}$. (Here d is a special symbol.) For any set D and type t we define $dom(t, D)$ to be:*

$$\begin{aligned} dom(d, D) &\stackrel{\text{def}}{=} D \\ dom(t_1 \times \ldots \times t_n, D) &\stackrel{\text{def}}{=} dom(t_1, D) \times \ldots \times dom(t_n, D) \\ dom(\{t\}, D) &\stackrel{\text{def}}{=} \mathcal{P}_{fin}(dom(t, D)) \end{aligned}$$

The empty product (obtained by taking $n = 0$ in $t_1 \times \ldots \times t_n$) is denoted with *unit*; for any D, $dom(unit, D) = \{\langle\rangle\}$. Also, we abbreviate $\underbrace{t \times \ldots \times t}_{n \text{ times}}$ with t^n.

Definition 2.4 *A* **database schema** *is $\sigma = (t_1, \ldots, t_k)$, while a* **database instance** *over σ is $\mathcal{D} = (D; R_1, \ldots, R_k)$, with $R_i \in dom(\{t_i\}, D)$, for $i = 1, k$.*

All the results in this paper hold also for multi-sorted databases (with more than one base type, e.g. $\mathbb{Z}, string, \mathbb{R}, \ldots$), but in order to keep our formalism simple, we shall restrict ourselves in the sequel to only one base type D. To accommodate values of different base types, we may assume $D = \mathbb{Z} \cup string \cup \mathbb{R} \cup \ldots$

Definition 2.5 *For any schema σ and type t, a query F of type $\sigma \to t$ is a partial function mapping a database instance over σ, $\mathcal{D} = (D; R_1, \ldots, R_k)$ into $F(\mathcal{D}) \in dom(\{t\}, D)$.*

In this paper we consider **databases with external functions**, by augmenting database instances with a number of **external functions** $P_1, \ldots, P_l$. That is a database instance now becomes $\mathcal{D} = (D; P_1, \ldots, P_l; R_k, \ldots, R_k)$, where $R_1, \ldots, R_k$ are as before, while $P_1, \ldots, P_l$ are partial functions "over D". In their simplest form, the external external functions are *scalar*, i.e. of type $D^n \to D$, as in [EMHJ93], but we allow external functions of any types, i.e. $P_j : dom(u_j, D) \to dom(v_j, D)$, where u_j and v_j are arbitrary types called the **domain** and **codomain** of P_j.

Definition 2.6 *A* **database schema** *with external functions is* $\sigma = (u_1 \to v_1, \ldots, u_l \to v_l; t_1, \ldots, t_k)$. *A* **database instance** *over* σ *is* $\mathcal{D} = (D; P_1, \ldots, P_l; R_k, \ldots, R_k)$, *where* $R_i \in dom(\{t_i\}, D)$, $i = 1, k$, *and* $P_j : dom(u_j, D) \to dom(v_j, D)$ *is a partial function.* $\mathcal{D}$ *is called* **total** *iff all functions* P_j, $j = 1, l$ *are total, otherwise it is called* **partial**.

The **active domain** of a database instance $\mathcal{D}$ is the set of all atomic values mentioned in $R_1, \ldots, R_k$, i.e. the smallest set $D_0 \subseteq D$ for which $R_1 \in dom(\{t_1\}, D_0), \ldots, R_k \in dom(\{t_k\}, D_0)$. Note that the active domain is always finite, although D may be infinite (because all relations $R_1, \ldots, R_k$ are required to be finite). Unless otherwise specified, we shall assume throughout the paper that the database schemas and instances are *with external functions.*

Example 2.7 *Consider the schema* $\sigma = (d \times d \to d, \{d\} \to d; d^5)$ *(recall* d^5 *means* $d \times d \times d \times d \times d$*). As an example for a database instance over* σ*, consider* $\mathcal{D} = (D; raise\_salary, new\_ID; R)$ *where:*

$D = Strings \cup Int \quad , \quad R \subseteq D^5$

$R =$

ID	*Name*	*Dept*	*Manager*	*Salary*
15023	*Smith*	*Customer Serv.*	*Lucy*	20,000
15024	*Lucy*	*Customer Serv.*	*Dale*	40,000
15046	*Jones*	*Sales*	*Dale*	30,000
15028	*Dale*	*Management*	*Brian*	50,000

The external functions raise_salary and new_ID encapsulate domain-specific knowledge of the particular company. Thus $raise\_salary(n, s)$ *will return a new salary for the employee with name* n*, assuming its old salary is* s*. This may depend on the company's policy, even on the particular contract signed by that employee. Typically such a function is written in a general-purpose programming language – like C – and is part of the database instance. Note that* $raise\_salary(n, s)$ *may be undefined for certain values of* n *and* s*, e.g. when* n *is an integer, or* s *is a string. In a similar fashion,* $new\_ID(S)$ *will return a new identifier not present in* S*, i.e.* $new\_ID(S) \notin S$*. Most likely, it encapsulates the company's policy of assigning new id's to employees.*

Consider the following queries: "compute new salaries for each employee as follows: each employee receives one raise, except for the managers who receive two raises". Here $F : \sigma \to d^2$*. On our particular instance,* F *would return:*

$F(\mathcal{D}) =$

Name	*New_Salary*
Smith	$raise\_salary(\text{"}Smith'', 20,000)$
Lucy	$raise\_salary(\text{"}Lucy'', raise\_salary(\text{"}Lucy'', 40,000))$
Jones	$raise\_salary(\text{"}Jones'', 30,000)$
Dale	$raise\_salary(\text{"}Dale'', raise\_salary(\text{"}Dale'', 50,000))$

For another example, consider: "assuming there is only one manager not recorded as an employee, return its name and a new id for him". Here $G : \sigma \to d^2$*:*

$G(\mathcal{D}) =$

Name	*New_ID*
Brian	$new\_ID(\{15023, 15024, 15046, 15028\})$

Obviously a query over the database of Example 2.7 may not necessarily be domain independent in the traditional sense, because it has the ability of generating new atomic values by calling the external functions. The first goal of this paper is to investigate the notion of domain independence of queries with external functions.

Unlike the *interpreted functions* of [CH80] which are fixed, our external functions are closer in spirit to those in [AB88, EMHJ93]. In these papers, the authors define two related notions of domain independence: the *bounded-depth domain independence* in [AB88], and the *embedded domain independence* in [EMHJ93]. Strictly speaking, the embedded domain independence implies that of bounded-depth domain independence, but they rely on the same idea. Both notions are used only in conjunction with query languages without iterative queries, and, as we show in this paper, fail when extended to languages with iterations. We describe a fixpoint query which is not embedded domain independent in Example 4.3. In this paper we introduce a new notion, called *external-function domain independence*, which is more general than the embedded domain independence, and show that all queries expressed in query languages with iterations (fixpoints, loops, structural recursions, etc.) are external-function domain independent.

Our second goal in this paper is to investigate the computability of queries in the presence of external functions. The complexity and the computability of number-theoretic functions of higher type have been considered before [HR87, Con73, KC91], but computable *queries* in the presence of external functions have not been considered previously. We propose here two natural definitions for computability of queries with external functions, and prove that they are equivalent under reasonable assumptions.

Previous work [Top87, AB88, EMHJ93] has been concerned with identifying recursive sets of first order formulas, which define domain independent queries. We do not address this problem here, but consider only algebraic query languages instead, where all queries are domain independent. We believe that the notion of *embedded allowed formulas* from [EMHJ93] could be extended to a higher order logic with fixpoints, such that all "embedded allowed" formulas define an ef-domain independent, computable query, but this has to be addressed in future work.

3 External-function Domain Independence

3.1 Definitions

Allowing the external functions to be partial, as opposed to total, constitues a key technical tool in our definition of domain independent queries over databases with external functions. To test whether a query F is domain independent when applied to some database instance $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$, we would like to probe it with databases with smaller domains, $\mathcal{D}_0 = (D_0; P_{01}, \ldots, P_{0l}; R_1, \ldots, R_k)$, where $D_0 \subseteq D$ contains the active domain of $\mathcal{D}$, and $P_{01}, \ldots, P_{0l}$ are the restrictions of $P_1, \ldots, P_l$. If D_0 is chosen arbitrarily, then $P_{01}, \ldots, P_{0l}$ will be partial, in general.

We will define a *database morphism* $\lambda : \mathcal{D} \to \mathcal{D}'$ to be a partial, injective function $\lambda : D \to D'$ between the domains of two databases, which "preserves the structure" of theses databases, in a sense to be made precise. First notice that any partial function $\lambda : D \to D'$ can be lifted from the base type to partial functions at any type t, $\lambda_t : dom(t, D) \to dom(t, D')$, as follows:

$$\begin{aligned}
\lambda_d(x) &\stackrel{\text{def}}{=} \lambda(x) \\
\lambda_{t_1 \times \ldots \times t_n}\langle x_1, \ldots, x_n\rangle &\stackrel{\text{def}}{=} \langle \lambda_{t_1}(x_1), \ldots, \lambda_{t_n}(x_n)\rangle \\
\lambda_{\{t\}}(\{x_1, \ldots, x_n\}) &\stackrel{\text{def}}{=} \{\lambda_t(x_1), \ldots, \lambda_t(x_n)\}
\end{aligned}$$

In all cases, $\lambda_t(x)$ is undefined whenever one of the subexpressions on the right hand side is undefined. We abbreviate λ_t with λ, whenever the type t is implicit in the context.

Definition 3.1 *Let σ be some database schema, and $\mathcal{D}, \mathcal{D}'$ two database instances over σ: $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$, $\mathcal{D}' = (D'; P'_1, \ldots, P'_l; R'_1, \ldots, R'_k)$. A* **database morphism** $\lambda : \mathcal{D} \to \mathcal{D}'$ *is a partial injective function $\lambda : D \to D'$, such that the following two conditions hold:*

1. *For every i, $\lambda(R_i)$ is defined and $\lambda(R_i) = R'_i$.*
2. *For any*[6] *$x \in dom(u_j, D)$ for which both $P_j(x)$ and $x' \stackrel{\text{def}}{=} \lambda(x)$ are defined, $P'_j(x')$ and $\lambda(P_j(x))$ are defined too, and they are equal: $P'_j(x') = \lambda(P_j(x))$.*

[6]Recall that $P_j : dom(u_j, D) \to dom(v_j, D)$.

We represent condition 2 above as the following diagram:

$$\begin{array}{ccc} x & \xrightarrow{P_j} & P_j(x) \\ \lambda\downarrow & & \downarrow\lambda \\ \lambda(x) & \xrightarrow{P'_j} & P'_j(\lambda(x)) = \lambda(P_j(x)) \end{array}$$

Whenever both left and upper arrows are defined, then the bottom and right arrows are defined too, and the diagram commutes.

Alternatively, let us write $e_1 \sqsubseteq e_2$, whenever expression e_1 is undefined, or $e_1 = e_2$. For two functions f_1, f_2, let $f_1 \sqsubseteq f_2$ mean that $\forall x$, $f_1(x) \sqsubseteq f_2(x)$, or, equivalently, $graph(f_1) \subseteq graph(f_2)$. Then, from the diagram above we can see that λ is a database morphism iff $\lambda(R_i) = R'_i$ for all i, and $P_j \circ \lambda^{-1} \sqsubseteq \lambda^{-1} \circ P'_j$ for all j (to be precise, $P_j \circ \lambda_{u_j}^{-1} \sqsubseteq \lambda_{v_j}^{-1} \circ P'_j$, but recall that we drop the type t from λ_t).

Note that the totally undefined function $\lambda : D \to D'$ is not a database morphism. Indeed λ has to be defined at least on the active domain of D: more, we will show that it has to be defined on every atomic value "accessible" from the active domain by repeated application of the external functions, see Lemma 3.6.

Consider the particular case in which the schema σ has no relation symbols (i.e. $k = 0$) and all external functions are scalar, i.e. $\sigma = (d^{n_1} \to d, d^{n_2} \to d, \ldots, d^{n_l} \to d;)$. Then a database instance $\mathcal{D} = (D; P_1, \ldots, P_l;)$ coincides with a *partial algebra* [Gra80]. Moreover the total database morphisms $\lambda : \mathcal{D} \to \mathcal{D}'$ are precisely those functions which are injective and for which λ^{-1} is a *homomorphism of partial algebras* of [Gra80].

Definition 3.2 *Let σ be a database schema and t some type. A* **database query** *of type $\sigma \to t$ is a partial function F mapping a database instance over σ, $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$, to $F(\mathcal{D}) \in dom(\{t\}, D)$. F is* **external-function domain independent**, *or, shortly,* **ef-domain independent**, *iff for every database morphism $\lambda : \mathcal{D} \to \mathcal{D}'$, if $F(\mathcal{D})$ is defined, then so are $\lambda(F(\mathcal{D}))$ and $F(\mathcal{D}')$, and they are equal: $\lambda(F(\mathcal{D})) = F(\mathcal{D}')$.*

We represent this condition as in the following diagram:

$$\begin{array}{ccc} \mathcal{D} & \xrightarrow{F} & F(\mathcal{D}) \\ \lambda\downarrow & & \downarrow\lambda \\ \mathcal{D}' & \xrightarrow{F} & \lambda(F(\mathcal{D})) = F(\mathcal{D}') \end{array}$$

As for the case of a database morphism, whenever the upper arrow is defined[7], then the bottom and right arrows are defined too, and the diagram commutes. From here we can infer again that F is a

[7]The left arrow is always defined.

ef-domain independent query iff $F \circ \lambda^{-1} \sqsubseteq \lambda^{-1} \circ F$, for every database morphism λ. Also, note that $\lambda(F(\mathcal{D})) \sqsubseteq F(\mathcal{D}')$.

This notion generalizes the classic notions of genericity and domain independence (Definition 2.1) to databases with external functions. More, ef-domain independence extends the C-genericity of [HS89]. The latter is intended to characterize queries expressed in languages which can refer to some constants C in the domain. A query is called C-generic iff it is invariant w.r.t. all isomorphisms which leave the constants in C unchanged. Although constants are not explicitly present in our definition of a database schema, they can be represented as functions of type *unit* $\rightarrow d$.

Fact 3.3 *Let $\sigma = (; t_1, \ldots, t_k)$ (i.e. there are no external functions). Then a query F is ef-domain independent iff it is generic and domain independent in the sense of Definition 2.1.*

Moreover, when σ contains only atomic constants (i.e. functions of type unit $\rightarrow d$), then a query is ef-domain independent iff it is C-generic [HS89] and domain independent.

Proof. We use Fact 2.2. Suppose F is ef-domain independent, and let $\lambda : D \rightarrow D'$ be as in Fact 2.2. Then λ is a database morphism and, by Definition 3.2, we have either (1) both $\lambda(F(\mathcal{D}))$ and $F(\mathcal{D}')$ are defined and equal, or (2) both $\lambda(F(\mathcal{D}))$ and $F(\mathcal{D}')$ are undefined, or (3) $F(\mathcal{D})$ is undefined and $F(\mathcal{D}')$ is defined. In cases (1) and (2) we are done. To refute case (3), consider $\lambda^{-1} : D' \rightarrow D$. It is still a database morphism so, since $F(\mathcal{D}')$ is defined, so must be $F(\mathcal{D})$.

For the converse, assume F satisfies the conditions in Fact 2.2, and let $\lambda : \mathcal{D} \rightarrow \mathcal{D}'$ be a database morphisms, where $\mathcal{D} = (D; ; R_1, \ldots, R_k)$. Let $D_0 \subseteq D$ be the active domain of $\mathcal{D}$, and let $\mathcal{D}_0 \stackrel{\text{def}}{=} (D_0; ; R_1, \ldots, R_k)$. Assume $F(\mathcal{D})$ is defined. Since F is domain independent we have $F(\mathcal{D}) = F(\mathcal{D}_0)$. Moreover λ is totally defined on the active domain, so its restriction $\lambda_0 : D_0 \rightarrow D'$ satisfies the conditions of Fact 2.2. Hence $\lambda_0(F(\mathcal{D}_0))$ is defined too, and $\lambda(F(\mathcal{D}_0)) = \lambda_0(F(\mathcal{D}_0)) = F(\mathcal{D}')$.

The relationship with C-genericity is obtained similarly. $\square$

3.2 Intuition

The definition of external-function domain independence captures in a single condition three rather independent, and more intuitive properties of queries. We will show in this subsection that a query is external-function domain independent iff it is simultaneously (1) *monotone*, i.e. more defined on more defined databases, and (2) *forward-looking*, i.e. it only applies the external functions forwardly, like in $P(x), P(P(x)), \ldots$, and not like in $P^{-1}(x)$, and (3) isomorphism preserving, i.e. maps isomorphic database instances into isomorphic results.

First we define an "approximation" of a database instance. We say that $\mathcal{D}$ **approximates** $\mathcal{D}'$, written $\mathcal{D} \sqsubseteq \mathcal{D}'$, iff $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_l)$, $\mathcal{D}' = (D'; P_1', \ldots, P_l'; R_1, \ldots, R_l)$ (i.e. they have the same relations), $D \subseteq D'$, and $P_j \sqsubseteq P_j'$, for all $j = 1, l$. In this case the inclusion $\lambda : \mathcal{D} \rightarrow \mathcal{D}'$ is a total database morphism.

The term *approximation* is borrowed from denotational semantics [Plo81, Gun92]. There a function f approximates another function f' if $graph(f) \subseteq graph(f')$. We emphasize that in our context the term "approximation" refers only to the domain and the external functions, and not to the relations. For a schema σ without external functions (i.e. $l = 0$), a database $\mathcal{D}$ approximates another database $\mathcal{D}'$ iff $D \subseteq D'$ and $R_i = R_i'$, for $i = 1, k$.

Definition 3.4 *A query F is* **monotone** *if whenever $\mathcal{D} \sqsubseteq \mathcal{D}'$, we have $F(\mathcal{D}) \sqsubseteq F(\mathcal{D}')$.*

Next we will define a query to be *forward-looking* if it uses the external functions only in a restricted way, namely only as subroutines. That is, a forward-looking query may start by applying the external functions $P_1, \ldots, P_l$ to inputs built from the active domain of the database. In the process it may construct new atomic values, and it may use these to build new inputs for the external functions, etc. A forward-looking query however is not allowed to compute $P^{-1}(x)$, or to ask global questions about the external functions, like "is P injective ?".

Let $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$ be a database. For every $n \geq 0$ we define $term^n(\mathcal{D}) \subseteq D$ inductively by n:

$$\begin{aligned} term^0(\mathcal{D}) &\stackrel{\text{def}}{=} atoms(R_1) \cup \ldots \cup atoms(R_k) \\ term^{n+1}(\mathcal{D}) &\stackrel{\text{def}}{=} term^n(\mathcal{D}) \cup \{atoms(P_j(x)) \mid x \in dom(u_j, term^n(\mathcal{D})), j = 1, l\} \end{aligned}$$

where $atoms(x)$ is defined to be all values in D mentioned in the complex object x. Note that $term^0(\mathcal{D})$ is the active domain of $\mathcal{D}$.

The definition of $term^n(\mathcal{D})$ is the same as in [EMHJ93], where it is introduced in order to define embedded domain independence, reviewed here in Subsection 3.3.

Definition 3.5 *The* **accessible domain** $\bar{D}$ *of some databases $\mathcal{D}$ is $\bar{D} \stackrel{\text{def}}{=} \bigcup_{n\geq 0} term^n(\mathcal{D})$. The* **accessible approximation** *of $\mathcal{D}$ is $\bar{\mathcal{D}} \stackrel{\text{def}}{=} (\bar{D}; \bar{P}_1, \ldots, \bar{P}_l; R_1, \ldots, R_k)$, where $\bar{P}_j$ is the restriction of P_j to $\bar{D}$, for $j = 1, l$.*

Obviously the accessible domain extends the active domain.

The accessible approximation is a very special approximation of a database $\mathcal{D}$. Namely it has the property that the inverse $\rho \stackrel{\text{def}}{=} \lambda^{-1}$ of the inclusion database morphisms λ, with $\rho : \mathcal{D} \to \bar{\mathcal{D}}$, is a database morphism too. To see that, let $x \in dom(u_j, D)$, and let $\rho(x)$, $P_j(x)$ be defined. Since $\rho(x)$ is defined, we have $x \in dom(u_j, \bar{D}) \subseteq dom(u_j, D)$, and $\rho(x) = x$. Since $P_j(x)$ is defined, obviously $\bar{P}_j(x)$ is defined too. We have to prove that $\rho(P_j(x))$ is also defined, which is in general not true when ρ is the inverse of some arbitrary approximation. But here, since $x \in (u_j, \bar{D})$, we have $atoms(x) \subseteq term^n(\mathcal{D})$ for some $n \geq 0$, hence $atoms(P_j(x)) \subseteq term^{n+1}(\mathcal{D})$, so $P_j(x) \in dom(v_j, \bar{D})$, and therefore $\rho(P_j(x))$ is defined too.

Lemma 3.6 *Let $\lambda : \mathcal{D} \to \mathcal{D}'$ be a database morphism. Then λ is totally defined on $\bar{D}$, the accessible domain of $\mathcal{D}$.*

Proof. We prove by induction on n that λ is defined on $term^n(\mathcal{D})$. For $n = 0$ this is obvious, since λ is defined on the active domain. For the induction step, observe that any atomic value in $term^{n+1}(\mathcal{D})$ is in some $atoms(P_j(x))$, with $atoms(x) \subseteq term^n(\mathcal{D})$. By induction hypothesis $x' = \lambda(x)$ is defined, hence $\lambda(P_j(x))$ must be defined too. □

Definition 3.7 *A database query is* **forward-looking** *iff for any database instance $\mathcal{D}$, if $F(\mathcal{D})$ is defined, then $F(\bar{\mathcal{D}})$ is defined too, where $\bar{\mathcal{D}}$ is the accessible approximation of $\mathcal{D}$.*

Finally we call a query **isomorphism preserving** iff for any database isomorphism[8] $\lambda : \mathcal{D} \to \mathcal{D}'$, we have $F(\mathcal{D}') = \lambda(F(\mathcal{D}))$.

Proposition 3.8 *A query F is ef-domain independent iff it is monotone, forward-looking, and isomorphism preserving.*

Proof. Let F be ef-domain independent; obviously it is isomorphism preserving. Let $\mathcal{D}$ be an approximation of $\mathcal{D}$', $\mathcal{D} \sqsubseteq \mathcal{D}'$, and $\lambda : \mathcal{D} \to \mathcal{D}'$ be the inclusion database morphisms. We have $F(\mathcal{D}) = \lambda(F(\mathcal{D})) \sqsubseteq F(\mathcal{D}')$, hence F is monotone. To prove that it is forward-looking, we consider the database morphism $\rho : \mathcal{D} \to \bar{\mathcal{D}}$, for some database instance $\mathcal{D}$. Then whenever $F(\mathcal{D})$ is defined, $F(\bar{\mathcal{D}})$ must be defined too.

Conversely, consider a monotone and forward-looking query F. Let $\lambda : \mathcal{D} \to \mathcal{D}'$ be a database morphism, and assume $F(\mathcal{D})$ is defined. Recall that λ is totally defined on $\bar{D}$, hence $\mathcal{D}'' \stackrel{\text{def}}{=} \lambda(\bar{\mathcal{D}})$ is an isomorphic image of the accessible approximation $\bar{\mathcal{D}}$: let $\bar{\lambda} : \bar{\mathcal{D}} \to \mathcal{D}''$ be the restriction of λ to the domain $\bar{D}$. Then we decompose λ in a canonical way as: $\mathcal{D} \xrightarrow{\rho} \bar{\mathcal{D}} \xrightarrow{\bar{\lambda}} \mathcal{D}'' \sqsubseteq \mathcal{D}'$. Since $F(\mathcal{D})$ is defined, so is $F(\bar{\mathcal{D}})$, and they are equal (because $\bar{\mathcal{D}}$ is an approximation of $\mathcal{D}$). Hence $F(\mathcal{D}'')$ is defined too, and equal with $\bar{\lambda}(F(\mathcal{D}))$, because $\bar{\lambda}$ is an isomorphism. Finally $F(\mathcal{D}')$ is defined and equal to $\bar{\lambda}(F(\mathcal{D})) = \lambda(F(\mathcal{D}))$. □

3.3 Connection with related work

Escobar-Molano, Hull, and Jacobs [EMHJ93] define *embedded domain independent queries*, in the context of relational databases and scalar external functions. We extend their notion straightforwardly to complex objects and external functions with arbitrary domains and codomains. For this, following [EMHJ93], we first say that two databases instances with the same relations $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_l)$ and $\mathcal{D}' = (D'; P_1', \ldots, P_l'; R_1, \ldots, R_k)$ *agree up to level* n iff (1) $term^{n+1}(\mathcal{D}) =$

[8] I.e. a total, bijective database morphism, whose inverse is also a database morphism.

$term^{n+1}(\mathcal{D}')$, and (2) for any j, P_j and P'_j agree on any input whose atoms are in $term^n(\mathcal{D})$, i.e. $\forall x \in dom(u_j, term^n(\mathcal{D}))$, $P_j(x) = P'_j(x)$. Then we call a query F **embedded domain independent at level** n, or **em-domain independent at level** n, if $F(\mathcal{D}) = F(\mathcal{D}')$ whenever $\mathcal{D}$ and $\mathcal{D}'$ agree to level n. Finally we say that a query F is **em-domain independent**, iff it is isomorphism-preserving and there is some n for which F is em-domain independent at level n.

Abiteboul and Beeri in [AB88] define *bounded-depth domain independence*, which coincides with that of em-domain independence over database instances whose external functions are closed under inverses. They need closure under inverses because they consider a logic-based query language, in which queries like $\{x \mid P(x) = y\}$ are allowed.

Intuitively, em-domain independence allows some query to repeatedly apply the external functions at most n times, for some n which is independent on the database instance $\mathcal{D}$. This condition is indeed satisfied by the queries expressed in languages without iterations, like those considered in [AB88, EMHJ93], but fails once an iterative construct (like fixpoints) is added to the language, see Example 4.3. For iterative queries, the number n of applications of the external functions is still finite, but may depend on the particular database instance. To overcome this limitation of em-domain independence, we strengthen it, by switching the quantifiers. We call a query F to be **strongly embedded domain independent** (**sem-domain independent**), iff it is isomorphism-preserving and for any database instance $\mathcal{D}$ for which $F(\mathcal{D})$ is defined, there is some n such that: for any other database instance $\mathcal{D}'$ which agrees with $\mathcal{D}$ up to level n, it is the case that $F(\mathcal{D}) = F(\mathcal{D}')$. Call n **the level** of F at $\mathcal{D}$. Obviously em-domain independence implies sem-domain independence. Note that when $F(\mathcal{D})$ is undefined there may not exist such an n (because F may loop inspecting the whole accessible domain $\bar{D}$).

In the rest of this subsection we will compare our notion of ef-domain independence to sem-domain independence. Sem-domain independent queries are forward looking, but the converse is not true, because forward-looking queries may use the entire accessible domain $\bar{D}$, and not just a finite approximation of it, $term^n$: we will show below that forward-looking and *continuous* queries are sem-domain independent (this is implicit in Theorem 3.12).

On the other hand sem-domain independent queries are not necessarily monotone. We will show that monotone sem-domain independent queries are ef-domain independent.

First we define continuous queries. For that we start by defining the least upper bound of a directed family of databases. Let $\mathcal{S}$ be a set of database instances of the same signature. We say that $\mathcal{S}$ is **directed** iff it is nonempty and $\forall \mathcal{D}_1, \mathcal{D}_2 \in \mathcal{S}$, $\exists \mathcal{D} \in \mathcal{S}$ such that $\mathcal{D}_1 \sqsubseteq \mathcal{D}$ and $\mathcal{D}_2 \sqsubseteq \mathcal{D}$. Note that all databases in $\mathcal{S}$ share the same relations $R_1, \ldots, R_k$, and differ only in their domain and their external functions. But external functions from any two such databases are "compatible", i.e. the union of their graphs is still a function. For a directed family of database instances $\mathcal{S}$ we define its **least upper bound** $\bigcup \mathcal{S} \stackrel{\text{def}}{=} (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$ where D is the union of the domains of all databases in $\mathcal{S}$, and the graph of P_j is the union of all graphs of the corresponding function of the databases in $\mathcal{S}$. Note that this is indeed the least upper bound, in the sense that $\forall \mathcal{D}'$, if $\mathcal{D} \sqsubseteq \mathcal{D}'$ for all $\mathcal{D} \in \mathcal{S}$, then $\bigcup \mathcal{S} \sqsubseteq \mathcal{D}'$.

Definition 3.9 *A query F is called* **continuous** *iff for any directed set of database instances $\mathcal{S}$, $\{F(\mathcal{D}) \mid \mathcal{D} \in \mathcal{S}\}$ exists and $F(\bigcup \mathcal{S}) = \bigcup\{F(\mathcal{D}) \mid \mathcal{D} \in \mathcal{S}\}$*

Here $\bigcup\{F(\mathcal{D}) \mid \mathcal{D} \in \mathcal{S}\}$ denotes the least upper bound of the relations $F(\mathcal{D})$ with $\mathcal{D} \in \mathcal{S}$. In short, it exists if and only if either $F(\mathcal{D})$ is undefined for all $\mathcal{D} \in \mathcal{S}$ or all values $F(\mathcal{D})$ which are defined are equal. In the latter case the least upper bound is equal to this unique value.

Proposition 3.10 *A query F is continuous iff for any database instance $\mathcal{D}$ for which $F(\mathcal{D})$ is defined, there exists some finite approximation $\mathcal{D}_0$ of it, i.e. D_0 is finite and $\mathcal{D}_0 \sqsubseteq \mathcal{D}$ such that $F(\mathcal{D}_0) = F(\mathcal{D})$.*

The proof uses standard arguments in the theory of continuous functions on algebraic cpo's, see e.g. [Plo81, Gun92]. We will use mostly the characterization of continuous functions of Proposition 3.10 throughout the paper.

Obviously all domain independent queries over databases without external functions are continuous, since it suffices to take D_0 to be the active domain, which is finite. We also have:

Proposition 3.11 *Any sem-domain independent query is continuous. Hence, any em-domain independent query is continuous too.*

Indeed, it suffices to take $D_0 = \mathit{term}^{n+1}(\mathcal{D})$, where n is the level of F at $\mathcal{D}$.

Now we can establish the relationship between our notion of ef-domain independence (Definition 3.2) and that of em-domain independence of [EMHJ93].

Theorem 3.12 *A query F is ef-domain independent and continuous iff it is sem-domain independent and monotone.*

Proof. First we show that any sem-domain independent query F is forward-looking. For some database instance $\mathcal{D}$, if $F(\mathcal{D})$ is undefined, then there is nothing to prove, so suppose $F(\mathcal{D})$ is defined and let n be the level of F at $\mathcal{D}$. Take $(\mathcal{D})^{(n)}$ be the database approximation of $\mathcal{D}$ defined by the domain $\mathit{term}^{n+1}(\mathcal{D})$. Then $F((\mathcal{D})^{(n)}) = F(\mathcal{D})$ because $(\mathcal{D})^{(n)}$ and $\mathcal{D}$ coincide up to level n. But the accessible approximation $\bar{\mathcal{D}}$ of $\mathcal{D}$ also coincides up to level n with $(\mathcal{D})^{(n)}$, hence $F((\mathcal{D})^{(n)}) = F(\bar{\mathcal{D}})$. So $F(\mathcal{D}) = F(\bar{\mathcal{D}})$, and in particular the latter is defined.

This implies that monotone sem-domain independent queries are ef-domain independent; we know from Proposition 3.11 that they are also continuous.

Conversely, let F be an ef-domain independent, continuous query; by proposition 3.8 it suffices to show that F is sem-domain independent. Let $\mathcal{D}$ be some database instance for which $F(\mathcal{D})$ is defined.

Because F is forward-looking, $F(\bar{\mathcal{D}})$ is defined too and equal to $F(\mathcal{D})$. Now we use the fact that F is continuous, and find some finite approximation $\mathcal{D}_0 \sqsubseteq \bar{\mathcal{D}}$ for which $F(\mathcal{D}_0) = F(\bar{\mathcal{D}})$. Pick some n for which the domain of $\mathcal{D}_0$ is included in $term^n(\mathcal{D})$ (this is possible because $\mathcal{D}_0$ is finite). It implies that $F(\mathcal{D}_0) = F(\mathcal{D}^{(n)})$, because $\mathcal{D}_0$ is an approximation of $\mathcal{D}^{(n)}$. We prove that the level of F at $\mathcal{D}$ is at most n. Indeed pick some other database $\mathcal{D}'$ which agrees with $\mathcal{D}$ up to level n. That means that $\mathcal{D}^{(n)}$ is an approximation of both $\mathcal{D}$ and $\mathcal{D}'$, and since $F(\mathcal{D}^{(n)})$ is defined, we have $F(\mathcal{D}) = F(\mathcal{D}')$. $\square$

Finally we show that the characterization of Theorem 3.12 is tight, i.e. that the additional conditions of continuity and monotonicity are necessary.

Example 3.13 *The following query F is ef-domain independent but not continuous. Consider the database schema $\sigma = (d \to d; d)$, and define $F : \sigma \to d$ to be:*

$$F(D; P; R) \stackrel{\text{def}}{=} \begin{cases} R & \textit{if the set } \{P^{(n)}(x) \mid x \in R, n \geq 0\} \textit{ is infinite} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Obviously F is not continuous. Moreover, it is easy to check that the query is monotone and forward-looking, hence ef-domain independent.

Example 3.14 *The following query F is em-domain independent (hence sem-domain independent) but not monotone. Consider the same schema $\sigma = (d \to d; d)$, and let F be the query:*

$$F(D; P; R) \stackrel{\text{def}}{=} \begin{cases} R & \textit{if } \forall x \in R,\ P(x) \textit{ is defined} \\ \emptyset & \textit{otherwise} \end{cases}$$

The obviously F is embedded domain independent at level 0, because it only inspects the active domain, but it is not monotone. To see that, take some database $(D; P; R)$ with $R \neq \emptyset$, in which P is totally defined, hence $F(D; P; R) = R$. Consider the approximation in which P is totally undefined: we have $F(D; \emptyset; R) = \emptyset$. It is defined, but not equal to $F(D; P; R)$.

We end this section with a brief comment on continuity, i.e. the property of a query to depend only on a finite approximation of the database. Certainly, we would expect all queries expressed in a query language with external functions to be continuous, because during any computation we can inspect only a finite fragment of a potentially infinite input. The fact that ef-domain independence does not enforce continuity seems worrisome. However we will see in Section 5 that all *computable* queries are continuous. Thus, in our setting, continuity is a consequence of computability, and not of domain-independence.

4 A Language

We will present in this section a database query language with external functions, and we will show that all queries expressed in this language are ef-domain independent. They remain ef-domain

$$\frac{p_j \in \Sigma}{p_j : u_j \to v_j} \qquad \frac{}{id^s : s \to s} \qquad \frac{f : r \to s \quad g : s \to t}{(g \circ f) : r \to t}$$

$$\frac{}{!^s : s \to unit} \qquad \frac{f_1 : s \to t_1 \quad \ldots \quad f_n : s \to t_n}{\langle f_1, \ldots, f_n \rangle : s \to t_1 \times \ldots \times t_n} \qquad \frac{}{\pi_i^{t_1,\ldots,t_n} : t_1 \times \ldots \times t_n \to t_i}(i = 1, n)$$

$$\frac{f : s \to t}{map(f) : \{s\} \to \{t\}} \qquad \frac{}{\eta^s : s \to \{s\}} \qquad \frac{}{\mu^s : \{\{s\}\} \to \{s\}} \qquad \frac{}{\rho_2^{s,t} : s \times \{t\} \to \{s \times t\}}$$

$$\frac{}{\emptyset^s : unit \to \{s\}} \qquad \frac{}{\cup^s : \{s\} \times \{s\} \to \{s\}} \qquad \frac{}{=: D \times D \to \{unit\}} \qquad \frac{}{not : \{unit\} \to \{unit\}}$$

Figure 1: The language $\mathcal{NRA}(\Sigma)$

independent even when we extend the language with various forms of iterations. By contrast, queries expressed with iterations need not be em-domain independent.

Let Σ be a **signature**, that is Σ is a set $\Sigma = \{p_1, \ldots, p_l\}$ of l symbols, and each symbol p_j has an associated *domain* u_j and *codomain* v_j, where u_j, v_j are types. To emphasize the domains and codomains, we will write the signature as $\Sigma = \{p_1 : u_1 \to v_1, \ldots, p_l : u_l \to v_l\}$, and call $p_1, \ldots, p_l$ *external function symbols*. We define the Nested Relational Algebra over Σ, $\mathcal{NRA}(\Sigma)$, following the formalism in [BNTW95], as an algebra of functions. The constructs are given in Figure 1.

For any database $\mathcal{D} = (D; P_1, \ldots, P_l;)$ (i.e. which has only external functions, no relations), every expression $f : s \to t$ in the language $\mathcal{NRA}(\Sigma)$ denotes a function $f_{\mathcal{D}} : dom(s, D) \to dom(t, D)$, as described shortly. We will drop the index $\mathcal{D}$ from $f_{\mathcal{D}}$, and write s, t instead of $dom(s, D)$, $dom(t, D)$, whenever the database $\mathcal{D}$ is implicit from the context.

Every expression $p_j : u_j \to v_j$ denotes the function P_j. id^s denotes the identity at type s, $g \circ f$ is function composition, $!^s : s \to unit$ is $!^s(x) \stackrel{\text{def}}{=} \langle\rangle$ for all $x \in s$, $\langle f_1, \ldots, f_n \rangle(x) \stackrel{\text{def}}{=} \langle f_1(x), \ldots, f_n(x) \rangle$, $\pi_i^{t_1,\ldots,t_n}$ is the i'th projection. Next, $map(f)(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} \{f(x_1), \ldots, f(x_n)\}$, $\eta^s(x) \stackrel{\text{def}}{=} \{x\}$, $\mu^s(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} x_1 \cup \ldots \cup x_n$. The function $\rho_2^{s,t}$ is the "monad strength" of [BNTW95], and compensates for the fact that in this presentation of $\mathcal{NRA}(\Sigma)$ we don't have free variables: $\rho_2^{s,t}\langle x, \{x_1, \ldots, x_n\} \rangle \stackrel{\text{def}}{=} \{\langle x, y_1 \rangle, \ldots, \langle x, y_n \rangle\}$. $\emptyset^s, \cup^s$ have the obvious meanings. Again following [BNTW95], we view the type $\{unit\}$ as encoding the booleans with $\{\}$ standing for *false* and $\{\langle\rangle\}$ standing for *true*. Then $= \langle x, y \rangle$ is defined to be $\{\langle\rangle\}$ when $x = y$, and $\{\}$ when $x \neq y$. Similarly, $not(\{\}) \stackrel{\text{def}}{=} \{\langle\rangle\}$, and $not(\{\langle\rangle\}) \stackrel{\text{def}}{=} \{\}$.

$\mathcal{NRA}(\Sigma)$ can express the following functions: cartesian product, $\times : \{s\} \times \{t\} \to \{s \times t\}$; equality at every type t, $=: t \times t \to \{unit\}$; membership test at type t, $member : t \times \{t\} \to \{unit\}$; set difference $- : \{t\} \times \{t\} \to \{t\}$; selection $\sigma_p : \{t\} \to \{t\}$ over any predicate $p : t \to \{unit\}$ defined in the language; $unnest : \{s \times \{t\}\} \to \{s \times t\}$; $nest : \{s \times t\} \to \{s \times \{t\}\}$. See [BNTW95] for more details and discussions.

Each expression $f : \{t_1\} \times \ldots \times \{t_k\} \to \{t\}$ in $\mathcal{NRA}(\Sigma)$ induces a query F of type $\sigma \to t$, where $\sigma = (u_1 \to v_1, \ldots, u_l \to v_l; t_1, \ldots, t_k)$, which on a database instance $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$ computes the relation $F(\mathcal{D}) \stackrel{\text{def}}{=} f\langle R_1, \ldots, R_k\rangle$. As a query language, $\mathcal{NRA}(\Sigma)$ is essentially equivalent to Abiteboul and Beeri's *extended algebra without powerset* [AB88] with external functions $p_1, \ldots, p_l$.

Proposition 4.1 *Every query expressed in $\mathcal{NRA}(\Sigma)$ is ef-domain independent.*

Proof. We will prove the following slightly stronger statement, by induction on the structure of an expression f:

For any two database instances $\mathcal{D} = (D; P_1, \ldots, P_l;)$ and $\mathcal{D}' = (D'; P'_1, \ldots, P'_l;)$ and any database morphism $\lambda : \mathcal{D} \to \mathcal{D}'$, it is the case that $f_{\mathcal{D}} \circ \lambda_s^{-1} \sqsubseteq \lambda_t^{-1} \circ f_{\mathcal{D}'}$. It is "slightly stronger" in that it does apply even to functions f which, technically, are not queries. We illustrate some of the cases for f.

p_j**.** Here $P_j \circ \lambda^{-1} \sqsubseteq \lambda^{-1} \circ P'_j$ follows from the definition of a database morphism.

$g \circ f$**.** We have:

$$\begin{aligned} g \circ f \circ \lambda^{-1} &\sqsubseteq g \circ \lambda^{-1} \circ f \text{ by induction hypothesis for } f \\ &\sqsubseteq \lambda^{-1} \circ g \circ f \text{ by induction hypothesis for } g \end{aligned}$$

The other higher-order constructs $\langle f_1, \ldots, f_n\rangle$ and $map(f)$ are treated similarly.

$\cup$**.** Assume $\lambda^{-1}(x') \cup \lambda^{-1}(y')$ is defined. This only means that $x = \lambda^{-1}(x')$ and $y = \lambda^{-1}(y')$ are defined; obviously $x' = \lambda(x), y' = \lambda(y)$. Then $\lambda(x \cup y)$ is defined too, because of the way λ is extended from base types to set types, and $\lambda(x \cup y) = x' \cup y'$. Hence $\lambda^{-1}(x' \cup y')$ is defined too, and equal to $x \cup y$, which proves $\lambda^{-1}(x') \cup \lambda^{-1}(y') = \lambda^{-1}(x' \cup y')$. So we have shown $\cup \circ \lambda^{-1} \sqsubseteq \lambda^{-1} \circ \cup$.

All other first-order constructs $id, !, \pi, \emptyset, \eta, \mu, \rho_2, not$, and $=$ are treated similarly.

□

Next we consider extensions of $\mathcal{NRA}(\Sigma)$ with various kinds of iterations. Figure 2 gives the typing rules for these iterative constructs.

For the fixpoint construct we adopt the inflationary semantics: $\mathit{fix}(f)(z) = \bigcup_{n\geq 0} x_n$, where $x_0 \stackrel{\text{def}}{=} \emptyset$, $x_{n+1} \stackrel{\text{def}}{=} x_n \cup f\langle z, x_n\rangle$. By convention, $\mathit{fix}(f)(x)$ is undefined when any of x_n is undefined, or when $\bigcup x_n$ is infinite. The type r is meant to compensate for the lack of free variables in the language. See [GV91, GG91, Suc94] for fixpoints on complex objects. We denote with $\mathcal{NRA}(\Sigma) + \mathit{fix}$ the extension of $\mathcal{NRA}(\Sigma)$ with the fixpoint construct.

$$\frac{f : r \times \{t\} \to \{t\}}{\mathit{fix}(f) : r \to \{t\}}$$

$$\frac{f : t \to t}{\mathit{loop}(f) : \{s\} \times t \to t}$$

$$\frac{e : r \to t \quad i : s \times t \to t}{\mathit{sri}(e,i) : r \times \{s\} \to t}$$

$$\frac{e : r \to t \quad f : r \times s \to t \quad u : t \times t \to t}{\mathit{dcr}(e,f,u) : r \times \{s\} \to t}$$

Figure 2: Iterative constructs

The *loop* construct has the following meaning: $\mathit{loop}(f)\langle x, y\rangle = f^{(n)}(y)$, where $n =\mid x \mid$.

The *structural induction on the insert presentation* is defined by: $\mathit{sri}(e,i)\langle z, \emptyset\rangle \stackrel{\text{def}}{=} e(z)$, $\mathit{sri}(e,i)\langle z, \{x\} \cup y\rangle \stackrel{\text{def}}{=} i\langle x, \mathit{sri}(e,i)\langle z, y\rangle\rangle$. In order for it to be defined, i has to be *commutative* and *idempotent*, see [BTS91]. Again, the type r is meant to compensate for the lack of free variables.

Finally the *divide and conquer recursion* on sets [SBT94], $\mathit{dcr}(e,f,u)$, is defined by: $\mathit{dcr}(e,f,u)\langle z, \emptyset\rangle \stackrel{\text{def}}{=} e(z)$, $\mathit{dcr}(e,f,u)\langle z, \{x\}\rangle \stackrel{\text{def}}{=} f\langle z, x\rangle$, and $\mathit{dcr}(e,f,u)\langle z, y \cup y'\rangle \stackrel{\text{def}}{=} u\langle \mathit{dcr}(e,f,u)(y), \mathit{dcr}(e,f,u)(y')\rangle$, when $y \cap y' = \emptyset$. The function $\mathit{dcr}(e,f,u)$ is defined only if u is commutative, associative, and has $e(z)$ as identity. This form of recursion is of interest because it captures the parallel complexity class NC over flat, ordered databases [SBT94].

Proposition 4.2 *All queries expressed in any of the languages below are ef-domain independent:*

$$\mathcal{NRA}(\Sigma, \mathit{fix}), \mathcal{NRA}(\Sigma, \mathit{loop}), \mathcal{NRA}(\Sigma, \mathit{sri}), \mathcal{NRA}(\Sigma, \mathit{dcr})$$

Proof. The proof consists in extending the induction in the proof of Proposition 4.1 to the case when f is one of the iterative constructs. In all cases, we have to proof that our statement holds, by induction on the number of steps of the iterative construct. We illustrate this for the *dcr* construct, for which the induction on the number of steps becomes a structural induction on a finite set. We shall abbreviate $\mathit{dcr}(e,f,u)$ with *dcr*.

We prove by induction on the set y' that whenever $\mathit{dcr}(\lambda^{-1}(z'), \lambda^{-1}(y'))$ is defined, so is $\lambda^{-1}(\mathit{dcr}(z', y'))$, and they are equal. We consider the cases when y is (1) $\emptyset$, (2) $\{x\}$, and (3) $y \cup y'$.

(1) $\emptyset$.

$$\begin{aligned}
\mathit{dcr}\langle \lambda^{-1}(z), \emptyset\rangle &= e(\lambda^{-1}(z')) \text{ by definition of } \mathit{dcr} \\
&\sqsubseteq \lambda^{-1}(z') \text{ by induction hypothesis on } e \\
&= \lambda^{-1}(\mathit{dcr}\langle z', \emptyset\rangle)
\end{aligned}$$

(2) $\{x'\}$.

$$\begin{aligned} dcr\langle\lambda^{-1}(z'), \lambda^{-1}(\{x'\})\rangle &= dcr\langle\lambda^{-1}(z'), \{\lambda^{-1}(x')\}\rangle \text{ by definition of } \lambda \\ &= f\langle\lambda^{-1}(z'), \lambda^{-1}(x')\rangle \text{ by definition of } dcr \\ &\sqsubseteq \lambda^{-1}(f\langle z', x'\rangle) \text{ by induction hypothesis on } f \\ &= \lambda^{-1}(dcr\langle z', \{x'\}\rangle) \text{ by definition of } dcr \end{aligned}$$

(3) $y_1' \cup y_2'$.

$$\begin{aligned} & dcr\langle\lambda^{-1}(z'), \lambda^{-1}(y_1' \cup y_2')\rangle \\ = \;& dcr\langle\lambda^{-1}(z'), \lambda^{-1}(y_1') \cup \lambda^{-1}(y_2')\rangle \text{ by the definition of } \lambda \\ = \;& u\langle dcr\langle\lambda^{-1}(z'), \lambda^{-1}(y_1')\rangle, dcr\langle\lambda^{-1}(z'), \lambda^{-1}(y_2')\rangle\rangle \text{ by definition of } dcr \\ \sqsubseteq \;& u\langle\lambda^{-1}(dcr\langle z', y_1'\rangle), \lambda^{-1}(dcr(z', y_2'))\rangle \text{ by induction hypothesis on } y_1' \text{ and } y_2' \\ \sqsubseteq \;& \lambda^{-1}(u\langle dcr\langle z', y_1'\rangle, dcr\langle z', y_2'\rangle\rangle) \text{ by induction hypothesis for } u \\ = \;& \lambda^{-1}(dcr\langle z', y_1' \cup y_2'\rangle) \text{ by definition of } dcr \end{aligned}$$

□

While all queries in $\mathcal{NRA}(\Sigma)$ are em-domain independent, the following example proves that the queries in $\mathcal{NRA}(\Sigma, \mathit{fix})$ are not:

Example 4.3 *Consider $\Sigma = \{p\}$, where $p : d \to d$ is some unary external function, and let $f : \{d\} \to \{d\}$ be the query $f(z) = \mathit{fix}(\lambda(z, x).z \cup \mathit{map}(p)(x))(z)$*[9]*. That is, $f(z)$ applies repeatedly p to all elements of z, until no new element is generated. If the set of all generated elements is finite and all calls to p are defined, then $f(z)$ returns that set; else it is undefined. Then f is sem-domain independent, but not em-domain independent (nor is it bounded-depth domain independent [AB88]).*

5 Computable Queries

In the previous Section we offered examples of query languages $\mathcal{L}(\Sigma)$ parameterized by some signature Σ. By adding new language constructs we obtain more powerful query languages: e.g. we have $\mathcal{NRA}(\Sigma) \subset \mathcal{NRA}(\Sigma, dcr) = \mathcal{NRA}(\Sigma, loop) = \mathcal{NRA}(\Sigma, sri)$, see [SW95]. For *fix*, we have, for $\Sigma = \emptyset$, $\mathcal{NRA}(\mathit{fix}) = \mathcal{NRA}(loop)$, while for certain Σ, $\mathcal{NRA}(\Sigma, \mathit{fix}) \not\subseteq \mathcal{NRA}(\Sigma, loop)$, because, over total database, the former language can express partial queries, while the latter expresses only total queries.

The question arises: when do we stop adding new constructs to a query language $\mathcal{L}(\Sigma)$? We expect each query expressed in a reasonable query language to be ef-domain independent. In addition,

[9] We use a more liberal notation of queries in $\mathcal{NRA}(\Sigma, \mathit{fix})$ with variables. Here $\lambda\langle z, x\rangle.z \cup \mathit{map}(p)(x)$ denotes the function g such that $g(u) = \pi_1(u) \cup \mathit{map}(p)(\pi_2(u))$. See [BNTW95] for a discussion.

all such queries are *computable*, in a sense made precise in this section. Hence a good criteria for stopping to enrich a database language is to test whether it can express all ef-domain independent computable queries: we call such a query language *complete*.

Chandra and Harel [CH80] call a database query F (on databases without external functions) *computable* iff there is some Turing Machine T which, whenever presented with an encoding of an input database instance $\mathcal{D}$, computes an encoding of $F(\mathcal{D})$ (and diverges when $F(\mathcal{D})$ is undefined).

There are two ways of presenting external functions as inputs to some Turing Machine T:

1. Require all external functions to be *Turing computable*, i.e. *recursive* [HR87], as number theoretic functions, and replace each function P_j by its Gödel number e_j, i.e. a number representing the encoding of the Turing Machine computing P_j. Provide T with both encodings for $R_1, \ldots, R_k$, and the l numbers $e_1, \ldots, e_l$ as inputs. Then T will compute an encoding of $F(\mathcal{D})$. We will discuss the notion of computability resulting from this model in Subsection 5.1.
2. Extend the Turing Machine T with *oracles* [HR87], one for each function P_j. That is T still receives on its input tape only the encoding of $R_1, \ldots, R_k$, as in the case without external functions, but now at any time during its computation, T may choose to ask one of the l oracles, say oracle j, for the output of $P_j(x)$, for some value x encoded on the tape. We will follow this idea, but combine it with order-independent computations [AV91] in Subsection 5.2.

The second approach is somehow broader, in the sense that it applies to database instances where the external functions are not necessarily computable, a case which is of little interest in practice. But when the external functions are computable and total, then we will prove that the two notions coincide. Interestingly from a theoretical point of view, the two notions differ on partial databases: a Turing Machine which receives Gödel numbers $e_1, \ldots, e_l$ on its tape can perform more complex computations than a Turing Machine with oracles.

We will restrict ourselves for the remaining of this paper to database instances with countable domain D, and for each such database instance consider some fixed enumeration of its domain, $\lambda : \mathbb{N} \to D$.

5.1 Computations with Gödel Numbers

Definition 5.1 *A database instance* $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$ *is a* **computable database instance** *iff all external functions* $P_1, \ldots, P_l$ *are computable.*

Definition 5.2 *A query* F *is* **computable** *iff there exists a Turing Machine* T *such that for any computable database instance* $\mathcal{D}$*, when* T *is started with an encoding of* $R_1, \ldots, R_k$ *and with the Gödel numbers* $e_1, \ldots, e_l$ *of the external functions in* $\mathcal{D}$ *on its tape, halts iff* $F(\mathcal{D})$ *is defined, and in this case leaves an encoding of* $F(\mathcal{D})$ *on its tape.*

First we prove that any computable, ef-domain independent query is continuous. For this we need the following recursion-theoretic lemma. Let $\varphi_0, \varphi_1, \ldots$ be a standard enumeration of all recursive functions [HR87]. We write $\varphi_e(x) \downarrow$ when $\varphi_e(x)$ is defined, and $\varphi_e(x) \uparrow$ otherwise. Also we denote with $\varphi_e^y(x)$ the result of letting the Turing Machine encoded by e run for at most y steps: that is $\varphi_e^y(x) = \varphi_e(x)$ if it takes at most y steps to compute x, and $\varphi_e^y(x)$ is undefined otherwise.

Lemma 5.3 *Let $f : \mathbb{N} \to \mathbb{N}$ be some recursive function with the property $\varphi_e \sqsubseteq \varphi_{e'} \Rightarrow f(e) \sqsubseteq f(e')$. (That is, whenever $graph(\varphi_e) \subseteq graph(\varphi_{e'})$ and $f(e)$ is defined, then $f(e') = f(e)$.) Then $\forall e$, if $f(e)$ is defined, then there is some e_0 such that φ_{e_0} is a finite function, $\varphi_{e_0} \sqsubseteq \varphi_e$, and $f(e_0) = f(e)$.*

The lemma essentially says that, whenever f maps encodings of functions to numbers in a monotone way, then $f(e)$ is fully determined by the action of f on the encodings of the finite approximations of φ_e.

Proof. Suppose that for all e_0 for which $\varphi_{e_0} \sqsubseteq \varphi_e$ and φ_{e_0} is finite, $f(e_0)$ is undefined. Then, we give a semi-decision procedure for $\bar{K}$ (here $\bar{K}$ is the complement of $K = \{z \,/ \varphi_z(z) \downarrow\}$), which is a contradiction. Indeed, let $k(z)$ be defined by: $\varphi_{k(z)}(y) = (if\ \varphi_z^y(z) \uparrow\ then\ \varphi_e(y)\ else \uparrow)$[10]. When $z \in \bar{K}$, that is $\varphi_z(z) \uparrow$, then $\varphi_{k(z)} = \varphi_e$, because $\varphi_z^0(z) \uparrow, \varphi_z^1(z) \uparrow, \varphi_z^2(z) \uparrow, \ldots$, hence $\varphi_{k(z)}(y) = \varphi_e(y)$, $\forall y$. When $z \in K$, then $\varphi_{k(z)}$ is a finite restriction of φ_e. Indeed, then $\varphi_z(z) \downarrow$. Let y be the number of steps performed by $\varphi_z(z)$. Then $\varphi_{k(z)}(0) = \varphi_e(0), \ldots, \varphi_{k(z)}(y-1) = \varphi_e(y-1)$, and $\forall y' \geq y$, $\varphi_{k(z)}(y)$ is undefined. So $f(k(z)) \downarrow$ iff $z \in \bar{K}$. This implies that $\bar{K}$ is r.e., which is a contradiction. $\square$

The lemma immediately implies:

Corollary 5.4 *All computable, ef-domain independent queries are continuous.*

Proof. Let F be some computable, ef-domain independent query, and let

$$\mathcal{D} = (D; R_1, \ldots, R_k; P_1, \ldots, P_l)$$

be some computable database instance, s.t. $F(\mathcal{D})$ is defined. Fix the encodings for $R_1, \ldots, R_k$, and let the Gödel numbers for $P_1, \ldots, P_l$ vary, on the input tape of the Turing Machine computing F. Call f the function computed by that Turing Machine, i.e. $f(e_1, \ldots, e_l)$ returns the encoding of $F(D; P_1, \ldots, P_l; R_1, \ldots, R_k)$, for fixed $R_1, \ldots, R_k$, and for any functions $P_1, \ldots, P_l$ with Gödel numbers $e_1, \ldots, e_l$. Since F is ef-domain independent, it follows that f is monotone, i.e. whenever $\varphi_{e_1} \sqsubseteq \varphi_{e'_1}, \ldots, \varphi_{e_l} \sqsubseteq \varphi_{e'_l}$, we have $f(e_1, \ldots, e_l) \sqsubseteq f(e'_1, \ldots, e'_l)$. Then, by applying Lemma 5.3 l times, once for each argument, we get that, for given $e_1, \ldots, e_l$ for which $f(e_1, \ldots, e_l)$ is defined, there are finite approximations $e_1^0, \ldots, e_l^0$ (i.e. $\varphi_{e_1^0} \sqsubseteq \varphi_{e_1}, \ldots, \varphi_{e_l^0} \sqsubseteq \varphi_{e_l}$ and $\varphi_{e_1^0}, \ldots, \varphi_{e_l^0}$ are finite) such that $f(e_1^0, \ldots, e_l^0)$ is defined, and, by monotonicity, equal to $f(e_1, \ldots, e_l)$. So it suffices to chose as finite

[10] $\varphi_z^y(z) \uparrow$ means that $\varphi_z(z)$ does not converge after y steps, and is a decidable property.

approximation $\mathcal{D}_0 \stackrel{\text{def}}{=} (D_0; R_1, \ldots, R_k; P_1^0, \ldots, P_l^0)$, where $P_1^0, \ldots, P_l^0$ are the functions encoded by $\varphi_{e_1^0}, \ldots, \varphi_{e_l^0}$, and D_0 is the active domain (all atoms in $R_1 \cup \ldots \cup R_k$) union all atoms in the graphs of $P_1^0, \ldots, P_l^0$, to get $F(\mathcal{D}_0) = F(\mathcal{D})$. □

Finally, we can define *complete* query languages $\mathcal{L}$ relative to some class $\mathcal{C}$ of database instances.

Definition 5.5 *Let $\mathcal{C}$ be a class of database instances. A query language $\mathcal{L}$ is* **complete** *over $\mathcal{C}$ iff it can express all computable, ef-domain independent queries over databases from $\mathcal{C}$.*

The reason for defining computability only relative to a class $\mathcal{C}$ of database instances is that we need to impose three kinds of restrictions on databases: (1) a restriction to *computable* databases (in this Subsection), (2) a restriction to *total* databases (in Section 6), and (3) a restriction to database instances whose external functions provide some limited arithmetic power (in Section 7).

5.2 Order-independent Computations with Oracles

Our second notion of computable queries is based on a variant of Turing Machines with oracles. In [HR87], oracles are introduced to compare the relative degrees of computability of number theoretic functions: the interesting cases are when the oracles are non-computable functions. For different purposes, Abiteboul and Vianu in [AV91] introduce the notion of *loose Generic Machine*, later simplified to *Relational Machines* in [AVV92]. These perform order-independent computations on their input databases. In some sense they can be also viewed as Turing Machines with oracles, where the oracle performs, on request, first-order transformations on a relational store. The Relational Machines do not gain more computational power than the Turing Machines, but allow a clear separation of the unordered data in the relational store from the ordered data on the tape.

Here we borrow ideas from both extensions of the Turing Machines, and define **Relational Machines for Complex Objects (RMC) over a signature** Σ, where $\Sigma = \{p_1 : u_1 \to v_1, \ldots, p_l : u_l \to v_l\}$, to be a one tape, deterministic Turing Machine extended with a fixed number of relational registers, $R_0, R_1, \ldots, R_r$ of types $t_0, t_1, \ldots, t_r$ respectively. Register R_i will hold values of type $\{t_i\}$ throughout the computation. The instructions of M consist of traditional Turing Machine instructions, and some additional instructions affecting the relational store only. The latter can have one of the following two forms:

Conditional *if $R_i = \emptyset$ then goto q else goto q'*: inspect the content of some register R_i and enter in state q or q', depending on whether R_i is empty or not.

Assignment $R_i \leftarrow h\langle R_{i_1}, \ldots, R_{i_k}\rangle$; *goto* q: Replace the content of register R_i with $h\langle R_{i_1}, \ldots, R_{i_k}\rangle$, where $h : \{t_{i_1}\} \times \ldots \times \{t_{i_k}\} \to \{t\}$ is a query in the language $\mathcal{NRA}(\Sigma)$, then enter state q. In particular h may be one of the external functions in Σ, or may be some expression involving external functions from Σ: we view this assignment as an oracle inquire, asking for the value of h on particular inputs. We keep in mind that h may be partial: if h is not defined for the current values of $R_{i_1}, \ldots, R_{i_k}$, then M's computation never terminates.

In order to run an RMC M we have to fix first the external functions $P_1, \ldots, P_l$. Then M starts with an empty tape, with some input relations in the registers $R_1, \ldots, R_k$, and with all other registers initialized to $\emptyset$. During its execution M may write on the tape, move the tape's head, change the values of the registers during assignment instructions, and test the values in registers during conditional instructions. During an assignment instruction it may invoke some of the external functions $P_1, \ldots, P_l$: since they are partial functions, and assignment may be undefined, and then we say that the entire computation of M is undefined. If M ever stops (by entering a terminal state), then the result of the computation is the value in R_0; otherwise its computation is undefined.

We associate to each RMC M with k designated input registers a database query F as follows. On some database instance $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$, $F(\mathcal{D})$ is the result of running M on the inputs $R_1, \ldots R_k$ and with the external functions bound to $P_1, \ldots, P_l$. The type of F is $\sigma \to t_0$, where $\sigma = (u_1 \to v_1, \ldots, u_l \to v_l; t_1, \ldots, t_k)$. Here $u_1 \to v_1, \ldots, u_l \to v_l$ are from Σ, while $t_0, t_1, \ldots, t_k$ are the types of the first $k+1$ registers in M.

There are two key differences between RMC's and the Relational Machines in [AVV92]. The first is the presence of external functions in RMC's. The second is the fact that RMC's handle complex objects, as opposed to only flat relations. This latter fact alone marks a significant difference between RM's and RMC's, which we explain next.

For some first order structure I, and for each $k \geq 1$, define the equivalence relation $\equiv_I$ on k-tuples from I to be: $\bar{u} \equiv_I \bar{v}$ iff for every first order formulas φ with k free variables, $I \models \varphi(\bar{u})$ iff $I \models \varphi(\bar{v})$. Relational Machines cannot break equivalence classes under $\equiv_I$, and hence they cannot compute every generic (in the sense of Chandra and Harel [CH80]) and computable query [AV91]: for example they cannot compute the parity of a unary relation. RMC's on the other hand *can* break equivalence classes. In fact it follows from Theorem 6.2 below that, even without external functions, RMC's can compute precisely the generic and computable queries, both on flat relations and on complex objects. Here we illustrate in the following two examples how to compute parity with an RMC.

Example 5.6 *The following RMC computes the query powerset* $: (;\{d\}) \to \{\{d\}\}$. *It has four registers* R_0, R_1, R_2, R_3 *of types* $\{d\}, d, \{d\}, \{d\}$ *respectively:* R_0 *is the output register,* R_1 *the input register.* M *has five states* q_0, q_1, q_2, q_3, q_4, *with* q_0 *the initial state and* q_4 *the final (or terminal) state. Its instructions are:*

$$
\begin{aligned}
q_0 &: R_0 \leftarrow \eta(\emptyset) \cup map(\lambda\langle x, y\rangle.\eta(x) \cup y)(R_1 \times R_0) \textit{ goto } q_1 \\
q_1 &: R_3 \leftarrow R_0 - R_2 \textit{ goto } q_2 \\
q_2 &: \textit{if } R_3 = \emptyset \textit{ then goto } q_4 \textit{ else goto } q_3
\end{aligned}
$$

$$q_3 \ : \ R_2 \leftarrow R_0 \ goto \ q_0$$
$$q_4 \ :$$

Given some set $\{x_1, \ldots, x_n\}$ *in register* R_1*,* M *starts in state* q_0 *where it stores* $\{\emptyset\}$ *in* R_0*: note that* λ *here is denotes function abstraction.* M *does* n *iterations through the states* q_0, q_1, q_2, q_3*: at each step it increments* R_0 *to* $\{\emptyset\} \cup \{\{x\} \cup y \mid x \in R_1, y \in R_0\}$*, such that at step* k*,* R_0 *contains all subsets of* R_1 *of cardinality* $\leq k$*.* R_2 *is used to store the previous result: when no progress is made,* M *halts. Note that* M *doesn't use its tape at all.*

Example 5.7 *Any RMC can compute, and write on its tape, the cardinality of any of its registers. Indeed, it suffices to observe that the RMC in Example 5.6 takes exactly* n *iterations to compute the powerset of some set* $\{x_1, \ldots, x_n\}$*. As a consequence it can compute parity.*

Example 5.8 (Encoding of ordered databases on the tape) *Here we show how a RMC can write on its tape an encoding of an ordered database, stored in its registers. Assume* R_1, R_2, R_3 *to be of types* $t_1, d, d \times d$ *respectively, and assume that for some given relation in* R_1*,* R_2 *contains the active domain* $D = \{o_1, \ldots, o_n\}$ *of* R_1 *(i.e. all atoms mentioned in* R_1*), while* R_3 *contains a total order (sometimes called a permutation)* $o_1 < \ldots < o_n$*, i.e.* $R_3 = \{\langle o_i, o_j \rangle \mid i < j\}$*. This induces a function* $\lambda : N_0 \to D$*, where* $N_0 = \{0, 1, \ldots, n-1\}$*, namely* $\lambda(i) \stackrel{\text{def}}{=} o_{i+1}$*. Then a RMC* M *can write on its tape an encoding of the relation* R_1 *using the numbers in the set* N_0*, more precisely, an encoding of* $S_1 \in dom(t_1, N_0)$ *for which* $\lambda_{t_1}(S_1) = R_1$*. To prove this, observe first that, for any type* t *and any* $S \in dom(t, N_0)$*,* M *can compute* $\lambda(S)$*. Indeed, take* $t = d$*. Then* S *is a number, and* $\lambda(S) = o_{S+1}$*, which can be computed by iterating* S *times on* R_3*. For a set type* $t = \{t'\}$*,* M *will iterate over every element* ν *in the set* S *(recall that* S *is stored on* M*'s tape), compute* $\lambda(\nu)$*, and collect the results in a set register. Similarly for a product type. Then, in order to find* S_1*,* M *will generate in some order all relations* S_1 *in* $dom(t_1, N_0)$*, and will test whether* $\lambda(S_1) = R_1$*.*

We consider only **deterministic** RMC's in this paper, i.e. for which for any state q and any tape symbol a there is at most one instruction matching q, a. A computation may fail due to one of the following reasons: (1) it may loop, or (2) it may not have a matching instruction for the current, non-final state and tape symbol, or (3) it may perform an assignment for which the function h is not defined. In cases (2) and (3) we will say that the machine is **stuck**.

Note that M computes the query $F(\mathcal{D})$ without the need of encoding $\mathcal{D}$ on M's tape: this is the idea behind the order-independent computation of [AV91]. An assignment $R_i \leftarrow h(R_{i_1}, \ldots, R_{i_k})$ in which h is one of the external function symbols in Σ, or is an expression mentioning one or more such function symbols, makes an implicit call to the oracle for that external function.

Definition 5.9 *Some query* F *is called* **RMC-computable** *over some class* $\mathcal{C}$ *of database instances, iff there is some RMC* M *computing* F *over* $\mathcal{C}$*, i.e. for every database instance* $\mathcal{D}$ *in* $\mathcal{C}$*,* M *computes* $F(\mathcal{D})$*.*

Proposition 5.10 *Any RMC-computable query F over some class $\mathcal{C}$ is ef-domain independent and continuous over the class $\mathcal{C}$.*

Proof. Let F be computed by M. To show that F is ef-domain independent, consider a database morphism $\lambda : \mathcal{D} \to \mathcal{D}'$, and consider the two runs of M over $\mathcal{D}$ and $\mathcal{D}'$. We prove by induction on the number of steps t that either M got stuck on $\mathcal{D}$ earlier, or:

1. after t steps, both M on $\mathcal{D}$ and M on $\mathcal{D}'$ have exactly the same tape content, have the head in the same position, and are in the same state, and:
2. if we denote with $R_0, \ldots, R_r$ and $R'_0, \ldots, R'_r$ the content of M's register at step t when run on $\mathcal{D}$ and $\mathcal{D}'$ respectively, then $R'_i = \lambda(R_i)$, for $i = 0, r$.

We illustrate the more interesting cases, when the next instruction is a conditional or an assignment. For a conditional *if* $R_i = \emptyset \ldots$, note that R_i and R'_i are either both empty, or both nonempty, due to $R'_i = \lambda(R_i)$. For an assignment $R_i \leftarrow h\langle R_{i_1}, \ldots, R_{i_k}\rangle, goto\ q$, if h is undefined for the run of M on $\mathcal{D}$ then there is nothing to prove. Else, we also have that $\lambda(R_{i_1}), \ldots, \lambda(R_{i_k})$ are defined, so, by Definition 3.2 and Proposition 4.1 we have that $h\langle R'_{i_1}, \ldots, R'_{i_k}\rangle$ is defined too and $\lambda(h\langle R_{i_1}, \ldots, R_{i_k}\rangle) = h\langle R'_{i_1}, \ldots, R'_{i_k}\rangle$.

Continuity follows from the fact that M performs only a finite number of moves. Suppose $F(\mathcal{D})$ is defined. Then M's run on $\mathcal{D}$ will halt after a finite number of moves. Let D_0 be the set of all atomic values present at any time during the computation in M's registers: D_0 is finite. The database $\mathcal{D}_0$ obtained by restricting $\mathcal{D}$ to the domain D_0 is a finite approximation of D and $F(\mathcal{D}_0) = F(\mathcal{D})$.

□

Relational Machines for flat relations are not complete [AV91], e.g. they cannot compute parity. By contrast, even for $\Sigma = \emptyset$, Relational Machines for Complex objects are complete: this is a Corollary of Theorem 6.2. This striking difference comes from the ability of RMC to simulate parallel computations through the use of complex objects: to perform an order-dependent computation, an RMC would start by computing its active domain, then it would generate the set of all possible order relations on the active domain, finally perform in parallel the order-dependent computation for each order relation. The next lemma shows how these parallel computations can be done.

Lemma 5.11 (The Map Lemma) *Let M be some RMC computing the function $F : t \to t'$. Then there is some RMC M' computing $map(F) : \{t\} \to \{t'\}$.*

Proof. Assume M has $r + 1$ registers $R_0, \ldots, R_r$ of types $t_0, \ldots, t_r$. For simplicity, assume that M has only one input register, R_1. In this case $t = \{t_1\}, t' = \{t_0\}$ (recall that, by convention, each register R_i holds a value of type $\{t_i\}$). On some input $\{x_1, \ldots, x_n\}$, M' will have to simulate the n computation threads of M on each x_j, $j = 1, n$. Suppose first that M has no conditional instruction. Then we design M' to have $r+1$ registers $R'_0, \ldots, R'_r$, with R'_i of type $\{t_1\} \times \{t_i\}$ (so the value of R'_i is of type $\{\{t_1\} \times \{t_i\}\}$). M will have a separate input register of type $\{t_1\}$ (hence its value is of type $\{\{t_1\}\}$), and a separate output register of type $\{t_0\}$. Suppose that when we run M on x_j, register R_i contains y_j at time t. Then we design M' such that its register R'_i contains at time t the value $\{\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\}$, i.e. all the n values of R_i from the n computation threads, tagged with their original input. M' will have the same instructions as M, except for the assignments $R_i \leftarrow h\langle R_{i_1}, \ldots, R_{i_k}\rangle$, which are replaced by $R'_i \leftarrow h'(R'_{i_1}, \ldots, R'_{i_k})$. Here h' will: (1) compute $R'_{i_1} \times \ldots \times R'_{i_k}$, (2) select from the result only those values which have the same tag, to obtain a set whose elements are tuples of the form $\langle\langle x_j, y_{i_1}\rangle, \langle x_j, y_{i_2}\rangle, \ldots, \langle x_j, y_{i_k}\rangle\rangle$, (3) apply $map(\lambda\langle\langle x_j, y_{i_1}\rangle, \ldots, \langle x_j, y_{i_k}\rangle\rangle.\langle x_j, h\langle y_{i_1}, \ldots, y_{i_k}\rangle\rangle)$ to the previously computed set[11]. It is easy to see that M' simulates n parallel threads of computation of M on some input $\{x_1, \ldots, x_n\}$, in a synchronous way. Finally we add a prologue to M' to load the register R'_1 from its input register, and an epilogue to extract the result from the register R'_0.

Now consider the more complex case, when M has conditionals. Here the synchronism is no longer possible, because the n threads of computations will in general take different branches. Let us define a *trace* $\tau \in \{0,1\}^*$ of the computation of M on input x_j to be the sequence of 0's and 1's corresponding to the conditional instructions of that computation: 0 stands for the *then* branch, 1 stands for the *else* branch. That is τ will have a length equal to the number of conditional instructions executed during that particular computation. Several computations of M, say on inputs $x_{j_1}, x_{j_2}, \ldots$, can be performed in parallel, as before, as long as they have the same traces. On the other hand, there are at most n distinct traces for the n computations of M on $x_1, \ldots, x_n$. So we design M' to generate on its tape all possible traces, i.e. all sequences in $\{0,1\}^*$, in some order. For each of them, M' simulates in a synchronous parallel way the computations of M on those inputs x_j which have that particular trace. As it proceeds with the computation for one trace, M' will eliminated from its registers $R'_0, \ldots, R'_r$ all values tagged with x_j's which have a different trace: the "wrong" x_j's will become obvious when the computation reaches conditionals, since some values in the register R'_i subject to a conditional will want to take the other branch than that given by the current trace. So as M' proceeds with the simulation of the computations corresponding to the current trace, the set of active tags x_j shrinks, starting from the original set $\{x_1, \ldots, x_n\}$. If M' reaches M's final state, it adds the active values currently remaining in R'_1 to another working register, say R'_{r+1}. If not, M' will stop when it exhausts the current trace, i.e. when it needs to take a branch no longer recorded in the current trace. In both cases M' continues with the next trace.

M' will stop altogether when it observes that all original n inputs $x_1, \ldots, x_n$ occur as tags in the register R'_{r+1}: in this case all n computations have been performed, and M' extracts the values from R'_{r+1} into the output register.

Should any of the n computation threads get stuck, then so will M', when it reaches a trace leading to that stuck computation. Should M loop forever on some of the n computations, then M' will loop forever too, generating ever longer traces, without being able to finish all n computations. □

[11] That is, the function is $map(\lambda\langle\langle x_1, y_1\rangle, \ldots, \langle x_k, y_k\rangle\rangle.\langle x_1, h\langle y_1, \ldots, y_k\rangle\rangle)$.

Now we can prove completeness of RMC's with no external functions. Namely let T be a Turing Machine computing some generic, domain independent query F. We build some RMC M computing F: on some input x, M starts by constructing the active domain of x, say $A = \{o_1, \ldots, o_n\}$, and then generates all $n!$ permutations of A. Each permutation allows M to simulate T [AV91]. Next, we use the map lemma to simulate T on all $n!$ orders: since the original Turing Machine T defines an order-independent computation, all $n!$ computations will lead to the same result, which we extract at the end from a singleton set. Theorem 6.2 extends this idea to RMC's with external functions. As we shall see, the external functions add significantly to the complexity of the proof.

6 Equivalence of the Two Notions of Computability

We shall assume in this Section that all database instances are computable (i.e. their external functions are computable).

Proposition 6.1 *Any RMC-computable query is computable.*

Proof. Let M be a RMC, with registers $R_0, R_1, \ldots, R_r$, computing the query F. We design T to simulate M as follows. T will maintain on its tape both a copy of M's tape, and encodings of the contents of the registers $R_0, R_1, \ldots, R_r$. Then T will simulate M's moves in an obvious way. The more interesting part is the simulation consists in the simulation of an assignment instruction $R_i \leftarrow h(R_{i_1}, \ldots, R_{i_k})$, because here T has to compute h given the encodings of $R_{i_1}, \ldots, R_{i_k}$. The interesting part is when h is (or contains) some external function, say p_j. Then T uses the Gödel number e_j to simulate the Turing Machine computing P_j. $\Box$

The converse of Proposition 6.1 is more involved, and only holds for queries on total databases.

Theorem 6.2 *Let F be a computable, ef-domain independent database query. Then F is RMC-computable over the class of computable, total databases.*

Proof. We will prove that for any computable, ef-domain independent query F there exists some RMC-computable query F' such that F and F' coincide on computable, total databases. Let T be the Turing Machine computing F. Recall that T expects on its input tape both the encoding of $R_1, \ldots, R_k$, and the Gödel numbers $e_1, \ldots, e_l$ of the Turing Machines computing $P_1, \ldots, P_l$.

We will design a RMC M which simulates T. M will be able to simulate T only on total databases (on partial databases, M may be undefined, even when T is defined). We take F' to be the query defined by M.

At a first glance, M may simulate T as follows: it will start by computing the active domain of its input registers, say $\{o_1, \ldots, o_n\}$. Next it will generate all permutations on $\{o_1, \ldots, o_n\}$. Given one such permutation, it will use the numbers $0, 1, \ldots, n-1$ to encode the values in $R_1, \ldots, R_k$ on its tape, as in Example 5.8, will run T on this encoding, and will decode the result in some register R_i. Finally, we apply the Map Lemma to argue that M can perform these computations simultaneously, for all permutations of the active domain. The problem is that, before simulating T, M needs to write on T's tape the Gödel numbers $e_1, \ldots, e_l$ of the external functions $P_1, \ldots, P_l$. This is difficult, since the external functions are hard-coded in M as oracles: M may ask for the value $P_j(x)$ for some x and any external function P_j, but it doesn't know the Gödel number of P_j. Moreover, this number also depends on the particular enumeration of the infinite domain D of the database $\mathcal{D}$. The idea here is to let M search for a finite approximation $\mathcal{D}_0$ of $\mathcal{D}$ on which T halts (hence $F(\mathcal{D}_0) = F(\mathcal{D})$). M will generate successively all encodings of finite databases $\mathcal{D}_0$ on its tape (including the Gödel numbers $e_1, \ldots, e_l$), for which there exists a database morphism $\lambda : \mathcal{D}_0 \to \mathcal{D}$, and will run T on each of them. It will stop when it finds some $\mathcal{D}_0$ on which T stops. We expand this idea in the sequel.

It is easier to describe first a *nondeterministic* RMC M simulating T, and then explain how to transform M into a deterministic one. M receives its inputs in $R_1, \ldots, R_k$, and starts by computing the active domain D_0 in R_{k+1}, say $R_{k+1} = \{o_1, \ldots, o_n\}$. Later, the active domain will be extended to larger subsets of the accessible domain $\bar{D}$, such that at any time $R_{k+1} = \{o_1, \ldots, o_m\}$, with $m \geq n$ (and $m = n$ initially). M uses R_{k+2} to keep a subset of all permutations of the extended active domain: initially, R_{k+2} contains all $n!$ permutations of $\{o_1, \ldots, o_n\}$. The types of R_{k+1} and R_{k+2} are d and $\{d \times d\}$ respectively.

On the other hand, M keeps on its tape the isomorphic image $\mathcal{N}_0$ of some finite approximation of $\mathcal{D}$, more precisely an encoding of a finite database instance $\mathcal{N}_0 = (N_0; Q_1, \ldots, Q_l; S_1, \ldots, S_k)$, whose domain consists of m numbers $N_0 = \{z_1, z_2, \ldots, z_m\} \subseteq \mathbb{N}$, with $z_1 < z_2 < \ldots < z_m$ (the same m as the cardinality of D_0). The external functions $Q_1, \ldots, Q_l$ of the database $\mathcal{N}_0$ will be stored as encodings of their graphs — which are finite — rather than as Gödel numbers. Storing a finite function Q as a graph $\{\langle x_1, Q(x_1)\rangle, \ldots, \langle x_q, Q(x_q)\rangle\}$ is stronger than storing its Gödel number, because it allows us to ask the question "is $Q(x)$ defined ?". Initially, $N_0 = \{0, 1, \ldots, n-1\}$, and $Q_1, \ldots, Q_l$ are totally undefined. Any permutation in R_{k+2} of the elements in D_0, $o_{i_1} < o_{i_2} < \ldots < o_{i_m}$, induces a function $\lambda_0 : N_0 \to D$, by $\lambda_0(z_1) \stackrel{\text{def}}{=} o_{i_1}, \ldots, \lambda_0(z_m) \stackrel{\text{def}}{=} o_{i_m}$. M maintains the invariant that for any permutation in R_{k+2}, the function λ_0 induced by it defines a database morphism $\lambda_0 : \mathcal{N}_0 \to \mathcal{D}$. The relations $S_1 = \lambda_0^{-1}(R_1), \ldots, S_k = \lambda_0^{-1}(R_k)$ are not stored explicitly, and in fact they are different for λ_0's induced by the different permutations in R_{k+2}. However all databases $(N_0; Q_1, \ldots, Q_l; \lambda_0^{-1}(R_1), \ldots, \lambda_0^{-1}(R_k))$ will be isomorphic, and we denote with $\mathcal{D}_0$ their image under λ_0. Thus $\mathcal{D}_0$ will be a finite approximation of $\mathcal{D}$ isomorphic to $\mathcal{N}_0$.

Finally, M keeps a number s on its tape, representing the number of steps M will allow T to run on $\mathcal{N}_0$. Initially $s = 0$.

Each step of the simulation consists of two parts:

1. First M simulates T on the database instance $\mathcal{N}_0$ for s steps. To see how this is done, assume some permutation in R_{k+2} to be given, and let λ_0 be its associated function. M computes

encodings of $\lambda_0^{-1}(R_1), \ldots, \lambda_0^{-1}(R_k)$ as in Example 5.8, then computes the Gödel numbers[12] $e_1, \ldots, e_l$ for the functions $Q_1, \ldots, Q_l$, then runs T for s steps; if T terminates, then M decodes the result in one of its registers, say R_{k+3}. Of course, M cannot pick one particular permutation from R_{k+2}: here we use the Map Lemma to argue that M can run several simulations of T in parallel, one for every permutation in R_{k+2}. Since these permutations will produce isomorphic databases $\mathcal{N}_0$, all simulations will produce the same result in R_{k+3}, hence if any simulation of T terminates in s steps, then all terminate in s steps and produce the same result. In this case M halts successfully. Else, if the simulations do not terminate, then M enters the second part.

2. Nondeterministically M chooses one of the following ways of extending $\mathcal{N}_0$ or s:

   (a) M increments s, or

   (b) M starts over again, with a new subset $N_0 = \{z_1, \ldots, z_m\}$ of n distinct numbers, $z_1 < \ldots < z_m$. $Q_1, \ldots, Q_l$ will be initialized with the totally undefined functions, M assigns the active domain $\{o_1, \ldots, o_n\}$ to R_{k+1}, and the set of all its $n!$ permutation to R_{k+2}. This step is not really necessary for M to do the right job, but it simplifies our argument below that M is complete.

   (c) M extends the active domain D_0 of $\mathcal{D}$. For this M nondeterministically picks some external operation P_j and applies it to all possible inputs made up from the atoms in the current active domain $D_0 = R_{k+1} = \{o_1, \ldots, o_m\}$. Here it is important for P_j to be total, else this step doesn't terminate. New atomic values may be generated in this way, and M adds them to the active domain, extending R_{k+1} to $R_{k+1} = \{o_1, \ldots, o_{m'}\}$, with $m' \geq m$. Next, M extends N_0 from $\{z_1, z_2, \ldots z_m\}$, with $z_1 < \ldots < z_m$, to $\{z'_1, z'_2, \ldots, z'_{m'}\}$, with $z'_1 < \ldots < z'_{m'}$. Finally, M extends all permutations of $\{o_1, \ldots, o_m\}$ in R_{k+2} in all possible ways to permutations of $\{o_1, \ldots, o_{m'}\}$. The invariant that all λ_0's corresponding to the permutations in R_{k+2} are database morphisms is preserved, because the external functions $Q_1, \ldots, Q_l$ have not been modified.

   (d) M extends some external function Q_j of $\mathcal{N}_0$ (Q_j is picked nondeterministically from $Q_1, \ldots, Q_l$). For this M picks some input $x \in dom(u_j, N_0)$ on which Q_j is undefined (if there is no such x, then this nondeterministic branch fails), and some output $y \in dom(v_j, N_0)$. Next M extends Q_j by defining $Q_j(x) \stackrel{\text{def}}{=} y$. To maintain the invariant, M selects now from R_{k+2} only those permutations for which the corresponding λ_0 is still a database morphism. All we need to check is whether $\lambda_0(y) = P_j(\lambda_0(x))$. Again, it is important that P_j be total, else it is impossible for M to decide whether $P_j(\lambda_0(x))$ is undefined. If R_{k+2} becomes empty, then M fails on this nondeterministic branch.

If M ever halts successfully (in step 1), then it computes the right result, i.e. the output of T on (the encoding of) $\mathcal{D}$, because M computes $\lambda_0(F(\mathcal{N}_0))$ for some finite database $\mathcal{N}_0$, where $\lambda_0 : \mathcal{N}_0 \to \mathcal{D}$ is a database morphism: but then, since $\lambda_0(F(\mathcal{N}_0))$ is defined, it is equal to $F(\mathcal{D})$. We only have to argue that "M is complete", i.e. it has at least one terminating computation (recall that M is

[12] It is possible to compute the Gödel number from the graph representation of a finite function.

nondeterministic), when $F(\mathcal{D})$ is defined. Assume M has no halting computation. Let $\lambda : \mathbb{N} \to D$ to be the standard enumeration of the domain of $\mathcal{D}$, and consider the set of all finite databases $\mathcal{N}_0$ generated by M during any of its computations, for which at least one of the morphisms $\lambda_0 : \mathcal{N}_0 \to \mathcal{D}$ corresponding to some permutation in R_{k+2} is included in λ (i.e. $\lambda_0 \sqsubseteq \lambda$ or, equivalently, $\mathit{graph}(\lambda_0) \subseteq \mathit{graph}(\lambda)$). Let $\mathcal{S}$ be the set of databases of the form $\mathcal{D}_0 = \lambda(\mathcal{N}_0)$, with $\mathcal{N}_0$ as above, for which $\mathcal{D}_0$ is a approximation of the *accessible* approximation of $\mathcal{D}$ (Definition 3.5). Note that $\mathcal{S}$ is directed, since for any two databases $\mathcal{D}_0, \mathcal{D}'_0 \in \mathcal{S}$, M has the choice of extending the domain of $\mathcal{D}_0$ and its external function such as to include the whole domain of $\mathcal{D}'_0$, and similarly it can extend the graphs of the external functions such as to include the graphs of those in $\mathcal{D}'_0$. Let:

$$\bar{\mathcal{D}} \stackrel{\text{def}}{=} \bigcup_{\mathcal{D}_0 \in \mathcal{S}} \mathcal{D}_0$$

We prove that $\bar{\mathcal{D}}$ is the *accessible approximation* of $\mathcal{D}$. Obviously $\bar{\mathcal{D}}$ is included in the accessible approximation. For the converse, first we show that $\bar{D}$ contains the active domain of $\mathcal{D}$. Here we use step 2b above: eventually, M will choose to start over again with $N_0 = \lambda^{-1}(D_0)$, where D_0 is the active domain of $\mathcal{D}$.

Second we show that all external functions are totally defined on the domain of $\bar{\mathcal{D}}$, which implies that $\bar{\mathcal{D}}$ is the accessible approximation. Indeed suppose this were not the case, i.e. for some j and some $x \in \mathit{dom}(u_j, \bar{D})$, $\bar{P}_j(x)$ is not defined, and let $y \stackrel{\text{def}}{=} P_j(x)$. Let $\mathcal{D}_0$ be some finite database in $\mathcal{S}$ containing x, and let $\mathcal{N}_0 = (N_0; Q_1, \ldots, Q_l; S_1, \ldots, S_k)$ be its preimage under λ. Also let $x_0 \stackrel{\text{def}}{=} \lambda^{-1}(x)$, $y_0 \stackrel{\text{def}}{=} \lambda^{-1}(y)$. Assume first that $\mathit{atoms}(y_0) \subseteq N_0$; then M has the choice of extending Q_j s.t. $Q_j(x_0) = y_0$ (step 2d above), obtaining some database $\mathcal{N}'_0$: then $\lambda(\mathcal{N}'_0) \in \mathcal{S}$, and we conclude that $\bar{P}_j(x)$ is in fact defined, which contradicts our assumption. Now assume $\mathit{atoms}(y_0) \not\subseteq N_0$: then using step 2c above, M has the choice of extending the domain of $\mathcal{N}_0$ to include $\mathit{atoms}(y_0)$, and we are back to the first case. This concludes the proof that $\bar{\mathcal{D}}$ is the accessible approximation of $\mathcal{D}$.

By Proposition 3.8, F is forward-looking, hence, since $F(\mathcal{D})$ is defined, $F(\bar{\mathcal{D}})$ is defined too (and $F(\mathcal{D}) = F(\bar{\mathcal{D}})$). Since F is also continuous, there exists some finite approximation of $\bar{\mathcal{D}}$ on which F is defined. Since the family $\mathcal{S}$ is directed, we may assume without loss of generality that such an approximation is some $\mathcal{D}_0 \in \mathcal{S}$. But then T had to terminate on $\mathcal{N}_0 = \lambda^{-1}(\mathcal{D}_0)$: this gives us a terminating computation for M, contradiction.

Finally, M can be made deterministic using standard techniques [CLR90]. Namely observe that the nondeterministic choices during a computation of M can be encoded by a string of natural numbers. Thus, the deterministic version M' of M systematically generates all strings of natural numbers, and simulates M on each of them, until it reaches a successful computation. $\square$

In the absence of external functions, Theorem 6.2 implies:

Corollary 6.3 *Relational Machines are complete for complex objects.*

We conclude this section by showing how Theorem 6.2 fails on partial databases: in this case the two notions of computable queries no longer coincide. What distinguishes them is the fact that the RMC-computable queries are *sequential*, in a sense related to the notion of *sequential function* in [Cur86], while computable queries need not be.

Definition 6.4 *A query F is* **sequential** *iff for any database $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$ for which $F(\mathcal{D})$ is undefined, one of the following holds:*

1. *$F(\mathcal{D}')$ is undefined for any $\mathcal{D}'$ for which $\mathcal{D}$ is an approximation (i.e. $\mathcal{D} \sqsubseteq \mathcal{D}'$ implies $F(\mathcal{D}')$ is undefined), or*
2. *There exists j and $x \in dom(u_j, D)$ such that for all $\mathcal{D}' \sqsupseteq \mathcal{D}$, if $F(\mathcal{D}')$ is defined then $P'_j(x)$ is defined.*

In case condition 2 above holds, then we call the pair (j, x) a *sequentiality index* of F at $\mathcal{D}$. This terminology is consistent with that in [Cur86]. Note that the sequentiality index need not be unique, see the comment below.

The intuition behind Definition 6.4 is that F is sequential iff it invokes the external functions sequentially, say in the order $P_{j_1}(x_1), P_{j_2}(x_1), P_{j_3}(x_3), \ldots$, and if it *waits* for the computation of $P_{j_n}(x_n)$ to terminate before proceeding any further. If F diverges on some database $\mathcal{D}$, one of two things may happen. Either all the calls to the external functions above are defined, but the F's computation is infinite: then case 1 above holds, because if $\mathcal{D}'$ is some extension of $\mathcal{D}$, $F(\mathcal{D}')$ will end up in the same infinite computation. Or one of the calls to the external functions stalls, say $P_{j_n}(x_n)$, is undefined, which, of course, causes F to be undefined. Then case 2 above holds: indeed for any extension $\mathcal{D}'$ of $\mathcal{D}$, the computation of $F(\mathcal{D}')$ will reach the same point in which it computes $P_{j_n}(x_n)$, so this one must be defined if $F(\mathcal{D}')$ is defined.

One can prove that any function computed by a RMC is sequential, because a RMC applies the external functions one at a time, in a sequential manner:

Proposition 6.5 *All RMC-computable queries are sequential.*

Proof. Let F and $\mathcal{D}$ be as in Definition 6.4, let M be a RMC computing F, and suppose condition 1 does not hold. Recall that M is undefined when run on database $\mathcal{D}$ if it either (1) runs into an infinite computation, or (2) gets in a configuration in which no instruction applies, or (3) tries to execute some assignment instruction $R_i \leftarrow h(R_{i_1}, \ldots, R_{i_k})$ for which the function h is not defined. (1) and (2) cannot be the case, since there is some extension $\mathcal{D}'$ of $\mathcal{D}$ on which M halts. Since (3) holds, there must be some external function P_j which h tries to apply on some input x, and which is undefined on x in $\mathcal{D}$. Then (j, x) is a sequentiality index of F at $\mathcal{D}$. Indeed, let $\mathcal{D} \sqsubseteq \mathcal{D}'$, and suppose $F(\mathcal{D}')$ is defined. Then $P'_j(x)$ must be defined in $\mathcal{D}'$, or else M would get stuck when running on $\mathcal{D}'$ in the same place in which it got stuck on $\mathcal{D}$. $\square$

The sequentiality index (j, x) is not unique. For example consider the query

$$F(D; P_1, P_2; R) \stackrel{\text{def}}{=} map(\lambda x.\langle P_1(x), P_2(x)\rangle)(R)$$

and some database instance $\mathcal{D}$ on which F is undefined, where $R = \{x_1, \ldots, x_n\}$. Then one of the following $2n$ expressions must be undefined:

$$P_1(x_1), \ldots, P_1(x_n), P_2(x_1), \ldots, P_2(x_2)$$

Each of the $2n$ pairs (j, x_i), $i = 1, n; j = 1, 2$, for which $P_j(x_i)$ is undefined, is a sequentiality index for F at $\mathcal{D}$.

However computable queries need not be sequential, because a Turing Machine may simulate in parallel several computations of external functions. The following is an example of a computable, ef-domain independent query which is not sequential:

Example 6.6 *Consider the schema $\sigma = (d \to d; d)$, and the following query F:*

$$F(\mathcal{D}) \stackrel{\text{def}}{=} \begin{cases} R_1 & \textit{when } \exists x \in R_1 \textit{ such that } P_1(x) = x \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

where $\mathcal{D} = (D; P_1; R_1)$. This query is ef-domain independent, and computable. To see that it is computable, suppose $R_1 = \{x_1, \ldots, x_n\}$; a Turing Machine T can perform in parallel the computation steps for $P_1(x_1), \ldots, P_1(x_n)$, and stop when one of these computations, say for $P_1(x_i)$, finishes with $P_1(x_i) = x_i$. Thus T will not get stuck when some other computation, say for $P_1(x_j)$, never terminates. However this query is not RMC-computable because it is not sequential. Indeed, consider the partial database in which $R_1 = \{x_1, x_2\}$ and $P_1(x_1) = P_1(x_2) = $ undefined. Then $F(\mathcal{D})$ is undefined, but neither $(1, x_1)$ nor $(1, x_2)$ is a sequentiality index for F at $\mathcal{D}$, because we may extend in two different ways the database $\mathcal{D}$ to a database $\mathcal{D}'$, such that F is defined on $\mathcal{D}'$, by either defining $P_1'(x_1) \stackrel{\text{def}}{=} x_1$ or by defining $P_1'(x_2) \stackrel{\text{def}}{=} x_2$.

7 Complete Query Languages with External Functions

We give in this Section examples of complete query languages with external functions. All use the same technique for gaining completeness: some combination of external functions which allow the representation of natural numbers. For some type u and two expressions $z : unit \to u$ and $s : u \to u$ in $\mathcal{NRA}(\Sigma, fix)$, we denote $\mathcal{C}_{z,s}$ the class of databases in which the elements $z, s(z), s^{(2)}(z), \ldots, s^{(k)}(z), \ldots$ are distinct.

Proposition 7.1 *For any expressions z, s as above, $\mathcal{NRA}(\Sigma, fix)$ is complete w.r.t. the class $\mathcal{C}_{z,s}$.*

Proof. We represent natural numbers as a subset of the type u, namely $\{z, s(z), s^{(2)}(z), \ldots\}$. First we show that $\mathcal{NRA}(\Sigma, \mathit{fix})$ can express all number-theoretic recursive functions. More precisely, call some function $f : \mathbb{N}^k \to \mathbb{N}$ "representable" if there exists some expression $f' : u^k \to \{u\}$ in $\mathcal{NRA}(\Sigma, \mathit{fix})$ such that, in any database $\mathcal{D} \in \mathcal{C}_{z,s}$, the interpretation of f' is $f'(x) = \{f(x)\}$; more precisely, $\forall n \geq 0$, $f'(s^{(n)}(z)) = \{s^{(f(n))}(z)\}$. Then we show that all recursive functions f are representable. For that, we have to prove that the functions zero and successor are representable, and that the class of representable functions is closed under composition, primitive recursion and under minimization. For illustration we show here closure under minimization. For clarity assume $k = 1$. So let $f : \mathbb{N}^2 \to \mathbb{N}$ be some function represented by f', and consider $g(x) \stackrel{\text{def}}{=} \mu_y.(f\langle x, y\rangle = 0)$. We write in $\mathcal{NRA}(\Sigma, \mathit{fix})$ a fixpoint expression which, at iteration n, will produce the intermediate result $r_n = \{\langle z, f'\langle x, z\rangle\rangle, \langle s(z), f'\langle x, s(z)\rangle\rangle, \ldots, \langle s^{(n)}(z), f'\langle x, s^{(n)}(z)\rangle\rangle\}$, but which will stop increasing its intermediate result when $f'\langle x, s^{(n)}(z)\rangle = \{z\}$. Such an expression is $\mathit{fix}(h)$, with $h\langle x, r\rangle \stackrel{\text{def}}{=} \mathit{if}\ \exists\langle y, \{z\}\rangle \in r\ \mathit{then}\ r\ \mathit{else}\ \{\langle z, f'\langle x, z\rangle\rangle\} \cup \mathit{map}(\lambda\langle y, q\rangle.\langle s(y), f'\langle x, s(y)\rangle\rangle)(r)$. Now we can construct g'. $g'(x)$ will start by computing $r = \mathit{fix}(h)(x)$; if it terminates, then g' returns $\{y\}$, where y is such that $\langle y, z\rangle \in r$.

In view of Theorem 6.2, to prove that $\mathcal{NRA}(\Sigma, \mathit{fix})$ is complete it suffices to prove that it can express any RMC-computable query. Let M be some RMC. A configuration of a RMC with $r+1$ registers $R_0, \ldots, R_r$ of types $t_0, \ldots, t_r$ can be represented as an $r+4$ tuple of type $\{t_0\} \times \{t_1\} \times \ldots \times \{t_r\} \times \{u\} \times \{u\} \times \{u \times u \times u\}$: the first $r+1$ components describe the content of the relational registers, the last three components describe the current state of M, the head position, and the tape content T. Both current state and head positions are singleton sets of the form $\{s^{(n)}(z)\}$, where n is the state number or the head position respectively. The tape content T is a set of pairs $\langle s^{(i)}(z), s^{(c)}(z), s^{(t)}(z)\rangle$, where i, c, and t are numbers denoting the fact that cell i contains the character c at time t. The one-step-move relation on configurations is expressible in the language: it consists in doing some arithmetic to deal with the next and previous cells, and some operations on the registers, which are expressible in the language by the definition of a RMC. Finally one has to iterate one-step-move function until a final state is reached. $\square$

We use Proposition 7.1 to show that various languages with some arithmetic capability stemming from their external functions are complete.

Object Inventions Consider some base type ι whose elements are called *object id's*, and some external function $\mathit{make\_object} : \{\iota\} \to \iota$ which "generates" new id's: more precisely, we consider $\mathcal{C}$ to be the class of databases for which $\mathit{make\_object}(x) \notin x$, for all x of type $\{\iota\}$. Intuitively $\mathit{make\_object}(x)$ generates an id which was not previously in use: the set x consists of all id's currently in use, so *make_object* can construct a new one. It can be thought of as a Skolem function of the following higher order formula, stating that the type ι is infinite: $\forall x : \{\iota\}.\exists y : \iota.y \notin x$. Other base types and/or external functions may be present (recall that all results in presented here work also in the context of multiple base types, see Section 3). Then $\mathcal{NRA}(\Sigma, \mathit{fix})$ satisfies the requirements of Proposition 7.1, by taking $u \stackrel{\text{def}}{=} \{\iota\}$, $z \stackrel{\text{def}}{=} \emptyset$ and $s(x) \stackrel{\text{def}}{=} x \cup \{\mathit{make\_object}(x)\}$. It has been known previously that object inventions in conjunction with fixpoints give rise to complete query languages [AK89]. Here we use related tools to obtain completeness in the presence of external functions.

Untyped Sets Consider some base type u whose meaning is a restriction[13] of the *untyped sets* in [HS89]. That is, the class $\mathcal{C}$ of databases we consider interprets u as follows: it contains all finite sets which can be constructed from elements in D and from other elements in u. For example $x = \{a, \{b, c\}\}$ is a legal element of u, where $a, b, c \in D$ are atomic elements. We give this semantics to u by including two external functions in Σ: $mk_d : d \to u$ and $mk_u : \{u\} \to u$ in Σ: their intended semantics is $mk_d(a) = a$ and $mk_u(x) = \{x\}$. For example x above can be expressed as $mk_u(\eta(mk_d(a)) \cup \eta(mk_u(\eta(mk_d(b)) \cup \eta(mk_d(c)))))$. Let $\mathcal{C}$ be the class in which both mk_d and mk_u are injective and have disjoint codomains. Then by Proposition 7.1 $\mathcal{NRA}(\Sigma, \mathit{fix})$ is complete w.r.t. $\mathcal{C}$. Indeed, it suffices to take $z \stackrel{\text{def}}{=} mk_u(\emptyset)$, and $s(x) \stackrel{\text{def}}{=} mk_u(\{x\})$. That is, the natural numbers are represented by $\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\{\{\emptyset\}\}\}, \ldots$.

Natural Numbers Obviously, if we include $\mathbb{N}$ among the base types and give names to some arithmetic functions, say $0, 1, +$, in Σ, then $\mathcal{NRA}(\Sigma, \mathit{fix})$ will be complete w.r.t. to the class $\mathcal{C}$ of databases in which $\mathbb{N}, 0, 1, +$ have the standard interpretation.

8 Conclusions

We have proposed the notion of external-function domain independence to characterize the "reasonable" queries over databases with external functions. Next we have proposed two natural definitions for computability of queries on databases with external functions, and shown that they are equivalent when the external functions are totally defined. Thus "computability with external functions" is a robust notion. A natural question to ask next is how to define query complexity in the presence of external functions. Here one gets different notions depending on which model of computation is adopted. For example we could define some query F to be in PSPACE either when it is computed by some PSPACE Turing Machine expecting both an encoding of the input relations and the Gödel numbers of the external functions, or when it is computed by some RMC whose tape *and* relational store are polynomially bounded. It is not clear however that these two definitions are equivalent, leaving open the question of what a good definition for a PSPACE query might be. This issue has to be addressed in the future.

9 Acknowledgments

We thank Victor Vianu, Serge Abiteboul, Catriel Beeri and Rick Hull for commenting on an earlier version of this paper. We also thank Jan Van den Bussche for pointing out to us an error in Lemma 5.3.

[13]For clarity we don't include tuples in the set of untyped sets u. This could be achieved e.g. by adding an injective external function $u \times u \to u$ to Σ.

References

[AB88] Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex objects. In *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988. Also available as INRIA Technical Report 846.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[APV94] S. Abiteboul, C.H. Papadimitriou, and V. Vianu. The power of reflective relational machines. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pages 230–240, Paris, France, July 1994.

[AV91] Serge Abiteboul and Victor Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, 1991.

[AVV92] S. Abiteboul, M. Vardi, and V. Vianu. Fixpoint logics, relational machines, and computational complexity. In *Structure and Complexity*, 1992.

[BBW92] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In J. Biskup and R. Hull, editors, *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992*, pages 140–154. Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[Bel94] Stephen Bellantoni. Comments on two notions of higher type computability, 1994. Manuscript available from `sjb@cs.toronto.edu`.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of proramming with collection types. *Theoretical Computer Science*, 149:3–48, 1995.

[BTS91] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with Sets/Bags/Lists. In *LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.

[CH80] Ashok Chandra and David Harel. Computable queries for relational databases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[CLR90] Thomas H. Cormen, Charels E Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Con73] R. L. Constable. Type two computational complexity. In *Proceedings of the 5th annual ACM symposium on theory of computing*, pages 108–122, Austin, Texas, 1973.

[Cur86] P. L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Pitman, 1986.

[Deu90] O. Deux. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[Dig93] Digital Equipment Corporation. *ISO-ANSI Working Draft — Database Language SQL (SQL3)*, August 1993.

[EMHJ93] Martha Escobar-Molano, Richard Hull, and Dean Jacobs. Safety and translation of calculus queries with scalar functions. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 253–264, Washington, D. C., May 1993.

[GG91] Marc Gyssens and Dirk Van Gucht. A comparison between algebraic query languages for flat and nested databases. *Theoretical Computer Science*, 87:263–286, 1991.

[Gra80] G. Gratzer. *Universal Algebra*. Springer-Verlag, 1980.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[GV91] Stephane Grumbach and Victor Vianu. Expressiveness and complexity of restricted languages for complex objects. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 191–202. Morgan Kaufmann, August 1991.

[HH93] Tirza Hirst and David Harel. Completeness results for recursive data bases. In *Proceedings of 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 244–252, Washington, DC, May 1993.

[HR87] Jr. Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1987.

[HS89] Richard Hull and Jianwen Su. Untyped sets, inventions, and computable queries. In *Proceedings 8th ACM Sumposium on Principles of Database Systems*, pages 347–360, 1989.

[KC91] Bruce Kapron and Stephen Cook. A new characteriztion of Mehlhorn's ppolynomial time functionals. In *Symposium on Foundation of Computer Science*, pages 342–347, San Juan, Puerto Rico, October 1991.

[PG92] Jan Paredaens and Dirk Van Gucht. Converting nested relational algebra expressions into flat algebra expressions. *ACM Transaction on Database Systems*, 17(1):65–93, March 1992.

[Plo81] G. D. Plotkin. Post-graduate lecture notes in advanced domain theory. Department of Computer Science, University of Edinburgh, 1981. Available by email from: `kondoh@harl.hitachi.co.jp`.

[SBT94] Dan Suciu and Val Breazu-Tannen. A query language for NC. In *Proceedings of 13th ACM Symposium on Principles of Database Systems*, pages 167–178, Minneapolis, Minnesota, May 1994. See also UPenn Technical Report MS-CIS-94-05.

[Suc94] Dan Suciu. Fixpoints and bounded fixpoints for complex objects. In Catriel Beeri, Atsushi Ohori, and Dennis Shasha, editors, *Proceedings of 4th International Workshop on Database Programming Languages, New York, August 1993*, pages 263–281. Springer-Verlag, January 1994.

[SW95] Dan Suciu and Limsoon Wong. On two forms of structural recursion, January 1995.

[Top87] Rodney W. Topor. Domain-independent formulas and databases. *Theoretical Computer Science*, 52:281–306, 1987.