Domain-Independent Queries on Databases with External Functions

Dan Suciu

December 9, 1997

Abstract

We investigate queries in the presence of external functions with arbitrary inputs and outputs (atomic values, sets, nested sets etc). We propose a new notion of domain independence for queries with external functions which, in contrast to previous work, can also be applied to query languages with fixpoints or other kinds of iterators. Next, we define two new notions of *computable queries with external functions*, and prove that they are equivalent, under the assumption that the external functions are total. Thus, our definition of computable queries with external functions is robust. Finally, based on the equivalence result, we give examples of complete query languages with external functions. A byproduct of the equivalence result is the fact that Relational Machines are complete for complex objects: it was known that they are not complete over flat relations.

1 Introduction

Database functionalities are important both for practical and for theoretical purposes. E.g. the system O_2 of [12] allows the query language to invoke any method written in the programming language C, while the language COL of [1] provides a toolbox of external functions, which may be freely used in arbitrary queries. The practical integration of external functions in query languages is generally well understood, but the semantics of queries in the presence of external functions has received less attention. [5, 13] offer two distinct definitions for domain independent queries with external functions, but which don't fit languages with fixpoints or other forms of recursions. To the best of our knowledge, no previous attempt has been made to define complete query languages with external functions.

In this paper we propose a new definition of *domain independent queries with external functions* (*ef-domain independence*), in a general setting, namely by allowing the inputs and outputs of the external functions to be scalar values, sets, nested sets, etc. Queries expressed in languages with external functions and fixpoints or other forms of iterations indeed satisfy this definition. We establish the relationship of our notion of domain independence with those in [5, 13]. Next we propose two definitions for *computable queries with external functions* and show that they coincide, when the external functions are total (theorem 7.2). We take this as evidence for the robustness of the underlying concept. The equivalence is a technically difficult theorem: an interesting byproduct is the corollary that Relational Machines [4] for complex object are complete, while it is known that they are generally not complete for flat relations [6]. Subtle differences separate the two notions of computable queries when the external functions are partial: one definition requires *sequential* computation of the external functions, while the other allows for *parallelism*. The coincidence of the two definitions of computable queries for total external functions enables us to define a robust notion of *complete query languages* with external functions, namely as languages which can express all computable, domain independent queries with total external functions. Finally we give examples of such languages.

Abiteboul, Papadimitrou, and Vianu [3] extend Relational Machines with *reflections*, i.e. the ability to dynamically create queries, and to answer them in constant time; the resulting Reflective Relational Machines are complete. We obtain completeness by a different, orthogonal extension, namely by replacing flat realtions with complex objects. 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 [3]. The prarallelism implicit in one of our definition of computable queries is of a different nature and consists in the ability of a device to initiate the computation of several external functions in parallel, and to stop when *one* of them terminates.

Chandra and Harel in [9] consider *extended databases* by adding an interpreted domain F to the uninterpreted one D: any given algebraic structure may accompany F. The connection between the two domains is given by functions S going only in one direction, from D to F. Due to their type, these functions can only be applied once, making them strictly less general than external functions considered in [5, 13] and here, which can be repeatedly applied to values in D. The functions on F corresponding to its algebraic structure are also strictly less general than the external functions, because F is already "interpreted".

Abiteboul and Beeri add external functions to their *algebra* and to the *calculus*, and define the notion of *bounded-depth domain independence*. They show that queries expressed both in the *extended algebra* and in the *extended calculus* are bounded-depth domain independent. Similarly, Escobar-Molano, Hull, and Jacobs [13] define *embedded-domain independent queries* with scalar functions (a special case of external functions), and show that any query expressed by an *embedded-allowed* calculus formula are embedded domain independent. But we show here in example 4.1 that in a language with fixpoints, queries fail to be bounded-depth domain independent or embedded-domain independent.

The first description of a complete query language can be found in [9]: it achieves completeness in a dynamically typed language, by encoding an integer n as a set of tuples of width n. Other complete query languages use different tools to achieve completeness: e.g. object inventions in [2], and untyped sets in [20]. Here we use essentially the same techniques to design complete languages with external functions, w.r.t. our definition of computable queries.

Section 2 reviews some basic database notions and offers an intuition for the constructs to follow. Section 3 defines domain independent queries with external functions (ef-domain independent), shows some of their properties, and establish their relationship with embedded domain independent queries [13]. Section 4 briefly describes the Nested Relational Algebra with external functions, and shows that all queries expressed in this language, possibly extended with iterators, are ef-domain independent. Sections 5 and 6 give the two definitions of computable queries, while section 7 proves their equivalence. Finally we give examples of complete query languages in section 8.

2 Background and Motivation

A database query can be viewed as a (partial) function F mapping any database instance $\mathcal{D} = (D; R_1, \ldots, R_k)$ into some relation $F(\mathcal{D})$ over D. D is the domain of the database instance and $R_1, \ldots, R_k$ are its relations. It is understood that the arities of the relations R_i, as well as the arity of the output relation are fixed. More, it is usually required that the query be be **generic**, **domain independent** and **computable**. *Generic* means that whenever $\mathcal{D}$ is isomorphic to some structure $\mathcal{D}'$, then the same isomorphism maps $F(\mathcal{D})$ into $F(\mathcal{D}')$; we will assume throughout this paper that all queries are generic in this sense, i.e. map isomorphic database instances to isomorphic outputs. *Domain independence* can be stated as the requirement that, if we replace the domain D with a larger one $D' \supseteq D$, but keep the same relations $R_1, \ldots, R_k$, then the query F returns the same answer on the new database instance $\mathcal{D}' = (D'; R_1, \ldots, R_k)$, i.e. $F(\mathcal{D}) = F(\mathcal{D}')$. Finally, a query is *computable* if there is some Turing Machine which, when started with an encoding of $R_1, \ldots, R_k$ on its tape, halts with an encoding of $F(\mathcal{D})$ on the tape, or diverges, when $F(\mathcal{D})$ is undefined.

Most of the external functions we will consider in this paper, like $+$, *succ*, *make_object*, etc. have infinite domains and codomains. This leads us to 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.

In the context of complex objects, we consider higher order structures instead of first order ones. Namely we define **complex object types** by the grammar $t ::= d \mid t \times \ldots \times t \mid \{t\}$, and define $dom(t, D)$, for some type t and set D to be: $dom(d, D) \stackrel{\text{def}}{=} D$, $dom(t_1 \times \ldots \times t_n, D) \stackrel{\text{def}}{=} dom(t_1, D) \times \ldots \times dom(t_n, D)$, $dom(\{t\}, D) \stackrel{\text{def}}{=} \mathcal{P}_{fin}(dom(t, D))$. 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 $R_i \subseteq dom(t_i, D)$. The empty product (obtained by taking $n = 0$ in $t_1 \times \ldots \times t_n$) is denoted with *unit*; for any D, $dom(unit, D) = \{()\}$. The notion of a query over flat databases carries over to the complex object databases. The definitions and notations are consistent with those of [15, 20], and all the results in this paper hold also for multisorted databases (with more than one base type: $d, d', \ldots$), but in order to keep our formalism simple, we shall restrict ourselves in the sequel to only one base type.

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 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 functions "over D". In their simplest form, the external external functions are *scalar*, i.e. of type $D^n \rightarrow D$, as in [13], but we allow external functions of any types, i.e. $P_j : dom(d_j, D) \rightarrow dom(c_j, D)$, where d_j and c_j are arbitrary types called the **domain** and **codomain** of P_j. A database schema will have then the form $\sigma = (d_1 \rightarrow c_1, \ldots, d_l \rightarrow c_l; t_1, \ldots, t_k)$. E.g. consider the database schema $\sigma = (\{d\} \rightarrow d; d \times d \times d)$. A database instance over σ is $\mathcal{D} = (D; P; R)$, where $P : \mathcal{P}_{\mathit{fin}}(D) \rightarrow D$. The relation R can be thought of as containing tuples for persons, with three columns: $SS\#$, $NAME$ and AGE. The function P applied to some set S of social security numbers generates a new $SS\#$ which is not in S, that is $P(S) \not\in S$, $\forall S$. Obviously a query over that database may not necessarily be domain independent in the traditional sense, because it has the ability of constructing new social security numbers by calling the function P. The first goal of this paper is to investigate the notion of domain independence of queries with external functions.

Traditionally external functions have been thougth of as *fixed* functions on the universal domain of the database. We give them a broader interpretation here by viewing them as library functions, subject to changes in time. E.g. let $P : D \times D \rightarrow D$, be a library function expecting an employee's name and salary, which increases its salary by a quantum. P may incorporate complex knowledge on the company's policy, and may change in time, as the company changes its policy. The following is an example of a query using P: "increase by one quanta the salaries in the *sales* department, by two quanta those in the *business* department, and leave the rest unchanged".

[5], and later [13], present an extension of the notion of domain independence for databases with external functions. Strictly speaking, the *embedded domain independence* of [13] implies the *bounded-depth domain independence* of [5] , but they rely on the same idea. Both notions are used only in conjunction with query languages without recursive queries (or any other kind of iterations), and, as we show in this paper, fail when extended to languages with fixpoints. See example 4.1 for a fixpoint query which is not embedded domain independent. In this paper we introduce a new notion, called *external-function domain independence* (ef-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 ef-domain independent.

Our second goal in this paper is to investigate computability of queries in the presence of external functions: we have no knowledge of any previous attempt to define computable queries in the presence of external functions. One way of understanding computable queries is to view external function as oracles: at any point during the computation of a query F, the device computing F may ask the oracle corresponding to some external function P_j for the value of $P_j(x)$, for some x of type d_j: after receiving the answer $y = P_j(x)$, the device may proceed. Note that the active domain, which initially contains all atomic values in $R_1, \ldots, R_k$, is extended dynamically, because the oracles may generate new atomic values. Another way of viewing the external functions, is to restrict them to *computable* functions; then we can encode a computable function as a finite string, e.g. as some program computing that function, or as the Gödel number of the Turing Machine corresponding to that function. The two views give rise to two notions of *computable queries*, and theorem 7.2 shows that they coincide over databases with *total* external functions.

Previous work [24, 5, 13] 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 [13] can be extended to a higher order logic with fixpoints, such that all "embedded allowed" formulas define an ef-domain independent, computable query. We intend to investigate this direction in future work.

3 Domain Independent Queries with External Functions

Before giving the formal definition, we argue for the necessity of considering *partial* external functions, as opposed to *total* ones. The *active domain* of some database instance $\mathcal{D}$ is the set of all atomic values mentioned in its relations. The active domain is always finite, although in this paper the domain may be infinite. Restricting the database domain to the active domain leads naturally to *partial external functions.* Formally, we define:

Definition 3.1 *A* **database schema** *with external functions is* $\sigma = (d_1 \to c_1, \ldots, d_l \to c_l; t_1, \ldots, t_k)$*;* $d_1, c_1, \ldots, d_l, c_l, t_1, \ldots, t_k$ *are types. A* **database instance** *over* σ *is* $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$*, where* P_i *is a partial function* $P_i : dom(d_i, D) \to dom(c_i, D)$*, and* R_i *is a finite subset of* $dom(t_i, D)$*.* $\mathcal{D}$ *is called* **total** *iff all functions* P_i *are total, otherwise it is called* **partial**.

Next we will define a *morphism* $\psi : \mathcal{D} \to \mathcal{D}'$ to be a partial, injective function $\psi : D \to D'$ between the domains of two databases, which "preserves the structure" of theses databases, in a sense to be made precise. For that, we notice that any partial function $\psi : D \to D'$ can be lifted from the base type to partial functions at any type t, $\psi_t : dom(t, D) \to dom(t, D')$. Namely $\psi_d \stackrel{\text{def}}{=} \psi$, $\psi_{t_1 \times \ldots \times t_n}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} (\psi_{t_1}(x_1), \ldots, \psi_{t_n}(x_n))$, and $\psi_{\{t\}}(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} \{\psi_t(x_1), \ldots, \psi_t(x_n)\}$. In all cases, $\psi_t(x)$ is undefined whenever one of the subexpressions on the right hand side is undefined. We abbreviate ψ_t with ψ.

Definition 3.2 *Let* σ *be some database schema, and* $\mathcal{D} = (D; \bar{P}; \bar{R})$*,* $\mathcal{D}' = (D'; \bar{P}'; \bar{R}')$ *be two database instances over* σ*. A* **morphism** $\psi : \mathcal{D} \to \mathcal{D}'$ *is a partial injective function* $\psi : D \to D'$*, such that (1) for every* i*,* $\psi(R_i)$ *is defined and* $\psi(R_i) = R'_i$*, and (2) for any* $x \in dom(d_j, D)$*, if* $P'_j(\psi(x))$ *is defined then so is* $\psi(P_j(x))$ *and* $P'_j(\psi(x)) = \psi(P_j(x))$*.*

We mention that, in the particular case in which all external functions are scalar, the database instances correspond to *partial algebras* of [14] and the total morphisms are precisely the *homomorphism of partial algebras* of [14].

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, ψ is a morphism iff $\psi(R_i) = R'_i$ for all i, and $P'_j \circ \psi \sqsubseteq \psi \circ P_j$ for all j (to be precise, $P'_j \circ \psi_{d_j} \sqsubseteq \psi_{c_j} \circ P_j$, but recall that we drop the type t from ψ_t).

Definition 3.3 *Let σ be a database schema and t some type. A* **database query** *from σ to t is a partial function F mapping any database instance over σ $\mathcal{D} = (D; \bar{P}; \bar{R})$ to $F(\mathcal{D}) \in dom(\{t\}, D)$. F is* **external-function domain independent**, *or* **ef-domain independent**, *iff for every morphism $\psi : \mathcal{D} \to \mathcal{D}'$, $F(\mathcal{D}') \sqsubseteq \psi(F(\mathcal{D}))$.*

That is, whenever $F(\mathcal{D}')$ is defined, $F(\mathcal{D})$ must be defined too, $\psi(F(\mathcal{D}))$ must also be defined, and $F(\mathcal{D}') = \psi(F(\mathcal{D}))$.

Notice that this notion generalizes those of generic and domain independent queries on databases without external functions. Indeed, observe that an isomorphism of database instances is, in particular, a morphism. Also, remark that a function of type $unit \to t$ can be assimilated with a constant of type t. Then the following is easy to check:

Proposition 3.4 *Suppose that the database schema σ doesn't contain any external functions (i.e. $l = 0$). Then a query is ef-domain independent iff it is generic and domain independent. Also, when σ only contains atomic constants (i.e. functions of type unit $\to d$), then a query is ef-domain independent iff it is C-generic [20] and domain independent.*

Next we look at how an ef-domain independent query behaves on 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 $j = 1, l$. Whenever $\mathcal{D} \sqsubseteq \mathcal{D}'$ there is a canonical morphism $\psi : \mathcal{D}' \to \mathcal{D}$ defined by:

$$\psi(x) = \begin{cases} x & \text{when } x \in D \\ \text{undefined} & \text{otherwise} \end{cases}$$

ψ has the property: $\forall x, \psi(x) \sqsubseteq x$. (Note however that the inclusion function $D \to D'$ is usually not a morphism.) Define a query F to be **monotone** if $\mathcal{D} \sqsubseteq \mathcal{D}'$ implies $F(\mathcal{D}) \sqsubseteq F(\mathcal{D}')$. For any ef-domain independent query F, databases $\mathcal{D} \sqsubseteq \mathcal{D}'$, and canonical $\psi : D' \to D$ we have $F(\mathcal{D}) \sqsubseteq \psi(F(\mathcal{D}')) \sqsubseteq F(\mathcal{D}')$, which proves:

Proposition 3.5 *Any ef-domain independent query is monotone.*

Next we will connect the notion of ef-domain independent query with that of *embedded domain independent query* defined in [13]. For this, following [13], we define $term^n(\mathcal{D})$, for some database $\mathcal{D}$ and $n \geq 0$, as follows:

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

where $atoms(R)$ are all values in D mentioned in the relation R. Two databases $\mathcal{D} = (D; \bar{P}; \bar{R})$ and $\mathcal{D}' = (D'; \bar{P}'; \bar{R})$ (note that they have the same relations) are said to *agree to level* n [13] iff (1) $term^{n+1}(\mathcal{D}) = term^{n+1}(\mathcal{D}')$, and (2) for any j, P_j and P'_j agree on any input whose atoms are in $term^n(\mathcal{D})$, i.e. $\forall x \in dom(d_j, term^n(\mathcal{D}))$, $P_j(x) = P'_j(x)$. A query F is called **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 call F **em-domain independent**, if there is some n for which F is em-domain independent at level n (this definition extends the notion of em-domain independence [13] to complex objects and non-scalar external functions).

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 fixpoints or loops, like those considered in [5, 13], but fails once an iterative construct (like fixpoints) is added to the language (see example 4.1). For iterative queries, the number n of applications of the external functions is still finite, but may depend on the particular relations $R_1, \ldots, R_k$. 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 for any database instance $\mathcal{D}$ 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.

Finally, let us call some query F **continuous** if for any database instance $\mathcal{D}$ for which $F(\mathcal{D})$ is defined, there is 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 use of the term "continuous" here is consistent with that of *continuous functions on algebraic cpo's*, see e.g. [21, 16]. Obviously all domain independent queries without external functions are continuous, since it suffices to take D_0 to be the active domain, which is finite. We also have:

Proposition 3.6 *Any sem-domain independent query is continuous. Hence, any em-domain independent query is continuous too.*

Now we can establish the relationship between our notion of ef-domain independence (definition 3.3) and that of em-domain independence of [13].

Theorem 3.7 *A query F is ef-domain independent and continuous iff it is sem-domain independent and monotone.*

The proof is given in appendix A. On the other hand, ef-domain independence does not imply continuity, as the following example shows. Consider the database schema $\sigma = (d \to d; d)$, and let F be the query:

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

where $\mathcal{D} = (D; P; R)$. This query is ef-domain independent, but it is not continuous.

Certainly, we would expect all queries expressed in a query language with external functions to be continuous: we shall prove indeed in the next section that all *computable* queries are continuous. Hence, we argue that continuity is connected to the property of a query being computable, and should be orthogonal to the notion of domain independence.

The notion of *bounded-depth domain independence* of [5] extends that of em-domain independence by allowing the computation of *inverses* of external functions, that is $P^{-1}(x)$, for P an external function: the two coincide when the set of external functions is closed under inverses.

4 A Language

Let Σ be a **signature**, that is $\Sigma = \{p_1, \ldots, p_l\}$ is a set of l symbols, each symbol p_j having associated two types called *the domain* d_j and *the codomain* c_j, written $p_j : d_j \to c_j$: we call $p_1, \ldots, p_l$ *external functions.* We defined briefly the Nested Relational Algebra over Σ, $\mathcal{NRA}(\Sigma)$, following the formalism in [8], as an algebra of functions. Namely $\mathcal{NRA}(\Sigma)$ contains: all external functions $p_j : d_j \to c_j$ in Σ, the identity functions $id_t : t \to t$, the composition of functions in $\mathcal{NRA}(\Sigma)$, $g \circ f : t_1 \to t_3$ (for $f : t_1 \to t_2$ and $g : t_2 \to t_3$ in $\mathcal{NRA}(\Sigma)$), the projections $\pi_i^n : t_1 \times \ldots \times t_n \to t_i$, n-tuples of functions $(f_1, \ldots, f_n) : t \to t_1 \times \ldots \times t_n$ (for $f_i : t \to t_i$, $i = 1, n$ in $\mathcal{NRA}(\Sigma)$), the empty set $\emptyset : unit \to \{t\}$, the singleton $\eta : t \to \{t\}$, the flattening function $\mu : \{\{t\}\} \to \{t\}$, union $\cup : \{t\} \times \{t\} \to \{t\}$, *map* of any function f in $\mathcal{NRA}(\Sigma)$, $map(f) : \{t\} \to \{t'\}$ (for every $f : t \to t'$), equality at base type $eq : d \times d \to \{unit\}$, and negation $not : \{unit\} \to \{unit\}$. The semantics of *map* is: $map(f)(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} \{f(x_1), \ldots, f(x_n)\}$. We refer the reader to [8] for full details of this language.

Each function $f : \{t_1\} \times \ldots \times \{t_k\} \to \{t\}$ in $\mathcal{NRA}(\Sigma)$ defines some query F, which on a database instance $\mathcal{D} = (D; P_1, \ldots, P_l; R_1, \ldots, R_k)$ computes the relation $F(\mathcal{D}) \stackrel{\text{def}}{=} f(R_1, \ldots, R_k)$. $\mathcal{NRA}(\Sigma)$ is essentially equivalent to Abiteboul and Beeri's *extended algebra without powerset* [5] with external functions $p_1, \ldots, p_l$.

Next we add fixpoints to the language, namely $\mathit{fix}(f) : t \to \{t'\}$ whenever $f : t \times \{t'\} \to \{t'\}$, with inflationary semantics: $\mathit{fix}(f)(x) = \bigcup_{n \geq 0} y_n$, where $y_0 \stackrel{\text{def}}{=} \emptyset$, $y_{n+1} \stackrel{\text{def}}{=} y_n \cup f(x, y_n)$ ($\mathit{fix}(f)(x)$ is undefined when $\bigcup y_n$ is infinite). See [15, 17, 22] for fixpoints on complex objects. We denote with $\mathcal{NRA}(\Sigma) + \mathit{fix}$ the extension of $\mathcal{NRA}(\Sigma)$ with the fipxoint construct. 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 4.1 *Consider $\Sigma = \{p\}$, where $p : d \to d$ is some unary external function, and let $f : \{d\} \to \{d\}$ be the query $f(x) = \mathit{fix}(\lambda(x, y).x \cup map(p)(y))(x)$[1]. That is, $f(x)$ applies repeatedly p to all elements of x, until no new element is generated. If the set of all generated elements is finite, then*

[1] We use a more liberal notation of queries in $\mathcal{NRA}(\Sigma) + \mathit{fix}$ with variables. See [8] for a discussion.

$f(x)$ 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 [5]).

However it is easy to prove the following:

Proposition 4.2 *All queries in $\mathcal{NRA}(\Sigma) + fix$ are ef-domain independent and continuous. Also, queries expressed with other forms of iterations, like loop of [18], the structural recursions sru, sri of [7, 8], and the divide and conquer recursion dcr of [23] are also ef-domain independent and continuous.*

We take the above proposition as evidence that the notion of ef-domain independence is more appropriate for queries with external functions than the notions of em-domain independence or bounded-depth domain independence.

5 Computable Queries

A database query F on databases without external functions is called *decidable* iff there is some Turing Machine T which, whenever presented with an encoding of an input structure $\mathcal{D}$, computes an encoding of $F(\mathcal{D})$ (and diverges when $F(\mathcal{D})$ is undefined). We will restrict ourselves for the remaining of this paper to database instances with countable domain and with some fixed enumeration of their domain.

Once we admit external functions as part of the database, there are two ways of presenting them as input to T:

1. Require all external functions to be *Turing computable*, i.e. *recursive* [19] (as number theoretic functions), and replace each function P_j by a number e_j which represents the Gödel number of the Turing Machine computing P_j [19]. Thus, T expects as input encodings for $R_1, \ldots, R_k$, as well as l numbers $e_1, \ldots, e_l$, and computes an encoding of $F(\mathcal{D})$.
2. Extend the Turing Machine T with oracles [19], one for each function P_j. Now T will be started only with the encoding of $R_1, \ldots, R_k$ on its tape, but will be allowed to inquire any of its l oracles during the computation.

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.

$\mathcal{D} = (D; \bar{P}; \bar{R})$ is a *computable database instance* iff all external functions $P_1, \ldots, P_l$ are computable.

Definition 5.1 *A query F is computable iff there is some Turing Machine T such that for any any computable structure $\mathcal{D}$, when T is started with an encoding of $R_1, \ldots, R_k$ and with the Gödel numbers $e_1, \ldots, e_l$ 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 [19].

Lemma 5.2 *Let $f : \mathbb{N} \to \mathbb{N}$ be some recursive function with the property $\varphi_e \sqsubseteq \varphi_{e'} \Rightarrow \varphi_{f(e)} \sqsubseteq \varphi_{f(e')}$. Then $\forall e, \forall x$, if $\varphi_{f(e)}(x)$ is defined, then there is some e_0 such that φ_{e_0} is a finite function, $\varphi_{e_0} \sqsubseteq \varphi_e$, and $\varphi_{f(e_0)}(x) = \varphi_{f(e)}(x)$.*

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

Corollary 5.3 *All computable, ef-domain independent queries are continuous.*

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

Definition 5.4 *Let $\mathcal{C}$ be a class of database instances. A query language $\mathcal{L}$ with external functions from a set Σ is* **complete** *w.r.t. Σ over $\mathcal{C}$ iff it can express all computable, ef-domain independent queries over total databases from $\mathcal{C}$.*

The reason for which we require L to be able to express queries over total databases is due to the fact that only in this case do we have a robust notion of computable queries, i.e. the computable queries coincide with the RMC-computable queries, to be defined in the next section.

6 Relational Machines for Complex Objects

The second notion of computable queries is based on a variant of Turing Machines with oracles. In [19], 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 [6] introduce the notion of *loose Generic Machine*, later simplified to *Relational Machines* in [4]. In some sense, these 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 some signature $\Sigma = \{p_1, \ldots, p_l\}$ of function symbols. A RMC M over Σ is a Turing Machine extended with a fixed number of relational registers, $R_0, R_1, \ldots, R_r$. At each step, M may perform some traditional Turing Machine move, or may affect the relational store in one of the following two ways: (1) it may inspect the content of some register R_i and enter one of two different states, depending on whether R_i is empty or not; we call this a *conditional*, or (2) it may replace the content of some register R_i with $h(R_{i_1}, \ldots, R_{i_k})$, where h is a query in the language $\mathcal{NRA}(\Sigma)$; we call this *an assignment*. 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 gets stuck.

The registers of a RMC are typed, i.e. only values of some type t_i may be stored in R_i, and all RMC's are required to be deterministic.

A RMC computes some database query F as follows: for some database instance $\mathcal{D}$, its k relations are placed in the registers $R_1, \ldots, R_k$ of the RMC, and the machine is started with an empty tape. When (and if) it stops, the result $F(\mathcal{D})$ is in R_0.

Definition 6.1 *Some query F is called* **RMC-computable** *iff there is some RMC, M, computing F.*

Proposition 6.2 *Any RMC-computable query F is ef-domain independent and continuous.*

As opposed to Relational Machines for flat relations, those for Complex Objects are *complete*, i.e. they can express all computable queries. The difference stems from the ability of a RMC to simulate parallel computations through the use of complex objects. This is a corollary of theorem 7.2 (see corollary 7.3), but below we sketch a shorter proof using the following lemma, which is also a key technical tool for theorem 7.2.

Lemma 6.3 (The Map Lemma) *Let M be some RMC computing the function $F : t \to t'$. Then there is some RMC M' computing $map(F) : \{t\} \to \{t'\}$.*

A sketch of the proof is given in appendix A. 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 [6]. Finally, we use the map lemma to simulate T on all $n!$ orders.

7 Computable Queries Coincide with RMC-Computable Queries on Total Databases

We shall assume in this section that all external functions are computable.

Proposition 7.1 *Any RMC-computable query is computable.*

The proof is straightforward, since a RMC can be simulated by a Turing Machine T, provided that T has access to the encodings of the external functions in Σ. The other direction is more involved, and only holds in the case of total external functions.

Theorem 7.2 *Over databases with total external functions, an ef-domain independent query is computable iff it is RMC-computable.*

The proof essentially consists in simulating some Turing Machine T by a Relational Machine for Complex Objects M. The difficulty is that M doesn't have the Gödel numbers $e_1, \ldots, e_l$ for the external functions $R_1, \ldots, R_l$ which T requires: instead, the external functions are hard coded into the Relational Machine M. Thus M has to search for $e_1, \ldots, e_l$, by systematically enumerating the Gödel numbers of all *finite* functions, and by "comparing" their input-output behavior to that of its oracle. The proof is given in appendix A.

In the absence of external functions, the theorem implies:

Corollary 7.3 *Relational Machines are complete for complex objects.*

If we drop the restriction to total databases, then the two notions of computable queries no longer coincide. What distinguish them is the fact that the RMC-computable queries are *sequential*, in a sense related to the notion of *sequential function* in [11], while computable queries need not be.

Definition 7.4 *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) For any $\mathcal{D}'$ s.t. $\mathcal{D} \sqsubseteq \mathcal{D}'$, $F(\mathcal{D}')$ is undefined, or (2) $\exists i, \exists x \in dom(d_i, D)$ such that for all $\mathcal{D}' \sqsupseteq \mathcal{D}$, if $F(\mathcal{D}')$ is defined then $P_i(x)$ is defined. We call the pair (i, x) the* sequentiality index *of F at $\mathcal{D}$ [11].*

Thus, F is sequential iff it invokes the external functions one at a time: if it gets stuck during the computation on some partial database $\mathcal{D}$ because the external functions are not defined, then there is a certain function P_i and a certain input x to P_i such that F gets stuck while trying to compute $P_i(x)$. 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 7.5 *All RMC-computable queries are sequential.*

But the following is an example of a computable, ef-domain independent query which is not sequential:

Example 7.6 *Consider the schema $\sigma = (d \to d; d)$, and the following query F:*

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

where $\mathcal{D} = (D; P; R)$. This query is ef-domain independent, and computable. To see that it is computable, suppose $R = \{x_1, \ldots, x_n\}$; a Turing Machine T can perform in parallel the computation steps for $P(x_1), \ldots, P(x_n)$, and stop when one of these computations, say for $P(x_i)$, finishes with $P(x_i) = x_i$. Thus T will not get stuck when some other computation, say for $P(x_j)$, never terminates. However this query is not RMC-computable because it is not sequential. Indeed, consider the partial database in which $R = \{x_1, x_2\}$ and $P(x_1) = P(x_2) =$ 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 $P'(x_1) \stackrel{\text{def}}{=} x_1$ or by defining $P'(x_2) \stackrel{\text{def}}{=} x_2$.

8 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. Let $\mathcal{L}$ be $\mathcal{NRA}(\Sigma) + \mathit{fix}$, n be some of its types, z a constant of type n, and s a function of type $s : n \to n$. Let $\mathcal{C}$ be a class of databases in which the elements $z, s(z), s^{(2)}(z), \ldots, s^{(k)}(z), \ldots$ are distinct. Then we have:

Proposition 8.1 *Any such language $\mathcal{L}$ is complete w.r.t. the class $\mathcal{C}$.*

Proof. (Sketch) Represent the naturals as $\mathbb{N} \stackrel{\text{def}}{=} \{z, s(z), s^{(2)}(z), \ldots\}$. First note that, at the number theoretic level, $\mathcal{L}$ can express all computable functions, in the following sense: if $f : \mathbb{N} \to \mathbb{N}$ is a recursive function, then $F : \mathbb{N} \to \mathbb{N}$ defined by $F(x) \stackrel{\text{def}}{=} \{s^{(f(x))}(z)\}$ is expressible in $\mathcal{L}$ [22]. To see that, we prove that the class of functions f for which the corresponding F is definable in the language is closed under minimization. Consider some predicate $p : n \to n$ in $\mathcal{NRA}(\Sigma) + \mathit{fix}$. To compute the partial function $F(x) = \{s^{(k)}(z) \mid k \text{ is the smallest s.t. } p(s^{(k)}(z))\}$, compute successively all sets $\{z, s(z), \ldots, s^{(k)}(z)\}$, $k = 0, 1, 2, \ldots$, until p is true on at least one element of the set; this can be expressed with a fixpoint. Next, select the "largest" element of the set. Similarly we prove that the class of functions f is closed under primitive recursion. In view of theorem 7.2 it suffices to prove that $\mathcal{L}$ can express any RMC-computable query. This is indeed the case, because $\mathcal{L}$ can simulate the computations of a RMC. Indeed a configuration of a RMC with $r+1$ registers $R_0, \ldots, R_r$ can be represented as an $r+4$ tuple: the first $r+1$ components describe the content of the registers, the other three components describe the current state, the head position, and the tape. The latter is a set of pairs (i, c), where i and c are "numbers" (i.e. objects of type n) denoting the fact that cell i contains the character c. The successor relation on configurations is expressible in the language: it consists in doing some arithmetic to deal with the next and previous cell, and some operations on the registers, which are expressible in the language by the definition of a RMC. Finally one has to iterate successor function until a final state is reached; a *partial fixpoint*, as opposed to an invlationary fixpoint is needed here, but the partial fixpoint can be expressed via an inflationary fixpoint using an additional set level, see [22]. $\square$

It follows that the following languages are complete:

Object Inventions Consider some base type ι whose elements are called *object id's*, and some external function *make_object* : $\{\iota\} \to \iota$ which "generates" new id's: more precisely, we consider $\mathcal{C}$ to be the class of databases for which $make\_object(x) \notin x$, for all x of type $\{\iota\}$. Intuitively $make\_object(x)$ generates an id which was not present in the set x. 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 we allow for more than one base sort, see section 3). This language satisfies the requirements of proposition 8.1, by taking $n \stackrel{\text{def}}{=} \{\iota\}$, $z \stackrel{\text{def}}{=} \emptyset$ and $s(x) \stackrel{\text{def}}{=} x \cup \{make\_object(x)\}$. It has been known previously that object inventions in conjunction with fixpoints give rise to complete query languages [2]. 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 of the *untyped sets* in [20]. 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 other base types in $\mathcal{L}$, and from other elements in u. E.g. $x = \{a, \{b, c\}, \{a, \emptyset, \{\{b\}\}\}\}$ is a legal element of u, provided that a, b, c are atomic elements. In particular, all elements of type $\{u\}$ are also of type u, and we consider some external function *include* : $\{u\} \to u$ to witness that inclusion of types. As any base type, u has an equality operator defined on it. Then $\mathcal{L}$ is complete w.r.t. to $\mathcal{C}$. Indeed, it suffices to take $n \stackrel{\text{def}}{=} u$, $z \stackrel{\text{def}}{=} included(\emptyset)$, and $s(x) \stackrel{\text{def}}{=} include(\{x\})$ in proposition 8.1. That is, the naturals are represented by the set $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\{\{\emptyset\}\}\}, \ldots\}$.

Natural Numbers Consider $\mathbb{N}$ to be one of the base types, and $0, 1, +$ to be among the functions in Σ, and let $\mathcal{C}$ be the class of databases in which $\mathbb{N}, 0, 1, +$ have the standard interpretation. The resulting $\mathcal{L}$ is a complete query language w.r.t. Σ for $\mathcal{C}$: take $z \stackrel{\text{def}}{=} 0$ and $s(x) \stackrel{\text{def}}{=} x + 1$ in proposition 8.1.

9 Conclusions

We have investigated the computability of queries in the presence of external functions. Our techniques do not extend straightforwardly to an investigation of the complexity of queries. E.g. we could define some query F to be in PSPACE either when it is computed by some PSAPCE Turing Machine expecting both an encoding of the 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 PSPACE query might be. We intend to address the complexity issues for queries with external functions in the future.

10 Acknowledgments

We thank Victor Vianu, Serge Abiteboul, Catriel Beeri and Rick Hull for commenting on an earlier version of this paper, and the anonymous reviewers for their suggestions. The author was supported by NSF grant CCR-90-57570.

References

[1] S. Abiteboul, S. Grumbach, A. Voisard, and E. Waller. An extensible rule-based language with complex objects and data-functions. In *Proceedings of 2nd Workshop on Database Programming Languages*, Gleneden Beach, Oregon, June 1989.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

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

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

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

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

[13] 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.

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

[15] 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.

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

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

[18] Marc Gyssens and Dirk Van Gucht. The powerset algebra as a natural tool to handle nested database relations. *Journal of Computer and System Sciences*, 45:76–103, 1992.

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

[20] 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.

[21] 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`.

[22] 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.

[23] 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.

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

A Proofs

Theorem C.7 *A query F is ef-domain independent and continuous iff it is sem-domain independent and monotone.*

Proof. Let F be a monotone, sem-domain independent query and $\psi : \mathcal{D} \to \mathcal{D}'$ be a morphism. 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 instance in which: (1) the domain is $term^{n+1}(\mathcal{D}')$, (2) the relations are the same as in $\mathcal{D}'$, (3) the external functions are those of $\mathcal{D}'$ restricted to $term^n(\mathcal{D})'$. Then $F((\mathcal{D}')^{(n)}) = F(\mathcal{D}')$ because $(\mathcal{D}')^{(n)}$ and $\mathcal{D}'$ coincide up to level n. Similarly we define $\mathcal{D}^{(n)}$. Let $\psi(\mathcal{D}^{(n)})$ be the image of the database $\mathcal{D}^{(n)}$ under ψ, that is its domain is $\psi(term^{n+1}(\mathcal{D}))$, its relations are $\psi(R_i)$, and the graphs of its external functions are the images of the graphs of P_j under ψ. Then $(\mathcal{D}')^{(n)}$ is an approximation of $\psi(\mathcal{D}^{(n)})$, i.e. $(\mathcal{D}')^{(n)} \sqsubseteq \psi(\mathcal{D}^{(n)})$, because ψ is a morphism. By monotonicity, we have $F((\mathcal{D}')^{(n)}) \sqsubseteq F(\psi(\mathcal{D}^{(n)}))$. Since we only consider queries which map isomorphic databases into isomorphic outputs, we also have $F(\psi(\mathcal{D}^{(n)})) = \psi(F(\mathcal{D}^{(n)}))$. Putting everything together, we have $F(\mathcal{D}') = F((\mathcal{D}')^{(n)}) \sqsubseteq F(\psi(\mathcal{D}^{(n)})) = \psi(F(\mathcal{D}^{(n)})) \sqsubseteq \psi(F(\mathcal{D}))$.

Conversely, let F be an ef-domain independent, continuous query; by proposition 3.5 it suffices to show that F is sem-domain independent. For some database $\mathcal{D} = (D; \bar{P}; \bar{R})$ on which F is defined, consider its approximation $\mathcal{D}_u = (D_u; \bar{P}_u; \bar{R})$, where $D_u = \bigcup_{n \geq 0} term^n(\mathcal{D})$, and $(P_j)_u$ is the restriction of P_j to D_u. Then one can check that the inclusion function $\psi : D_u \to D$ is a morphism. Hence $F(\mathcal{D}) \sqsubseteq F(\mathcal{D}_u)$, that is $F(\mathcal{D}) = F(\mathcal{D}_u)$, because the left hand side is defined. But now F being continuous, there is some finite approximation $\mathcal{D}_u^0$ of $\mathcal{D}_u$, such that $F(\mathcal{D}_u) = F(\mathcal{D}_u^0)$. Now to each atom x in D_u we associate a number n, called its *order*, which is the smallest one with the property $x \in term^n(\mathcal{D})$. Let n be the highest order of all atoms in the finite database $\mathcal{D}_u^0$. One can verify that the level of F at $\mathcal{D}$ is at most n.

□

Lemma E.2 *Let $f : \mathbb{N} \to \mathbb{N}$ be some recursive function with the property $\varphi_e \sqsubseteq \varphi_{e'} \Rightarrow \varphi_{f(e)} \sqsubseteq \varphi_{f(e')}$. Then $\forall e, \forall x$, if $\varphi_{f(e)}(x)$ is defined, then there is some e_0 such that φ_{e_0} is a finite function, $\varphi_{e_0} \sqsubseteq \varphi_e$, and $\varphi_{f(e_0)}(x) = \varphi_{f(e)}(x)$.*

Proof. Suppose that for all e_0 for which $\varphi_{e_0} \sqsubseteq \varphi_e$ and φ_{e_0} is finite, $\varphi_{f(e_0)}(x)$ is undefined. Then, we give a semidecision procedure for $\bar{K}$ (where $K = \{z \,/ \varphi_z(z) \downarrow\}$), which is a contradiction. Indeed, let $k(z)$ be defined by: $\varphi_{k(z)}(y) = (if\ \varphi_z^y(z) \uparrow then\ \varphi_e(y)\ else \uparrow)$[2]. When $z \in \bar{K}$, then $\varphi_{k(z)} = \varphi_e$, and when $z \in K$, then $\varphi_{k(z)}$ is a finite restriction of φ_e. So $\varphi_{f(k(z))}(x) \downarrow$ iff $z \in \bar{K}$. This would imply that $\bar{K}$ is r.e., which is a contradiction. □

Lemma F.3 (The Map Lemma) *Let M be some RMC computing the functions $F : t \to t'$. Then there is some RMC M' computing $map(F) : \{t\} \to \{t'\}$.*

Proof. Suppose first that M has no conditionals. Then M' has the same instructions as M, except for the assignments $R_i \leftarrow h(R_{i_1}, \ldots, R_{i_k})$, which are replaced by $R'_i \leftarrow map(h)(R'_{i_1}, \ldots, R'_{i_k})$ (recall that h is in $\mathcal{NRA}(\Sigma)$, hence so is $map(h)$). That is, M' simulates n parallel computations of M on some input $\{x_1, \ldots, x_n\}$, in a synchronous way. But when M has conditionals, then the synchronism is no longer possible. Call the *trace* of some computation of M on input x_i, the sequence of 0 and 1 corresponding to the conditional instructions of that computation. Then M' generates on its tape all traces, and for each of them simulates in a synchronous parallel way the computation of M on those inputs x_i having that trace. M' stops when all inputs x_i have been processed. □

Theorem G.2 *Over databases with total external functions, an ef-domain independent query is computable iff it is RMC-computable.*

Proof. (Sketch) More precisely, we have to prove that for any computable, ef-domain independent query F there is some RMC-computable query F' such that F and F' coincide on total databases; the other direction is taken care of by proposition 7.1. 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 [19] $e_1, \ldots, e_l$ of the Turing Machines computing $P_1, \ldots, P_l$. We first describe a nondeterministic RMC M which computes F', and then explain how to transform M to become deterministic. M receives its inputs in $R_1, \ldots, R_k$, and starts by computing the active domain in R_{k+1}, say $R_{k+1} = \{o_1, \ldots, o_n\}$. Later, the active domain will be extended, i.e. $R_{k+1} = \{o_1, \ldots, o_m\}$, with $m \geq n$ (and $m = n$ initially). M uses R_{k+2} to keep a subset of all permutations of the active domain: initially, R_{k+2} contains all $n!$ permutations of $\{o_1, \ldots, o_n\}$. On the other hand, M keeps on its tape a finite approximation of $\mathcal{D}$, i.e. description of a finite database instance $\mathcal{D}^0 = (D^0; P_1^0, \ldots, P_l^0; R_1^0, \ldots, R_k^0)$, whose atoms are m numbers $D^0 = \{o_1, \ldots, o_m\} \subseteq \mathbb{N}$. Initially, $D^0 = \{0, 1, \ldots, n-1\}$, and $P_1^0, \ldots, P_l^0$ are totally undefined (the relations are not kept explicitly). Any permutation in R_{k+2} uniquely defines

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

a partial surjective order-preserving function $\psi : D \to D^0$, and M preserves the invariant that each such ψ be a morphism. Finally, M keeps a number s on its tape, initially $s = 0$.

Each step of M consists of two parts:

1. First M simulates T on the database instance $\mathcal{D}^0$ for s steps. If T halts, then M decodes the result (using the permutations in R_{k+2}), and halts too. Else M enters the second part.
2. Nondeterministically M chooses one of the following ways of extending $\mathcal{D}^0$ or s:
   - M increases s, or
   - M extends the active domain of $\mathcal{D}$. Namely M picks some external operation P_i and applies it to all possible inputs made up from the atoms in the current active domain $R_{k+1} = \{o_1, \ldots, o_m\}$ (it is important for P_i 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 picks nondeterministically $m' - m$ numbers which are not in D^0, and inserts them in D^0. Finally, M extends the permutations of $\{o_1, \ldots, o_m\}$ in R_{k+2} in all possible ways to permutations of $\{o_1, \ldots, o_{m'}\}$.
   - M extends some external function of $\mathcal{D}^0$. Namely M picks some operation P_i^0, some input x on which P_i^0 is undefined, and some output y, where both x, y are complex objects in the database instance $\mathcal{D}^0$. Next M extends P_i^0 by defining $P_i^0(x) \stackrel{\text{def}}{=} y$, and selects from R_{k+2} only those permutations which still correspond to a morphism ψ, i.e. which satisfy $R_i^0 \circ \psi \sqsubseteq \psi \circ R_i$; this can be tested since the left hand side is a finite function whose graph is accessible to M, and the right hand side is a total function. (If R_{k+2} becomes empty, then M fails.)

Obviously, if M halts then it correctly computes the output of T on (the encoding of) $\mathcal{D}$. We have to argue that M indeed halts, when T does. Let $\Psi : D \to \mathbb{N}$ to be the standard encoding of the domain of $\mathcal{D}$, and consider the set $\mathcal{S}$ of all finite databases $\mathcal{D}^0$ generated by M, for which the morphism $\psi : \mathcal{D} \to \mathcal{D}^0$ is included in Ψ (i.e. $\psi \sqsubseteq \Psi$). Let

$$\bar{D}^0 \stackrel{\text{def}}{=} \bigcup_{\mathcal{S}} D^0$$
$$\bar{P}_i^0 \stackrel{\text{def}}{=} \bigcup_{\mathcal{S}} P_i^0$$

i.e. the graph of $\bar{P}_i^0$ is the union of all graphs of P_i^0 of the finite databases in $\mathcal{S}$, and $\bar{D}^0 \subseteq \mathbb{N}$ is the set of all atoms in their domains. The database $\bar{\mathcal{D}}^0 \stackrel{\text{def}}{=} (\bar{D}^0; \bar{P}_1^0, \ldots, \bar{P}_l^0; R_1^0, \ldots, R_k^0)$ is the homeomorphic image of a certain approximation of $\mathcal{D}$. Namely of that approximation whose domain is obtained from the active domain of $\mathcal{D}$ by repeatedly applying the external functions of $\mathcal{D}$. Hence $\bar{P}_i^0$ is not necessarily total. It is easy to check that $\bar{P}_i^0$ is indeed a function, and one can even prove that it is computable, although we don't really need that. One can check that Ψ is a morphism from $\mathcal{D}$ to $\bar{\mathcal{D}}^0$, in fact the canonical morphism corresponding to the approximation $\bar{\mathcal{D}}^0$ of $\mathcal{D}$ (see section 3). But, surprising, $\Psi^{-1} : \bar{\mathcal{D}}^0 \to \mathcal{D}$ is a morphism too, because $\bar{\mathcal{D}}^0$ is "upwards closed". Indeed, for some $y \in dom(d_i, \bar{\mathcal{D}}^0)$, let $x = \Psi^{-1}(y)$. We have to check that $P_i(x) \sqsubseteq \Psi^{-1}(\bar{P}_i^0(y))$, which is equivalent to $\Psi(P_i(x)) \sqsubseteq \bar{P}_i^0(y)$. Since Ψ is a morphism, and because the left hand side is defined, it suffices to show that $\bar{P}_i^0(y)$ is defined. This is indeed the case, since M extends the external functions P_i^0 in all possible ways, so there in $\mathcal{S}$ is at least one database $\mathcal{D}^0$ for which $P_i^0(y)$ is defined.

This implies that T, run on the database $\bar{\mathcal{D}}^0$ halts. But then there is some finite approximation $\mathcal{D}^0$ of it, on which T halts too, and M will eventually find that approximation.

Finally, M can be made deterministic using a standard technique [10]. 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.

$\square$