Queries on Databases With User-Defined Functions

Dan Suciu

University of Pennsylvania
Queries on Databases With User-Defined Functions

Abstract
The notion of a database query is generalized for databases with user-defined functions. Then, we can prove that the computable queries coincide with those expressible by an extension of the relational machine, with oracles ([4]). This implies that any complete query language, extended with user-defined function symbols in a "reasonable" way, is still complete. We give an example of a complete query language with user-defined functions, and discuss its connections with object inventions.

Comments
Queries on Databases with User-Defined Functions

MS-CIS-93-62
LOGIC & COMPUTATION 70

Dan Suciu

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

June 1993
Queries on Databases with User-Defined Functions

Dan Suciu
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
Email: suciu@saul.cis.upenn.edu
June 15, 1993

Abstract
The notion of a database query is generalized for databases with user-defined functions. Then, we can prove that the computable queries coincide with those expressible by an extension of the relational machine ([4]), with oracles. This implies that any complete query language, extended with user-defined function symbols in a "reasonable" way, is still complete. We give an example of a complete query language with user-defined functions, and discuss its connections with object inventions.

1 Introduction
A query is usually defined to be a generic database transformation, i.e. a function mapping relations to relations, which is invariant under isomorphisms. This corresponds in practice to the requirement that a query should be independent of the representation of the data in a data base. A further property usually imposed upon queries is domain independence, meaning that the output relation depends only on the elements of the input relation(s), and not on the underlying univers.

When the underlying domain comes equipped with some user-defined functions, then more mapping from relations to relations deserve to be called generic database transformation. For example, suppose we may use the user-defined function invent_object : \{object\} \rightarrow object taking a set of objects and returning an object outside that set. Then we can define some database transformation new_objects : \{object\} \rightarrow \{object\}, which, for a given set of objects \(x\) with \(n\) elements, "invents" \(n\) new objects, namely \(\text{new} \_\text{objects}(x) = x_n\), where \(x_0 := x\), \(x_{i+1} := x_i \cup \{\text{invent} \_\text{object}(x_i)\}\). It is neither generic, nor domain independent
in the traditional sense, so we need to redefine what a query is, in the presence of user-defined functions. The queries relative to a given set $\Sigma$ of user-defined functions will be called $\Sigma$-queries.

Although the user-defined functions play in a database query language the same syntactic role played by the interpreted functions in [1], they have different semantic roles. The user-defined functions are not already interpreted, but they are available during the computation of a query to be interrogated as oracles. For example, if $x$ is (some part of) the input of a query, and $p$ is some user-defined function, then we may use $p(x), p^2(x), p^3(x), \ldots$ when computing the output of the query. But we may never use $p^{-1}(x)$, as allowed in [1] where $p$ is viewed as an interpreted function.

The $\Sigma$-queries described in this paper correspond to embedded domain independent queries considered in [13]. An embedded domain independent query corresponds is a $\Sigma$-query together with a bound on the number of possible applications of the functions in $\Sigma$. Since we consider query languages with fixpoints, we lift the restriction on this bound.

The definition of a computable query needs also a slight adjustment, in the presence of user-defined functions. The main result of this paper, is the fact that if some query language $L$ is complete in the absence of user-defined functions, then it will remain complete when enriched (in a reasonable way) with user-defined functions.

Technically, this result is expressed as a coincidence of two definitions of computable queries. The first, more liberal one, considers Turing machines with oracles for all user-defined functions in $\Sigma$. The loose generic machine in [8], or the relational machine in [4], can be naturally extended to accommodate user-defined functions. The second definition is more conservative, and defines a query $f$ to be computable iff it is computable when viewed as a function of its input and of the recursive indexes of the user-defined functions in $\Sigma$.

Query languages are complete, when they are powerful enough to express arithmetic, which is needed to simulate a Turing machine. As shown in [3, 5], query languages with object inventions are also complete, but in a slightly different sense, since the queries they express are not functions, but relations. This suggests a relationship between $\Sigma$-queries with arithmetics, and $\Sigma$-queries with object inventions, which we also investigate in this paper.

The paper is organized as follows. Section 2 contains the definition of $\Sigma$-queries. In sections 3 and 4 we discuss computable queries and give an example of a complete query language. We prove our main result in section 5, and present the connections with object identities in section 6.

\footnote{Unlike in the flat case, the relational machines for complex objects can express all computable queries.}
2 \( \Sigma \)-Queries. Definitions.

Traditionally, a database is a tuple \( x = (D, R_1, \ldots, R_k) \), with \( D \) some set called the **domain** of \( x \), and \( R_i \subseteq D^{a_i}, i = 1, k \), finite relations (see, e.g. [11]). The \( k \)-tuple \( a = (a_1, \ldots, a_k) \) is called the **database schema**, or the **type** of the database. A **database transformation** (or database query) \( f \) is some partial function, assigning to each database \( x = (D, R_1, \ldots, R_k) \) of a given type \( a \), some relation \( f(x) \subseteq D^n \). \( f \) is called:

**generic** if for any isomorphism \( \psi \) between two databases \( x = (D, R_1, \ldots, R_k) \) and \( x' = (D', R'_1, \ldots, R'_k) \) (i.e. \( \psi : D \rightarrow D', \psi(R_i) = R'_i, i = 1, k \)), we have \( \psi(f(x)) = f(x') \) ([11] calls this property **consistency**). C-generic queries, where \( C \) is a set of constants, are introduced in [17]: \( f \) is C-generic iff the above property holds for all isomorphisms \( \psi \) which preserve the constants in \( C \).

**domain independent** if, for any databases \( x, x' = (D, R_1, \ldots, R_k) \) and \( x' = (D', R_1, \ldots, R_k) \) (i.e. same relations, but a different domain), it is the case that \( f(x) = f(x') \).

Generalizing to complex objects, we define our type system to be relative to a set of **base types** \( b_1, b_2, \ldots \). A **type** is either some base type \( b \), or the unit type \( unit \), or a product type \( \sigma \times \tau \), or a set type \( \{ \} \) (where \( \sigma, \tau \) are types). A **structure** \( A \) is a family of sets indexed by the base types, \( A = \{A_b\}_{b \in \text{base types}} \). For some given structure \( A \), we can define the interpretation of types \( \sigma, [\sigma]_A \), by induction on the types: \( [b]_A := A_b, [\sigma \times \tau]_A := [\sigma]_A \times [\tau]_A \), and \( \{\} \) \( \in \mathbb{P}(\{\} \), \( \text{fin} \), \( [\sigma]_A \rightarrow [\tau]_B \). Any function between structures \( \psi : A \rightarrow B \) (i.e. \( \psi \) is a family of functions \( \psi_b : A_b \rightarrow B_b \), for any base type \( b \)), extends naturally at all types: \( \psi_\sigma : [\sigma]_A \rightarrow [\tau]_B \). In the sequel, we shall consider only functions \( \psi \) which are injective. The reason for that, is that db-query languages can express equality.

In this setting, a **database transformation**, or a query, \( f : \sigma \rightarrow \tau \) is a just a family of partial functions indexed by structures, \( f^A : [\sigma]_A \rightarrow [\tau]_A \), such that the following diagram commutes, for any injective \( \psi : A \rightarrow B \):
One can notice that, when $\sigma$ is a product of flat types\(^2\), and $\tau$ is a flat type, then a query coincides with a “traditional” generic and domain independent query.

Now we introduce additional structure on our database: user-defined operations (functions). Let $\Sigma$ be a set of user-defined function symbols. Each function symbol comes equipped with types for its domain and codomain.

**Definition 1** Let $\Sigma = \{p_1, \ldots, p_k\}$ be given; each $p_i$ is a function symbol, having the domain $\sigma_i$ and codomain $\tau_i$, and we write $p_i : \sigma_i \rightarrow \tau_i$. A $\Sigma$-structure is $A = (A, p_1^A, \ldots, p_k^A)$, where $A = \{A_b\}_{b \in b}$ base types is a family of sets, and $p_i^A : [\sigma]_A \rightarrow [\tau]_A$ is a partial function.

**Definition 2** $B$ is a substructure of $A$, $B \subseteq A$, iff $B \subseteq A$ and, $\forall i = 1, k$ $p_i^B \subseteq p_i^A$. Substructures are called a week subalgebras in [14].

We consider partial structures instead of total structures (algebras), because we want the database queries to depend on the functions $p_i^A$ in a monotone and continuous way.

$B$ is a full substructure of $A$, if, in addition, $\forall p \in \Sigma$, $p : \sigma \rightarrow \tau$, $\forall x \in [\sigma]_B$, if $p^A(x)$ is defined, then $p^B(x)$ is also defined (which implies $p^B(x) \in [\tau]_B$). Full substructures are called subalgebras in [14].

We want to generalize the notion of a query, in the presence of function symbols in $\Sigma$, and call it a $\Sigma$-query. When $\Sigma$ is empty, we want it to coincide with a generic and domain independent query: it may not distinguish between its input elements other than testing them for equality. When $\Sigma$ contains $k > 0$

---

\(^2\)A flat type is a type of the form $\sigma = \{b_1 \times \ldots \times b_k\}$
function symbols, then it may, in addition, apply repeatedly functions \( p \) from \( \Sigma \) to the values of its input, or to previous results, and thus compute new values.

To summarize, we require a \( \Sigma \)-query to be:

**generic** i.e. invariant under those isomorphisms \( \psi : A \rightarrow B \) which commute with all \( p \in \Sigma \).

**domain independent** i.e. it should depend only on the input values, and those obtained by repeatedly applying the functions in \( \Sigma \). The *embedded domain independent* queries of [13] are domain independent in this sense, but impose the additional upper bound \( i \geq 0 \) on the number of applications of the functions in \( \Sigma \).

**monotone** i.e. when \( \mathcal{B} \) is a substructure of \( \mathcal{A} \), then \( f^\mathcal{B} \subseteq f^\mathcal{A} \).

It will turn out that any query is also a \( \Sigma \)-query, and that in general there are \( \Sigma \)-queries which are not queries. When \( \Sigma = \emptyset \), the only “queries” over scalar types (i.e. not involving the set constructor) are combinations of projections, pairings and conditionals, like \( f : b^4 \rightarrow b \), \( f(x) = \text{if } \pi_1(x) = \pi_3(x) \text{ then } \pi_3(x) \text{ else } \pi_4(x) \); traditionally, such functions are not even called “queries”. However, when \( \Sigma \neq \emptyset \), there might be more interesting \( \Sigma \)-queries at scalar types.

**Example 1** Let \( \Sigma = \{ p \} \), with \( p : \{ b \} \rightarrow b \), and consider the class \( \mathcal{C} \) of structures \( A \) with the property \( p^A(x) \neq x \) for all \( x \). \( g \) can be viewed as an “object generator”. Then the following query \( f : \{ b \} \rightarrow \{ b \} \) invents \( n \) new objects, where \( n \) is the number of elements in its input: \( f(x) := x_n \), where \( x_0 := x, \ x_{i+1} := x_i \cup \{ p^A(x_i) \}, \ n = \text{card}(x) \).

**Example 2** Consider \( \Sigma \) with only one function symbol \( p : b \rightarrow b \), and lets restrict ourselves to \( \Sigma \)-structures \( (A, p^A) \) with \( p^A : A \rightarrow A \) injective. Consider \( g : b \rightarrow b \), which computes the inverse of \( p \) (i.e. \( g^A(x) := y \text{ if } p^A(y) = x \), and undefined if there is no such \( y \)). This function is not domain independent.

**Example 3** For the same signature \( \Sigma \), let \( h : b \times b \rightarrow \text{bool} \) the the following query: \( h^A(x, y) := \text{true} \text{ iff } \forall n \geq 0 \text{ such that } p(x), p^2(x), \ldots, p^n(x) \text{ are all defined} \text{ and } p^n(x) = y \; h^A(x, y) := \text{false} \text{ iff } \forall n \geq 0, p^n(x) \text{ is defined and } p^n(x) \neq y \; \text{and } h^A(x, y) := \text{undefined}, \text{ otherwise} \). This is a database transformation relative to \( \Sigma \). but not a *computable* one.

Consider two \( \Sigma \)-structures \( (A, p^A_1, \ldots, p^A_k) \) and \( (B, p^B_1, \ldots, p^B_k) \), and for each base type \( b \), a partial, injective, function \( \psi_b : A_b \rightarrow B_b \). \( \psi \) extends to a partial function at all types: \( \psi_{\Sigma} : [\sigma]_A \rightarrow [\sigma]_B \), by defining:

\[
\psi_{\sigma \times \tau}(x, y) := \begin{cases} 
(\psi_\sigma(x), \psi_\tau(x)) & \text{when } \psi_\sigma(x) \text{ and } \psi_\tau(x) \text{ are defined} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

\[
\psi_{\{\sigma\}}(\{x_1, \ldots, x_n\}) := \begin{cases} 
\{\psi_{\sigma}(x_1), \ldots, \psi_{\sigma}(x_n)\} & \text{when } \psi_{\sigma}(x_1), \ldots, \psi_{\sigma}(x_n) \text{ are defined} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]
Definition 3 \( \varphi \) is called a partial isomorphisms if, \( \forall p \in \Sigma \):

\[
\begin{array}{ccc}
[\sigma]_A & \overset{p^A}{\longrightarrow} & [\tau]_A \\
\downarrow \psi_{\sigma} & \subseteq & \downarrow \psi_{\tau} \\
[\sigma]_B & \overset{p^B}{\longrightarrow} & [\tau]_B
\end{array}
\]

i.e. the function \( \psi_{\sigma} \circ p^A \) is an extension of the function \( p^B \circ \psi_{\sigma} \subseteq \psi_{\tau} \circ p^A \). When \( \psi \) is total, then this notion coincides with that of a homomorphism in \( [\mathcal{I}] \).

The motivation behind the above definition lies in the following two particular cases of partial isomorphisms:

1. When \( \psi : A \to B \) is surjective, then it corresponds to a substructure \( B \subseteq A \). Conversely, for \( B \subseteq A \): take \( \psi \) to be the identity on \( B \).

2. Any full substructure \( A \subseteq B \) gives rise to some total partial isomorphism \( \psi : A \to B \), namely the inclusion. The converse, however, is not true: if \( \psi \) is total, then \( A \) can be viewed as a subset of \( B \), whose operations can be “more defined” then the restriction of the operations on \( B \).

In general, a partial isomorphism \( \psi : A \to B \) corresponds to a full substructure \( B_0 \subseteq B \) (namely \( \text{Im}(\psi) \)), which is (isomorphic to) a substructure of \( A \).

In the sequel, we shall restrict ourselves to a class \( C \) to \( \Sigma \)-structures, closed under substructures.

Definition 4 A \( \Sigma \)-query, of type \( \sigma \to \tau \), is a family of partial functions \( f^A : [\sigma]_A \to [\tau]_A \), indexed by the \( \Sigma \)-structures \( (A, p^A_1, \ldots, p^A_\kappa) \) in \( C \), such that, for any partial isomorphism \( \psi : A \to B \), the following holds:
Now we can easily check that any $\Sigma$-query $f$ is generic, domain independent and monotone. To see genericity, just pick $\psi$ to be an isomorphism. For domain independence, let $B \subseteq A$ be the full substructure of $A$ generated by all atoms mentioned in some input value $x$; then the inclusion $\psi : B \to A$ is a partial isomorphism and implies that $f^A(x) = f^B(x)$, which, essentially, gives us domain independence. For monotonicity, let $B \subseteq A$ be any substructure, and $\psi : A \to B$ the identity on $B$; it follows that $f^B \subseteq f^A$.

Consider again the three examples on page 5. $f$ is clearly a $\Sigma$-query. $g$ is not a $\Sigma$-query; indeed, let $(A, p^A)$ be some arbitrary $\Sigma$-structure for which $p^A$ is injective, having at least some $y \in A$, for which $p^A(y) \neq y$. Let $x := p^A(y)$, and consider $B := \{x, p^A(x), p^A(p^A(x)), \ldots\}$. By defining $s^B$ to be the restriction of $p^A$ to $B$, we get a full substructure $B \subseteq A$, so the inclusion $\psi : B \to A$ is a partial isomorphism. But then $g^A(\psi(x)) = g^A(x) = y$ is defined, while $g^B(x)$ is undefined. So $g$ is not a $\Sigma$-query.

The query from the third example, $h$, is a $\Sigma$-query. Indeed, let $\psi : A \to B$ be any partial isomorphism, and suppose $x, y \in A$, and $h^B(\psi(x), \psi(y))$ is defined. We have to consider three cases. When $h^B(\psi(x), \psi(y)) = \text{false}$, then $s^B(\psi(x)), s^B(\psi(x)), \ldots$ are all defined, and different from $\psi(y)$. But then $p^A(x), p^A(p^A(x)), \ldots$ are also defined and different from $y$, so $h^A(x, y) = \text{false}$. The cases $h^B(\psi(x), \psi(y)) = \text{true}$ and $h^B(\psi(x), \psi(y)) = \text{undefined}$ are similar.

All queries in the nested relational algebra $\mathcal{NRA}(\Sigma)$ ([10]), even when extended with fixpoints or bounded fixpoints ($\mathcal{NRA}(\Sigma+fix$) and $\mathcal{NRA}(\Sigma+bfix$, see [26]), are $\Sigma$-queries. Here is a brief description of theses languages. First they contain the primitives: $p : d_p \to c_p$ for $p \in \Sigma$, $\pi_i : \tau_1 \times \tau_2 \to \tau_i$ the projections ($i = 1, 2$), $\eta : \sigma \to \{\sigma\}$ the singleton ($\sigma(x) := \{x\}$), $\mu : \{\sigma\} \to \{\sigma\}$ flatten ($\mu(x) := \bigcup_{y \in x} y$), $\cup : \{\tau\} \times \{\tau\} \to \{\tau\}$ union, $eq : b \times b \to \{\text{unit}\}$...
\[ (eq(x, x) := \{()\}, \text{and } eq(x, y) := \phi \text{ for } x \neq y)^8, \overline{\times} : \{\sigma\} \times \{\tau\} \rightarrow \{\sigma \times \tau\} \text{ cartesian product, and } not : \{unit\} \rightarrow \{unit\} \text{ (not}(\phi) := \{()\}, \text{not}(\{()\}) := \phi). \]

Next, \( \mathcal{NRA}(\Sigma) \) is closed under composition of functions \((g \circ f)\), pairing \(((f, g))\), and map \((\text{when } f : \sigma \rightarrow \tau, \text{then } map(f) : \{\sigma\} \rightarrow \{\tau\} \text{ is defined to be } map(f)(\{x_1, \ldots, x_n\}) := \{f(x_1), \ldots, f(x_n)\})\). The fixpoint construction is defined for some function \( f : \sigma \times \{\tau\} \rightarrow \{\tau\} \) to be \( \text{fix}(f)(x) := \bigcup_{n \geq 0} y_n \), where \( y_0 := \phi, y_{n+1} := y_n \cup f(x, y_n) \).

When no function symbols are present (\( \Sigma = \phi \)), then this definition just says that \( f \) is invariant under isomorphisms:

**Proposition 1** Let \( \Sigma = \phi \). Then \( f \) is a generic and domain independent query iff \( f \) is a \( \Sigma \)-query.

**Proof** Only one direction is nontrivial, so suppose that \( f \) is a generic and domain independent query. We show that for any partial isomorphism \( \psi, f^B \circ \psi \subseteq \psi \circ p^A \). Let \( x \in [\sigma]^A \), and \( A_0 \subseteq A \) be the collection of all atoms mentioned in \( x \). By domain independence, \( \psi(x) \) is defined iff \( \psi \) is defined on \( A_0 \). Because \( f \) is invariant under isomorphisms, \( f^A(x) \) mentions only atoms in \( A_0 \), so \( \psi(f^A(x)) \) is defined, when both \( \psi(x) \) and \( f^B(\psi(x)) \) are. \( \square \)

### 3 Computable \( \Sigma \)-Queries

We define, in this section a conservative notion of computable \( \Sigma \)-queries. The idea is that \( f : \sigma \rightarrow \tau \) is computable, if \( f^A(x) \) can be computed (by some Turing machine), given an encoding of \( x \), and recursive indexes \( e_1, \ldots, e_k \) for the functions \( p^A_1, \ldots, p^A_k \).

Consider some countable \( \Sigma \)-structure \((A, p^A_1, \ldots, p^A_k)\) and a family of bijections \( \psi_b : A \rightarrow \text{nat} \) for every base type \( b \). \( \psi \) induces a \( \Sigma \)-structure on \( \text{nat} \), by defining \( p^\text{nat}_b(x) := \psi(p^A_b(\psi^{-1}(x))) \). Then \( \psi \) is an isomorphism of \( \Sigma \)-structures.

**Definition 5** A \( \Sigma \)-structure is decidable, if there is some isomorphism \( \psi : A \rightarrow \text{nat} \), such that \( p^\text{nat}_1, \ldots, p^\text{nat}_k \) are all partial recursive functions. Call such a \( \psi \) an encoding of \( A \).

In the sequel, we assume that the class of structures \( \mathcal{C} \) contains only computable \( \Sigma \)-structures. Recall that \( \varphi_0, \varphi_1, \ldots \) is an enumeration of all partial recursive functions \( \text{nat} \rightarrow \text{nat} \) ([25]).

**Definition 6** A computable \( \Sigma \)-query is a query \( f : \sigma \rightarrow \tau \), with the following property:

- For any family of sets \( A = \{A_b\}_{b \in \text{base types}} \) and any isomorphism \( \psi : A \rightarrow \text{nat} \), there is a partial recursive function \( f^\text{nat} : \text{nat}^{k+1} \rightarrow \text{nat} \), such \(^3\)

\(^3\)Note that the injectivity of \( \psi \) is necessary to make \( eq \) a \( \Sigma \)-query.
that: for any decidable \( \Sigma \)-structure \((A, p_1^A, \ldots, p_k^A)\) on \(A\) for which \( \psi \) is an encoding, if \((A, p_1^A, \ldots, p_k^A) \in \mathcal{C}\) and \( p_1^{nat} = \varphi_1, \ldots, p_k^{nat} = \varphi_k \), then \( \forall x \in [\sigma]_A, \) either \( f^A(x) = \psi^{-1}(\text{encode}_{\sigma}(\text{encode}_{\sigma}(\psi(x)), e_1, \ldots, e_k)) \) or both are undefined. Here, \( \text{encode}_\sigma : [\sigma]_{nat} \to \text{nat} \) is some standard encoding of complex objects of type \( \sigma \) over natural numbers\(^4\).

Essentially, a \( \Sigma \)-query \( f \) is computable when there is a Turing machine which, when given a value \( x \) and the \( k \) indexes for the recursive functions \( p_1^A, \ldots, p_k^A \), computes \( f^A(x) \).

Recall the following known fact from recursion theory:

**Proposition 2** Let \( f : \mathbb{N} \to \mathbb{N} \) be a partial recursive function, such that \( \varphi_e \subseteq \varphi_{f(e)} \) implies \( \varphi_{f(e)} \subseteq \varphi_{f(e')} \) (i.e. \( f \) maps recursive indexes to recursive indexes, and is monotone). Then:

\[
\varphi_{f(e)} = \bigcup_{\varphi_{f(e')} \subseteq \varphi_{f(e)}} \varphi_{f(e')}
\]

(i.e. when \( f \) is monotone, it is also continuous).

**Proof** The inclusion \( \subseteq \) is the only nontrivial, so consider some \( x \) for which \( \varphi_{f(e)}(x) \) is defined, and suppose that for all \( e_0 \) for which \( \varphi_{e_0} \subseteq \varphi_e \) and \( \varphi_{e_0} \) is finite, \( \varphi_{f(e)}(x) \) is undefined. Then, we give a semidecision procedure for \( K \) (where \( K = \{ z / \varphi_e(z) \uparrow \} \)), which is a contradiction. Indeed, let \( k(z) \) be defined by: \( \varphi_{k(z)}(y) = (\text{if } \varphi_{k(z)}(y) \uparrow \text{ then } \varphi_{k(z)}(y) \text{ else } \uparrow) \). When \( z \in K \), then \( \varphi_{k(z)}(x) = \varphi_e \), and when \( z \in K \), then \( \varphi_{k(z)}(x) \) is a finite restriction of \( \varphi_e \). So \( \varphi_{f(k(z))}(x) \downarrow \) iff \( z \in K \). This would imply that \( K \) is r.e., which is a contradiction. \( \square \)

### 3.1 Relational Machines over \( \Sigma \)

We shall give in this section a more liberal definition for computable \( \Sigma \)-queries, called rm-queries, using a slight generalization of the (loose) generic machines (defined in \([8]\)). We call it a relational machine, after \([4]\). Most query languages, when extended with user-defined functions, express only rm-queries\(^5\).

Conversely, if, in the absence of user-defined functions, some query language \( L \) is complete w.r.t. computable queries, then its extension \( L' \) for user-defined functions is also complete w.r.t. rm-queries (we shall support this statement in the next section, by an example).

\(^4\)Not to be confused with the encoding \( \psi \)

\(^5\)\( \varphi_x^y \) \( \uparrow \) means that \( \varphi_x(z) \) diverges after \( y \) steps, and is a decidable property.

\(^6\)An exception is the algebra, in \([1]\), where inverses of user-defined functions can be also computed.
**Definition 7** (see [20, 8, 4]) A relational machine over \( \Sigma \), \( RM \), is a Turing machine extended with a finite number of registers \( R_1, \ldots, R_r \). The registers have types associated with them, say \( \sigma_1, \ldots, \sigma_r \). The machine can perform any of the following actions:

- A “traditional” TM instruction, of the form \((q_i, a_j, M, q_{i'}), \) where \( q_i, q_{i'} \) are the old and new state, \( a_j, a_{j'} \) are the old and new tape symbol, and \( M \in \{ \text{left, right} \} \) is the movement performed by the head.

- Assignment instructions, of the form: \((q_i, R_l := h(R_{l1}, \ldots, R_{lp}), q_{i'}), \) where \( h \) is any expression in \( NRA(\Sigma) \), of the right type (in particular, \( h \) can be any function symbol in \( \Sigma \)). The meaning is: when in state \( q_i \), assign to \( R_l \) the value of \( h(R_{l1}, \ldots, R_{lp}) \), and switch to state \( q_{i'} \).

- Conditional instructions, of the form: \((q_i, R_k, q_{i'}, q_{i''}), \) The type \( \sigma_k \) of \( R_k \) must be some set type, and the meaning is: when in state \( q_i \), switch to \( q_{i'} \) if \( R_k \) contains currently \( \phi \), or switch to \( q_{i''} \) otherwise.

We always assume \( RM \) to be deterministic.

Given some \( \Sigma \)-structure \((A, p_{\Sigma}^1, \ldots, p_{\Sigma}^k)\) and some relational machine \( RM \) over \( \Sigma \), we say that \( RM \) computes \( f^A : [\sigma]_A \rightarrow [\tau]_A \) for the \( \Sigma \)-structure \( A \), if, whenever started in its initial state, with some \( x \in [\sigma]_A \) in \( R_1 \), with \( R_2, \ldots, R_r \) undefined, and with an empty tape, \( M \) will reach a final state iff \( f^A(x) \) is defined, and in this case, \( M \) halts with \( f^A(x) \) in \( R_2 \). During the computation, the functions \( h \) in the assignment instructions are computed in the structure \( A \). If, at some stage of the computation, an assignment instruction \( R_l := h(R_{l1}, \ldots, R_{lp}) \) is reached and, either some of \( R_{l1}, \ldots, R_{lp} \) is undefined, or \( h^A(R_{l1}, \ldots, R_{lp}) \) is undefined, then the whole computation is undefined.

Note that \( h \) can be an “user-defined function” (from \( \Sigma \)), or it can be any complicated expression in \( NRA(\Sigma) \), possible involving \text{map}. It cannot, however, contain a fixpoint construction, or any other kind of iteration. But fixpoints, or other iterations (including the construction of the powerset) can be easily simulated.

A relational machine over \( \Sigma \) differs in two ways from the loose generic machine in [8], or the relational machine in [4]. First it uses complex objects instead of flat relations, so our relational machines are more powerful: it can be shown that for \( \Sigma = \phi \) they can express any computable query, and we shall prove in theorem 1 that this result holds for arbitrary \( \Sigma \). Secondly, a relational machine over \( \Sigma \) can interrogate the functions in \( \Sigma \) as oracles (see [25] for a definition of Turing machines with oracles).

**Definition 8** A query \( f : \sigma \rightarrow \tau \), for a given class of structures \( C \), is called rm-computable, if there is some relational machine \( RM \) which computes \( f^A \) on each structure \( A \in C \).

\(^7\)Recall that, unlike [8, 4], these types can be of complex objects.
We immediately have:

**Proposition 3** Any rm-computable function is a \( \Sigma \)-query.

**Proof** Obvious \( \square \)

**Proposition 4** The class of rm-computable functions contains all functions expressible in \( {\mathcal{NRA}(\Sigma)} \), and is closed under composition, pairing, fixpoints and bounded fixpoints (see \([26]\)).

**Proof** Any function in \( {\mathcal{NRA}(\Sigma)} \) can be computed in one step. It is easy to see that the \( \text{rm} \)-computable functions are closed under composition and pairing. \( fix(f) \) is also easily computed, by iteration. \( \square \)

**Proposition 5** The \( \text{rm} \)-computable functions are closed under map.

**Proof** Suppose \( f : \sigma \rightarrow \tau \) is computed by some relational machine \( RM \), with \( r \) registers, of types \( \sigma_1, \ldots, \sigma_r \). Recall that, \( \sigma = \sigma_1 \) and \( \tau = \sigma_2 \). We simulate \( \text{map}(f) : \{ \sigma \} \rightarrow \{ \tau \} \) by some relational machine \( RM' \), with two tapes\(^8\), with more than \( r \) registers. Call \( r \) of them \( R'_1, \ldots, R'_r \), with the types \( \{ \sigma \times \sigma_1 \}, \ldots, \{ \sigma \times \sigma_r \} \) respectively.

Suppose the input is \( z = \{ x_1, \ldots, x_n \} \). The first idea is to have any \( R'_i \) hold, at each time \( t \), the set \( \{ (x_1, y_1), \ldots, (x_n, y_n) \} \), where \( y_j \) is the value hold by the register \( R_i \), at the same moment \( t \), during the execution of \( RM \), on input \( x_j \). We use the inputs \( x_1, \ldots, x_n \) as tags: the pair \( (x_j, y_j) \) means that \( y_j \) was generated from \( x_j \). Suppose that, in some transition, \( RM \) performs the assignment \( R_5 := h(R_3, R_4) \), with \( h \in {\mathcal{NRA}(\Sigma)} \), and suppose \( RM' \) holds in \( R'_3 \) and \( R'_4 \) the values \( \{ (x_1, y_1), \ldots, (x_n, y_n) \} \) and \( \{ (x_1, z_1), \ldots, (x_n, z_n) \} \) respectively. Then \( RM' \) has to assign \( \{ (x_1, h(y_1, z_1)), \ldots, (x_n, h(y_n, z_n)) \} \) to \( R_3 \), which is easy to compute in \( {\mathcal{NRA}(\Sigma)} \), since we have an equality test on \( \sigma \).

The problem occurs when we try to handle the conditionals of \( RM \). Let \( (q_i, R_k, q'_i, q''_i) \) be some conditional instruction in \( RM \). \( RM' \) has to look at all values currently in \( R_k \), \( \{ (x_1, y_1), \ldots, (x_n, y_n) \} \): if all \( y_j \)'s are \( \phi \) (and \( RM' \) can determine that, in two steps, using an auxiliary register), then \( RM' \) has to switch to \( q''_i \). If all \( y_j \)'s are nonempty, then it has to switch to \( q''_i \). But when there are both empty and nonempty values among the \( y_j \)'s, then both branches must be simulated, and this forces us to design \( RM' \) in a more complicated way.

Thus, we revise our simulation of \( RM \) by \( RM' \), in the sense that we simulate all possible executions of \( RM \), using \( RM' \)'s second tape as a stack. For some \( j \), let \( t_j \) be the number of conditionals executed by \( RM \) during the computation of \( f(x_j) \). Define \( s_j \) to be a string from \( \{0,1\}^* \), of length \( t_j \), corresponding to the branches taken by \( RM \) at conditional: 0 means the register was empty, 1 means otherwise. \( RM' \) will generate, in some order, all sequences \( s \in \{0,1\}^* \)

\(^8\)A \( RM \) with two tapes can be easily simulated by some \( RM \) with only one tape.
on its second tape, and will simulate $RM$ on all inputs, making all decisions according to $s$.

Now we are ready to describe how $RM'$ simulates the action of $RM$ on each value of the input $x$. $RM'$ initializes a special register $R_{out}$ (different from $R_1', \ldots, R_n'$) with $\phi$, then starts generating all sequences $s$. For any such $s$, $RM'$ initializes $R_i' := \{(x_1, x_1), \ldots, (x_n, x_n)\}$ (some input register $R_n$ holds the unaltered input $\{x_1, \ldots, x_n\}$ all the time), $R_2' := \phi, \ldots, R_n' := \phi$, and then proceeds to simulate $RM$, but only on those computations having the a sequence of conditionals corresponding to $s$. At each step, each register $R_i$ holds only a subset of $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. When the instruction to be simulated is not a conditional, then the simulation takes place as described above. If the action is a conditional $(q_i, R_k, q_{i'}, q_{i''})$, then $RM'$ will inspect the next symbol of $s$. If it is 0, and the value of $R_{i'}$ is currently $\{(x_{j_1}, y_{j_1}), \ldots, (x_{j_m}, y_{j_m})\}$, then $RM'$ eliminates from this set all values $(x_{j_1}, y_{j_1})$ for which $y_{j_1} \neq \phi$. For each such value eliminated, it also eliminates all values of the form $(x_{j_1}, z_{j_1})$ from all other registers $R_1', \ldots, R_n'$. If $R_{i'}$ becomes empty, then $RM'$ aborts the computation for the current $s$, and goes to the next $s$. Else, $RM'$ advances one position in $s$, and continues the simulation. If the simulated $RM$ reaches the final state, then $RM'$ simply executes $R_{out} := R_{out} \cup R_2'$. Else, if $s$ is exhausted without reaching the final state, the current computation (for that particular $s$) is aborted. In all cases, $RM'$ goes to the next $s$.

$RM'$ ends the simulation, when $\forall x_j \in R_{in}, \exists z. (x_j, z) \in R_{out}$. The final result is easily extracted from $R_{out}$.

To see the correctness of the above simulation, notice that it will reach an end, if $f(x_1), \ldots, f(x_n)$ are all defined. Conversely, if some computation of $RM$, say for $f(x_j)$, loops, then so will $RM'$. If, during the computation of $f(x_j)$, $RM$ gets stuck because some function $h$ it wants to apply at some step is undefined, so will $RM'$. And if $RM$ tries to apply some $h$ on an undefined register $R_i$, while computing $f(x_j)$, then $RM'$ will never reach any value of the form $(x_j, z)$ in $R_{out}$, so it will loop forever. $\square$

**Corollary 1** Any query expressible in $\mathcal{NRA}(\Sigma) + \text{fix}$ is rm-computable.

This corollary can be easily extended to other query languages for complex objects:

**Fact 1** Let $L$ be any complex object programming language, extended with $\Sigma$. Then, all queries expressible in $L$ are rm-computable$^9$.

**Proposition 6** Let $f$ be a rm-computable $\Sigma$-query over a class $C$ containing only decidable structures. Then $f$ is a computable $\Sigma$-query over $C$.

$^9$We remind the reader, again, of the exception present in [1], where the languages considered can use the inverses of functions in $\Sigma$.  

12
Proof Let \( RM \) be the relational machine over \( \Sigma \) computing \( f \). We shall construct a Turing Machine \( T \) computing \( f \). \( T \) stores on its tape the encoded content of \( RM \)'s registers \( R_1, \ldots, R_r \). It starts by copying its input value \( x \) into \( R_1 \), and by “marking” \( R_2, \ldots, R_r \) as “undefined”. Then it simulates \( RM \): when it reaches some instruction of the form \((q_i, R_i := h(R_{i_1}, \ldots, R_{i_p}))\), it first checks whether some of \( R_{i_1}, \ldots, R_{i_p} \) is marked “undefined”. If so, then \( M \) enters an infinite loop. Else, it computes the value of \( h(R_{i_1}, \ldots, R_{i_p}) \), possibly using the indexes \( e_1, \ldots, e_k \) for the functions in \( \Sigma \). □

4 A Complete Query Language

When \( \Sigma = \phi \), a query language is called complete iff it can express all computable queries ([11]). Conceptually, the proof of the completeness of some query language \( L \) could be split into two parts:

1. The proof that \( L \) can express any \( rm \)-computable query.
2. The proof that any computable query is \( rm \)-computable.

When \( \Sigma \neq \phi \), we proceed along the same lines. We call a query language over \( \Sigma \) complete, iff it can express all computable \( \Sigma \)-queries. While we postpone the proof of 2 for section 5, we give an example of a complete query language, and prove 1.

The language we consider is \( NRA(\Sigma) + \text{fix} \). Clearly, each query in this language can be computed by some relational machine in \( \Sigma \) (corollary (1)). For the completeness, we add a new base type \( n \), and suppose \( \Sigma \) contains two function symbols: \( z : \text{unit} \rightarrow n \), and \( s : n \rightarrow n \). To emphasize that the new type \( n \), and the operations \( z, s \) were added to \( \Sigma \), we shall write \( \Sigma \cup n \).

We consider only structures \( A \) for which \( \{z^A, s^A(z), s^A(s^A(z)), \ldots\} \) is infinite: thus, we can identify this subset of \([n]_A\) with the naturals. We say that some numerical function \( f : \text{nat} \rightarrow \text{nat} \) is represented by \( f \) if, for any \((A, p_1^A, \ldots, p_k^A) \in \mathcal{C} \), the restriction of \( f^A \) to the naturals coincides with \( \eta \circ f \). For some \( x \in \text{nat} \), let \( x \) be a shorthand for \( s^A(z) \).

Lemma 1 All recursive functions \( f : \text{nat} \rightarrow \text{nat} \) are representable in the language \( NRA(\Sigma \cup n) + \text{fix} \).

Proof We show that the representable functions are closed under primitive recursion and minimization. Let

\[
f(0, y_1, \ldots, y_k) = h(y_1, \ldots, y_k) \\
f(x + 1, y_1, \ldots, y_k) = g(x, f(x, y_1, \ldots, y_k), y_1, \ldots, y_k)
\]
be a primitive recursion schema for $f$, and suppose $h, g$ are representable by $h, g$. To represent $f$, we first define a function $F : n \times n^k \rightarrow \{n \times n\}$, such that (informally) $F(x, y_1, \ldots, y_k) = \{(x', f(x', y_1, \ldots, y_k)) / 0 \leq x' \leq x\}$. Indeed, $F(x, y_1, \ldots, y_k) = \text{fix}(\lambda(x, y_1, \ldots, y_k, u).\{(z) \mapsto h(y_1, \ldots, y_k) \cup \text{ext}(\lambda(v, f).\text{if } v = x \text{ then } \phi \text{ else } \{s(v) \mapsto g(v, f, y_1, \ldots, y_k)(u))\})$. $f$ can be easily computed from $F$.

To represent a function $f$ defined by the minimization operator, $f(x) = \mu y.g(x, y)$, compute first the set $F(\bar{x}) = \{z, \overline{s(z)}, \overline{s^2(z)}, \ldots, \overline{s^q(z)}\}$, where $z = \mu y.g(x, y)$. $F(\bar{x})$ is easily computed with a fixpoint, which diverges, when $\mu y.g(x, y)$ is undefined.

**Proposition 7** $\mathcal{NRA}(\Sigma \cup n) + \text{fix}$ is complete in the following sense. Let $f : \sigma \rightarrow \tau$ be a rm-computable $\Sigma$-query, where $\tau$ is a set type ($\tau = \{\tau'\}$). Then $f$ can be expressed in $\mathcal{NRA}(\Sigma \cup n) + \text{fix}$, over the class $C$.

**Proof** The proof consists just in a simulation, in the language $\mathcal{NRA}(\Sigma \cup n) + \text{fix}$, of some relational machine $RM$. This is done with classical techniques, see [21,15]. No encoding/decoding of the input is necessary. During the simulation, the values of the $\tau$ registers are kept as a value of type $\sigma_1 \times \ldots \times \sigma_r$. The final result is easily extracted, because $\tau$ is a set type. $\square$

## 5 Computable $\Sigma$-Queries Can Be Computed by Relational Machines

In the absence of user-defined function symbols (i.e. $\Sigma = \phi$), relational machines can be viewed as a paradigmatic complete query language: the translations between some complete query language and relational machines are rather straightforward. A theorem proving that any computable query is rm-computable, constitutes the paradigmatic proof of the completeness of a query language.

In the presence of user-defined functions in $\Sigma$, the relational machines over $\Sigma$ can be viewed as a natural extension of a query language to accommodate user-defined functions. The theorem which we prove next, states that any computable $\Sigma$-query is still computable by some relational machine. So, any complete query language which is extended with user-defined functions in a "reasonable way", is still complete.

**Theorem 1** Given any Turing machine $T$ which computes the $\Sigma$-query $f : \sigma \rightarrow \tau$, there is some relational machine $RM$ such that, on any decidable and total structure $A$ (i.e. $p^1, \ldots, p^q$ are total functions), $T$ and $RM$ compute the same function.

**Proof** Let $x$ be some input of type $\sigma$ for $RM$ (i.e. $v \in [\sigma]_A$), in $R_1$. Suppose there is only one base type $b$, and that all values of type $b$ mentioned in $x$ are $\{x_1, \ldots, x_n\}$ (the "atoms" of $x$). We cannot directly simulate $T$ on $\psi(x)$ with
$RM$, because we know neither the encoding function $\psi : A \rightarrow \text{nat}$, nor the recursive indexes $e_1, \ldots, e_k$ for $p_1^A, \ldots, p_k^A$.

When no function symbols are present in $\Sigma$ (i.e. $k = 0$), then it suffices to pick any $n$ integers to encode $x_1, \ldots, x_n$, say $0, 1, \ldots, n-1$, because the mapping $\psi_0(x_i) := i - 1$, with $\psi_0$ undefined elsewhere, is a partial isomorphism. This fact is essential in proofs like those in [21, 15], where Turing machines are simulated within database query languages. But when $\Sigma \neq \emptyset$, the above $\psi_0$ is no longer a partial isomorphism.

The idea is to search systematically for $k$ functions with finite domains $\varphi_{c_1}, \ldots, \varphi_{c_k}$, and a finite partial isomorphism $\psi_0 : (A, p_1^A, \ldots, p_k^A) \rightarrow (\text{nat}, \varphi_{c_1}, \ldots, \varphi_{c_k})$. In doing so, we use the following representations:

- Each $\varphi_{c_i}$ is represented by its finite graph: $\{(w_1, \varphi_{c_i}(w_1)), \ldots, (w_p, \varphi_{c_i}(w_p))\}$. We can easily compute the index $c_i$ from the graph representation, but in addition, we can test whether $w \in \text{dom}(\varphi_{c_i})$, for some $w$.

- $\psi_0$ is represented by:
  
  1. Its domain $\{x_1, \ldots, x_m\}$, which always includes $\{x_1, \ldots, x_n\}$ (so $m \geq n$). We keep it in $R_3$ of $RM$, which has type $\{b\}$.
  2. Its codomain $\{u_1, \ldots, u_m\}$, a set of integers, which is kept on the tape of $RM$.
  3. A set $O$ of total order relations on $\text{dom}(\psi_0)$, which we keep in $R_4$, of type $\{b \times b\}$. In fact, each total order $\text{ord} \in R_4$ defines a different partial isomorphism $\psi_0$, and $R_4$ always contains all partial order defining a partial isomorphisms, with the given domain and codomain, for the given $\varphi_{c_1}, \ldots, \varphi_{c_k}$.

Call such a collection of $\text{dom}(\psi_0), \text{codom}(\psi_0), \varphi_{c_1}, \ldots, \varphi_{c_k}, O$, a representation, $R$. In fact, a representation comprises several partial isomorphisms $\psi_0$, with the same domain and codomain.

Now we can sketch the overall strategy of $RM$: it systematically enumerates all the pairs $(R, s)$, where $R$ is a representation, and $s$ a natural number. For each of them, it simulates the Turing machine $T$ on $\psi_0(x), c_1, \ldots, c_k$, for $s$ steps. If $T$ halts after at most $s$ steps, with output $u$, then $RM$ writes $\psi_0^{-1}(u)$ in $R_2$, and halts. Else, it continues with the next pair $(R', s')$.

To fill in the details, first we show how $GM$ can simulate $T$ on $\psi_0(x), c_1, \ldots, c_k$. Certainly, given ONE total order $\text{ord} \in O$ in some register $R$, we know that a relational machine can compute $\psi_0(x)$, and, in the end, write $y := \psi_0^{-1}(u)$ in some register $R'$ ($\psi_0$ is uniquely determined by $\text{dom}(\psi_0), \text{codom}(\psi_0)$, the order $\text{ord}$ on the domain, and the order of natural numbers on the codomain). So, it suffices to program $RM$ to map the function described above, over all order relations in $O$ using the technique of proposition 5. Clearly, all halting computations will produce the same result, say $y$, so the result of the map will be a set.

15
with one element, \( \{y\} \). Because \( y \) itself is also a set, we can get it by applying \( \mu \) (the “flatten” operation). By a similar argument, we can test whether there is some order relation in \( \mathcal{O} \) producing a halting computation.

Next, we have to show how to systematically generate the representations \( R \). At each step, we generate a new representation, in one of two ways:

1. We define \( \text{dom}(\psi_0) := \{x_1, \ldots, x_n\} \), choose \( n \) integers \( u_1, \ldots, u_n \), and define \( \text{codom}(\psi_0) := \{u_1, \ldots, u_n\} \). We let \( \mathcal{O} \) be the set of all permutations of \( \{x_1, \ldots, x_n\} \), and define \( \varphi_c_1, \ldots, \varphi_c_k \) to be the empty functions. Clearly, any bijection \( \psi_0 : \{x_1, \ldots, x_n\} \rightarrow \{u_1, \ldots, u_n\} \) is a partial isomorphism, from \( (A, p^A_1, \ldots, p^A_k) \) to \( (\text{nat}, \varphi_c_1, \ldots, \varphi_c_k) \).

2. Suppose we have already some representation, so \( R_3 \) contains \( \text{dom}(\psi_0) = \{x_1, \ldots, x_m\} \), \( \text{codom}(\psi_0) = \{u_1, \ldots, u_m\} \) and \( \varphi_c_1, \ldots, \varphi_c_k \) are on the tape, and \( R_4 \) contains \( \mathcal{O} \), the set of all total orders on \( \text{dom}(\psi_0) \) which define a partial isomorphism. Pick some \( p_i \in \Sigma \), \( p_i : \sigma_i \rightarrow \tau_i \), and let \( x \in \{\{\sigma_i\}\}_A \) and \( u \in \{\{\sigma_i\}\}_\text{nat} \) be all values of type \( \sigma_i \) which can be constructed from the atoms in \( \text{dom}(\psi_0) \) and \( \text{codom}(\psi_0) \). By applying \( p^A_i \) on each value in \( x \), we may produce some new atoms of type \( b \), which were not present in \( \text{dom}(\psi_0) \), say \( x_{m+1}, x_{m+2}, \ldots, x_{m'} \). Suppose \( m' > m \) (else, make other selections). Then, extend \( \text{dom}(\psi_0) \) with these new values, by executing \( R_3 := R_3 \cup \{x_{m+1}, x_{m+2}, \ldots, x_{m'}\} \), and extend \( \text{codom}(\psi_0) \) with \( m' - m \) new generated integers \( u_{m+1}, \ldots, u_{m'} \), to the set \( \{u_1, \ldots, u_{m'}\} \). Choose some extension of \( \varphi_c_i \), which is defined on all values in \( x \); when extending \( \varphi_c_i \), we may use for its codomain, all atoms from the larger set \( \{u_1, \ldots, u_{m'}\} \).

Finally, extend in all possible ways the total orders in \( R_4 \) to total orders on the new \( \text{dom}(\psi_0) \), and then eliminate all but those which define a partial isomorphism (it suffices to check commutativity with the new \( \varphi_c_i \), i.e. \( \varphi_c_i \circ \psi_0 \subseteq \psi_0 \circ p^A_i \)). If \( \mathcal{O} \) becomes empty, other selections have to be done.

It remains to prove that, if \( f^A(x) \) is defined, then there is some finite representation \( \text{dom}(\psi_0), \text{codom}(\psi_0), \varphi_c_1, \ldots, \varphi_c_k, \mathcal{O} \), for which \( T \) converges on \( \psi_0(x), \varphi_c_1, \ldots, \varphi_c_k \). Recall that \( \psi : (A, p^A_1, \ldots, p^A_k) \rightarrow (\text{nat}, \varphi_c_1, \ldots, \varphi_c_k) \) is a (total) isomorphism, and let \( \mathcal{R} \) be the set of all representations for which \( \varphi_c_1 \subseteq \varphi_c_1, \ldots, \varphi_c_k \subseteq \varphi_c_k \), and which contain at least some partial isomorphism \( \psi_0 \subseteq \psi \). Define \( \varphi_{C_1}, \ldots, \varphi_{C_k}, \Psi \) to be their union.\(^{12}\) We still have \( \varphi_{C_1} \subseteq \varphi_c_1, \ldots, \varphi_{C_k} \subseteq \varphi_c_k \) and \( \Psi \subseteq \psi \), but we don’t have equality, because \( \text{dom}(\Psi) \) contains only those elements which can be reached by repeatedly applying the functions in \( \Sigma \), and doesn’t contain the predecessors. Still, \( \Psi : (A, p^A_1, \ldots, p^A_k) \rightarrow (\text{nat}, \varphi_{C_1}, \ldots, \varphi_{C_k}) \) is a partial isomorphism.

\(^1\)Here, we rely on \( p^A_i \) being total.

\(^2\)When extending \( \varphi_c_i \), it is essential to be able to test if \( \varphi_c_i \) is defined on each value in \( x \).

\(^3\)\( \varphi_{C_1}, \ldots, \varphi_{C_k} \) are indeed recursive, because we can enumerate all finite \( \varphi_c_i \) for which \( \varphi_c_i \subseteq \varphi_{C_i} \) (to see this, recall that \( \varphi_c_i \) is total, and that we encode \( \varphi_c_i \) by its finite graph, so \( \varphi_c_i \subseteq \psi_{C_i} \) is decidable). This justifies the notation, but this is not used in the following.
observation is the fact that its inverse, \((\Psi)^{-1}\) is also a partial isomorphism!
This is due to the fact that the domain of \(\Psi\) is closed under the operations \(p_1, \ldots, p_k\). So \(\varphi^A \circ (\Psi)^{-1} : (nat, \varphi_{C_1}, \ldots, \varphi_{C_k}) \rightarrow (nat, \varphi_{C_1}, \ldots, \varphi_{C_k})\) is a partial isomorphism. Because \(f^{nat}(\text{encode}(\Psi(x)), c_1, \ldots, c_k)\) converges, so does \(f^{nat}(\text{encode}(\Psi(x)), C_1, \ldots, C_k)\). From the proposition 2, we know that there are some \(c_1, \ldots, c_k, \psi_0\) for which \(\varphi_{C_1}, \ldots, \varphi_{C_k}, \psi_0\) are finite, such that \(f^{nat}(\text{encode}(\Psi(x)), c_1, \ldots, c_k)\) converges. Finally, take \(\psi_0\) to be the restriction of \(\Psi\) to the atoms mentioned in the domain and codomain of \(\varphi_{C_1}, \ldots, \varphi_{C_k}\).

**Example** Let \(\Sigma = \{p\}\), with \(p : b \rightarrow b\), and let \(f : \{b\} \rightarrow \{b\}\) be the following query: \(f(\phi) := \phi, f(\{x_1, \ldots, x_n\}) := \{f(x_1)\}\), where \(k \geq 0\) is the smallest number for which \(f^k(x_1) = \ldots = f^k(x_n)\). \(f(\{x_1, \ldots, x_n\})\) is undefined if there is no such \(k\). A Turing machine \(T\) for computing \(f\) would receive \(\{x_1, \ldots, x_n\}\) and \(e\) (an index for \(p\)) as inputs on its tape. After checking that the input is nonempty, \(T\) would repeatedly apply \(p\) to all the elements of the input set, and would eliminate the duplicates, until a set with only one element is reached. We shall explain how \(RM\) can simulate \(T\), by assuming the input of \(RM\) to be \(\{x, y, z\}\), with \(x, y, z \in \mathcal{A}\), for some \(\Sigma\)-structure \(\mathcal{A}\). Suppose \(p(x) = p(y) = u, p(z) = v\) and \(p(u) = p(v) = w\), such that \(f(\{x, y, z\}) = \{w\}\). We give below three steps which could lead to a representation \(\text{dom}(\psi_0), \text{codom}(\psi_0), \varphi_{c_0}\) for which \(T\) halts.

**Step 0** \(\text{dom}(\psi_0) := \{x, y, z\}, \text{codom}(\psi_0) := \{0, 1, 2\}\) (we could have chosen any three integers), \(\varphi_{c_0}\) is empty, and \(O^2\) contains all permutations over \(\{x, y, z\}\). \(T\) will not halt on the input \(\{0, 1, 2\}, c_0\).

**Step 1** Compute \(p(x), p(y), p(z)\), producing two new values \(u, v\), and define \(\text{dom}(\psi_0) := \{x, y, z, u, v\}\). Chose any two new numbers, say \(3, 4\), and define \(\varphi_{c_1}\) arbitrarily on \(0, 1, 2\), using any values in \(\{0, 1, 2, 3, 4\}\): suppose we have chosen \(\varphi_{c_1}(0) = \varphi_{c_1}(2) = 4, \varphi_{c_1}(1) = 3\). Next, we extend all orders in \(O^2\) to orders on \(x, y, z, u, v\), and select only those which define a partial isomorphism: \(O^1 := \{(x, y, z, u, v), (y, z, x, v, u)\}\). Note that lots of “bad” choices for \(\varphi_{c_1}\) would lead to an empty \(O^1\). \(T\) still doesn’t halt on the input \(\{0, 1, 2, 3, 4\}, c_1\).

**Step 2** Compute \(p(x), p(y), p(z), p(u), p(v)\), producing only one new value: \(w\). So choose one new number, say \(5\), and choose some extension of \(\varphi_{c_2}\), say by defining \(\varphi_{c_2}(3) = \varphi_{c_2}(4) = 5\) (all other choices will lead to \(O^2 = \emptyset\)). Then \(O^2 := \{(x, z, y, u, v), (y, z, x, v, u, w)\}\). This time, \(T\) will halt on the input \(\{0, 1, 2, 3, 4, 5\}, c_2\), with the output \(\{5\}\). Under both partial isomorphisms contained in this representation, the preimage of \(\{5\}\) is \(\{w\}\) (the preimage is always unique, as proven in the theorem), so \(RM\) writes \(\{w\}\) in its output register and halts.

This example suggests that we could restrict \(RM\) to search only for representations for which \(\text{codom}(\psi_0)\) is of the form \(\{0, 1, 2, \ldots\}\), but this would make the proof of the termination of \(RM\) a bit more complicated.
The practical consequence of this theorem is the fact that a complete database query language without user-defined function symbols remains complete after extending it in a "reasonably" way with all function symbols in $\Sigma$.

**Corollary 2** A query language $L$ without function symbols is complete iff it can express all RM-computable queries.

**Proof** This is just theorem 1 for the case $\Sigma = \emptyset$. □

**Fact 2** Let $L$ be a complete database query language, without user-defined function symbols. Then, if $L'$ is a "reasonable" extension of $L$ with the function symbols in $C$, then $L'$ is also complete.

**Proof** If $L$ is complete, then for any relational machine $RM$, there is some expression $Q_{RM}$ in $L$, computing the same query as $RM$. If $L'$ is a "reasonable" extension of $L$, then, for any relational machine $RM'$ over $\Sigma$, there will be some expression $Q_{RM'}$ in $L'$ denoting the same query as $RM'$. Finally we use theorem 1 to argue that any computable $\Sigma$-query can be simulated by some relational machine $RM'$ over $\Sigma$. □

**Proposition 8** $NRA(\Sigma \cup n) + fix$ is a complete query language.

**Proof** This is a direct consequence of proposition 7 and theorem 1. □

## 6 The Successor Function and Object Identities

To simulate a computable $\Sigma$-query $f: \sigma \rightarrow \tau$ in $NRA(\Sigma) + fix$, we needed a kernel of arithmetic in our language, namely a type $n$, and two functions $z: unit \rightarrow n$ and $s: n \rightarrow n$, the latter being injective. How severe is this restriction? We shall argue, in this section, that this condition is equivalent to having object inventions in the language.

To define $NRA$ with object inventions, extend it with a new base type $o$, and a new function symbol $\text{invent} : unit \rightarrow o$. The semantics becomes more complicated. An $O-$*structure* $A$ is a tuple $(A, O, p_1^A, \ldots, p_k^A)$, where $(A, p_1^A, \ldots, p_k^A)$ is a $\Sigma$-structure, and $O$ is a set. A type $\sigma$ is interpreted in the old way, by $[b]_{(A,O)} := A_b$, $[o]_{(A,O)} := O$, $[\sigma \times \tau]_{(A,O)} := [\sigma]_{(A,O)} \times [\tau]_{(A,O)}$ and $[[\sigma]]_{(A,O)} := P_{\text{fin}}([\sigma]_{(A,O)})$. For each type $\sigma$, we have a function $\text{oids} : [\sigma]_{(A,O)} \rightarrow P_{\text{fin}}(O)$, $\text{oids}(x)$ = the set of all oids mentioned in the object $x$. A function $f: \sigma \rightarrow \tau$ is interpreted as a binary relation $[f] \subseteq [\sigma] \times [\tau]$, defined by induction on the structure of $f$. The interesting cases are:

- $[\text{invent}] := P((\)) \times O$ (invent can invent any oid).
- $[g \circ f] := [g] \circ [f]$ (relation composition).
If $f_i : \sigma \to \tau_i$ ($i = 1, 2$), then $\left[ (f_1, f_2) \right] := \{ (x, (y_1, y_2)) / (x, y_i) \in [f_i], i = 1, 2 \text{ and } \text{oids}(y_1) - \text{oids}(x)) \cap (\text{oids}(y_2) - \text{oids}(x)) = \emptyset \}$. The idea is that the set of oids invented by $f_1$ and $f_2$ should be disjoint.

The most complicated part is to define $[\text{map}(f)]$. Think of $f$ as a function; then $\text{map}(f)(\{a, b, c\}) = \{f(a), f(b), f(c)\}$. Clearly, we want the oids “invented” by $f(a), f(b), f(c)$ to be disjoint. But the same set $x = \{a, b, c\}$ can be written as $x = \{a, a, b, c\}$. Do we want $\text{map}(f)(\{a, a, b, c\})$ to be $\{f(a), f(a), f(b), f(c)\}$? The problem is that $f(a)$ and $f(a)$ may be distinct (because they invent distinct oids). Example: $\text{map}(\text{invent})(\{a, b, c\}) = \{p_1, p_2, p_3\}$; do we allow $\text{map}(\text{invent})(\{a, a, b, c\}) = \{p_1, p'_1, p_2, p_3\}$? Then $\text{map}(\text{invent})(x)$ could be a set, of any cardinality!

We shall choose the option in which $\text{map}(f)(x)$ always returns a set of cardinality $\leq \text{card}(x)$. This is consistent with the deterministic semantics of detTL ([5]) and IQL ([3]) 13.

$[\text{map}(f)] := \{ (x, y) / \exists \varphi : x \to y, \text{ surjective} \text{ s.t. } \forall u \in x, (u, \varphi(u)) \in [f], \text{ and } \forall u, u' \in x, u \neq u' \Rightarrow (\text{oids}(\varphi(u)) - \text{oids}(u)) \cap (\text{oids}(\varphi(u')) - \text{oids}(u')) = \emptyset \}.$

The interpretation of the fixpoint in the presence of object inventions is only slightly more difficult, because we are working with relations instead of functions:

- The fixpoint. Let $f : \sigma \times \{\tau\} \to \{\tau\}$; then $\text{fix}(f) : \sigma \to \{\tau\}$, and $\left[\text{fix}(f)\right] := \{ (x, y) / \exists n \geq 0, \exists y_0, \ldots, y_n, \text{ such that } y_0 := \phi, y_i := y_{i-1} \cup z_i \text{ where } ((x, y_i), z_i) \in [f], \text{ and } y_n = y_{n-1} \}.$

Inflationary and the two bounded fixpoints, are defined indirectly, using the partial fixpoint.

This completes the definition of the semantics of our new language, call it $\mathcal{NRA}() + \text{fix} + \text{invent}.$

Claim IQL and $\mathcal{NRA}(\Sigma) + \text{fix} + \text{invent}$ have the same expressive power.

We define a $\Sigma$-query, in the presence of object invention, following [3]. For this, we extend the definition of a partial isomorphism to O-$\Sigma$-structures.

**Definition 9** A partial isomorphism $\psi : (A, O) \to (B, O')$ is a partial isomorphism from $A$ to $B$, and a partial, surjective function $\psi_o : O \to O'$.

We also need:

---

13The other option is quite unnatural, and does not correspond to the nondeterministic semantics in [5].
Definition 10 (see [3]) An O-isomorphism \( \psi : (A, O) \to (A, O) \) is an isomorphism which is the identity on \( A \).

The definition of a \( \Sigma \)-query, in the presence of oid's, is a straightforward generalization of [3]:

Definition 11 A \( \Sigma \)-query \( f : \sigma \to \tau \) is a family of binary relations \( f^{(A, O)} \subseteq [\sigma]_{(A, O)} \times [\tau]_{(A, O)} \) for each \( O - \Sigma \)-structure \((A, O)\), such that:

1. For any partial isomorphism \( \psi : (A, O) \to (B, O') \), we have \( f^{(B, O')} \circ \psi \subseteq \psi \circ f^{(A, O)} \) (here \( \circ \) stands for relation composition, and \( \subseteq \) for relation inclusion).

2. Whenever \((x, y) \in f^{(A, O)} \) and \((x, y') \in f^{(A, O)} \), there exists some \( O \)-isomorphism \( \psi : (A, O) \to (A, O) \) such that \( x = \psi_x(x) \) and \( y' = \psi_y(y) \).

We say that \( f \) is a computable \( \Sigma \)-query iff, in addition, the graph of \( f^{(A, O)} \) is r.e., for any decidable \( O - \Sigma \)-structure \((A, O)\).

We shall establish now the connection between queries on \( O - \Sigma \) and \( \Sigma \cup n \) structures. The bottom line is that they are somehow the same, but the exact statement is complicated by the fact that \( O - \Sigma \)-queries are relations, while \((\Sigma \cup n)\)-queries are functions.

6.1 Converting \( \mathcal{NRA}(\Sigma \cup n) \) into \( \mathcal{NRA}(\Sigma) + \text{invent} \)

The translation is:

- On types: \( \overline{b} := b, \overline{n} := \{o\}, \overline{\sigma \times \tau} := \overline{\sigma} \times \overline{\tau}, \overline{\{a\}} := \{\overline{a}\} \).

- On functions: \( \overline{\varepsilon} := \phi, \overline{s(x)} := \{\text{invent()}\} \cup x \). Next, the translation is carried on, inductively.

To explain the properties of this translation, we define first a relation between \((\Sigma \cup n)\)-structures and \( O - \Sigma \)-structures:

Definition 12 \( \mathcal{U} \) is the following relation between \((\Sigma \cup n)\)-structures and \( O - \Sigma \)-structures:

\[ ((A, N, s^A, z^A), (A, O)) \in \mathcal{U} \text{ iff } N = P_{\text{fin}}(O) \text{ and } \forall x \in P_{\text{fin}}(O), \exists a \in O \text{ such that } a \notin x \text{ and } s^A(x) = \{a\} \cup x \]

Definition 13 Let \( f : \sigma \to \tau \) be a \((\Sigma \cup n)\)-query, in which \( \sigma \) doesn't mention \( n \). Define \( \tilde{f} : \sigma \to \overline{\tau} \) to be the following \( O - \Sigma \)-query\(^{14}\):

\[ \tilde{f}^{(A, O)} := \bigcup \{f^{(A, N, s, z)} / ((A, N, s, z), (A, O)) \in \mathcal{U} \} \]

\(^{14}\)Note that \( \overline{\sigma} = \sigma \).
Proposition 9 \( f^{(A,O)} \), as defined above, is indeed a \( O - \Sigma \)-query.

Proof

1. Let \( \psi : (A, O) \to (B, O') \) be a \( O - \Sigma \) partial isomorphism; in particular, it is a \( \Sigma \)-partial isomorphism \( \psi : A \to B \). We have to prove that \( f^{(B,O')} \circ \psi \subseteq \psi \circ f^{(A,O)} \). For this, it suffices to prove that, for any \( (\Sigma \cup n) \)-structure \( (B, N', s^B, z^B) \), in relation \( U \) with \( (B, O') \), there is a structure \( (A, N, s^A, z^A) \), in relation \( U \) with \( (A, O) \), such that \( \psi : (A, N, s^A, z^A) \to (B, N', s^B, z^B) \) is a partial isomorphism. Take \( z^A := \phi \) and \( s^A(x) := \psi^{-1} \circ s^B \circ \psi(x) \), when the latter is defined. Check the fact that \( \psi \) is a partial isomorphism: \( \psi(s^A(x)) = \psi(\psi^{-1}(s^B(\psi(x)))) \supseteq s^B(\psi(x)) \) (the latter is based on the fact that \( \psi : O \to O' \) is surjective). The surjectivity of \( \psi : O \to O' \) is used to prove that the above definition is indeed a partial isomorphism.

2. Let \( (x, y), (x, y') \in f^{(A,O)} \). Then, there are two structures \( (A, N, s, z) \) and \( (A, N, s', z') \), such that \( y = f^{(A,N,s,z)} \) and \( y' = f^{(A,N,s',z')} \) (recall that \( N = P_{fin}(O) \) and that \( z = z' = \phi \)). Consider an extension \( O^* \supseteq O \) of \( O \), and \( s^* : P_{fin}(O^*) \to P_{fin}(O^*) \) an extension of \( s \), with the following properties:
   - \( \forall x \in P_{fin}(O^*), \exists a \in O^* \), such that \( a \notin x \) and \( s^*(x) = \{a\} \cup x \).
   - \( s^* \) is total.

Such an extension can be obtained, for example, by taking \( O^* \) to be the closure of \( O \) under the finite sets construction, \( O^* = \bigcup_{n \geq 0} O^n \), where \( O^0 = \phi, O^{n+1} = s^n \cup P_{fin}(O^n) \), and next by defining \( s^*(x) := s(x) \) when \( x \subseteq O \) and \( s(x) \) is defined, and \( s^*(x) := \{x\} \cup x \) otherwise.

Let \( N^* \) be \( P_{fin}(O^*) \). Next, we define \( \psi \) to be the smallest partial isomorphism \( \psi : (A, N^*, s^*, z) \to (A, N, s', z) \), which is the identity on \( A \), and which is closed under the following rule: if \( x \subseteq O \) and \( \psi(x) \), \( s'(\psi(x)) \) are defined, then \( \psi(a) = b \) where \( a, b \) are such that \( s^*(x) = \{a\} \cup x \) and \( s'(\psi(x)) = \{b\} \cup x \).

Clearly \( \psi \) defines almost an \( O - \Sigma \) partial isomorphism from \( (A, O^*) \) to \( (A, O) \), with the property that \( \psi(x) = x^{15} \) and \( \psi(y) = y' \). The only problem is that it is not surjective. First, we take its restriction to \( (A, O) \); obviously, \( \psi(x) \) is still defined, and we still have \( \psi(y) = y' \). Next, we argue that there is a finite \( \psi_0 \subseteq \psi \), with these two properties. And finally, we extend \( \psi_0 \), arbitrarily, to an \( O \)-isomorphism.

We have a similar proposition for computable queries.

\[15\text{Here we use the fact that } \sigma \text{ doesn't mention } n.\]
Definition 14 Let $f : \sigma \rightarrow \tau$ be a computable $(\Sigma \cup n)$-query, in which $\sigma$ doesn't mention $n$. Define $\tilde{f} : \sigma \rightarrow \bar{\tau}$ to be the following $O - \Sigma$-query\(^{16}\):

$$\tilde{f}(\sigma, \tau) := \bigcup \{ f(\sigma, N, s, z) / s \text{ recursive, and } ((\sigma, N, s, z), (\sigma, \tau)) \in U \}$$

Proposition 10 $\tilde{f}(\sigma, \tau)$, as defined above, is indeed a $O - \Sigma$-query.

Proof Clearly $\tilde{f}(\sigma, \tau)$ is r.e., because we can restrict ourselves to finite $s$, which we can enumerate.

The rest of the proof goes as in proposition 9, with the following changes:

1. To prove $\tilde{f}(\sigma, \tau) \subseteq \psi \circ \tilde{f}(\sigma, \tau)$, it suffice to prove that, for any $(\Sigma \cup n)$-structure $(B, N', s^B, z^B)$ in relation $U$ with $(B, \sigma')$, with a finite $s^B$, there is a structure $(A, N, s^A, z^A)$, in relation $calU$ with $(A, \sigma)$, such that $\psi$ is a partial isomorphism between them. We can still take $s^A(x) := \psi_{-1} \circ s^B \circ \psi(x)$, because $s^A$ will be finite, hence computable.

2. Again, it suffices to consider $s$ and $s'$ to be finite. Now it suffices to extend $s$ to some $s^*$ just as much as to allow for a partial isomorphism from $(A, N, s^*, z)$ to $(A, N, s', z)$. Namely, define $\psi : O \rightarrow O$ to be closed under the following: whenever $\psi(x)$ and $s'(\psi(x))$ are both defined (say $s'(\psi(x)) = \{b\} \cup \psi(x)$), if $s(x)$ is defined ($s(x) = \{a\} \cup x$), then define $\psi(a) := b$; else define $s^*(x) := \{a\} \cup x$, for some fresh $a$, and $\psi(a) := b$.

After a finite number of steps, we get a finite extension $s^*$ of $s$, together with a partial isomorphism $\psi$.

We conclude the translation with the remark that the two overloaded notations $\tilde{f}$ for $f$ in $\mathcal{NRA}(\Sigma \cup n)$ $(+fix)$ indeed coincide: if we consider the $(\Sigma \cup n)$-query associated with $f$, and then translate it to an $O - \Sigma$-query, as in definition 14, we get exactly the $O - \Sigma$-query associated to the syntactic expression $f$ in $\mathcal{NRA}(\Sigma) + invent (+fix)$.

6.2 Converting $\mathcal{NRA}(\Sigma) + fix + invent$ into $\mathcal{NRA}(\Sigma \cup n) + fix$

This translation is less general than the one described above:

On types: $b := b$, $\bar{\sigma} := \{n\}$, $\bar{\sigma} \times \bar{\rho} := \bar{\sigma} \times \bar{\tau}$, $\{\bar{\sigma}\} := \{\bar{\sigma}\}$.

On functions: we translate some function $f : \sigma \rightarrow \tau$ into $\tilde{f} : \bar{\sigma} \rightarrow \bar{\tau}$, i.e. a relation. We cannot do better than that, because of the translation of $\text{map}$.

First, we define an auxiliary translation, of $f : \sigma \rightarrow \tau$ into $\tilde{f} : \bar{\sigma} \times \bar{\sigma} \rightarrow \bar{\tau}$. For its motivation, recall that the idea of the translation is to represent an object from $o$ by a finite set of “naturals”, i.e. an element of $\{n\}$. As we have to “invent” distinct objects in distinct, independent parts of some function, we

\(^{16}\)Note that $\bar{\sigma} = \sigma$.
pass to any function $f$ two additional arguments, say $p, n \in \{n\}$, containing all numbers $f$ should include ($p$) in any object it invents, as well as all numbers it should avoid when generating objects ($n$).

The following function will be useful in the sequel: $next: \{n\} \times \{n\} \rightarrow \{n\}$, with the properties: $p \subseteq next(p, n), n \cap next(p, n) = \emptyset$ and $p \neq next(p, n)$. $next$ can be defined by: $next(p, n) := fix(\lambda((p, n), w).if \ w-p-n \neq \emptyset \ then \ w \ else \{z\} \cup \ p \cup \text{map}(s)(w))\left(p - n\right)$. Also, let $\text{objects}_{\sigma} : \sigma \rightarrow \{n\}$ be the function returning the union of all objects mentioned in its argument.

The most interesting parts of the translation $f$ are:

- $i\text{invent}(), p, n) := next(p, n)$.
- $g \circ f(x, p, n) := ext(g)(\rho_1(f(x, (p, n)), p', n))$, where $p' := p \cup \text{objects}(f(x, p, n))$.
- $\text{map}(f)\{x_1, \ldots, x_k\}, p, n)$ is defined as follows. First, generate the set \{z, s(z), \ldots, s^{k-1}(z)\}. Then apply repeatedly $s$ on each element of this set, until none of its elements is in $p \cup n$. Let $g = \{c_1, \ldots, c_k\}$ be the resulting set. Next, for each permutation $i_1, \ldots, i_k$ of $g$, compute $\{f(x_i, p \cup \{c_{i_1}\}, n \cup g - \{c_{i_1}\})/i = 1, k\}$. Thus, when we invent objects for $f(x_i)$, we include $c_{i_1}$ but do not include all other $c$'s, assuring that different objects are invented for different $x_i$'s. As we cannot choose some particular permutation, the best we can do is to construct the collection of all such sets; hence the result type of $\text{map}(f)\{x_1, \ldots, x_k\}, p, n)$ is $\{\{\tau\}\}$ instead of $\{\tau\}$.
- $(f, g)$ is defined in a similar, but simpler, manner.

Finally, the translation of $f: \sigma \rightarrow \tau$ is simply $\tilde{f} : \sigma \rightarrow \{\tilde{\tau}\}$, $\tilde{f}(x) := \tilde{f}(x, \phi, \phi)$. The translation is characterized by the following, obvious, proposition:

**Proposition 11** Let $f$ be in $NRA(\Sigma) + fix + \text{invent}$ and $\tilde{f}$ the above translation. For any $(\Sigma \cup n)$-model $(A, N, s, z)$, we have $f(A, N, s, z) \subseteq \tilde{f}(A, O)$, where $O = \mathcal{P}_{\text{fin}}(N)$.

## 7 Conclusions

We have investigated an extension of the definition of a generic database transformation, to databases with user-defined functions. The definition naturally captures the property of genericity, domain independence and monotonicity in the user-defined functions. We prove that the computable queries coincide with those expressible in a (generalized version of the) relational machine (\cite{4}). As a consequence, complete query languages remain complete when extended in a "reasonable" way with user-defined functions. We gave an example of a complete query language, and investigated its relationship with object inventions.

Unfortunately, the simulation of a Turing machine $T$ by a relational machine $RM$ breaks any reasonable complexity class. Thus, we don't know whether
query languages which are complete w.r.t. to certain complexity classes (like \(P\text{Space}\), or the class of Kalmar elementary functions) remain still complete, when extended with user-defined functions. We intend in the future to answer this question, by replacing the simulation done in theorem 1, with a different simulation for each complexity class.

References


