Bounded Fixpoints for Complex Objects

Dan Suciu*
AT&T Labs
suciu@research.att.com

Abstract

We study a query language for complex-object databases, which is designed to (1) express only tractable queries, and (2) be as expressive over flat relations as first-order logic with fixpoints. The language is obtained by extending the *nested relational algebra*, $\mathcal{NRA}$, with a "bounded fixpoint" operator. Similar to results for flat relations, all tractable queries over *ordered* databases are expressible in this language. The main result consists in proving that this language is a conservative extension of the first-order logic with fixpoints, or of the *while*-queries (depending on the interpretation of the bounded fixpoint: inflationary or partial). That is, a query from flat relations to flat relations is expressible in our language if and only if it is expressible in first-order logic with fixpoints, or in the *while*-queries respectively. The proof technique for this theorem uses indexes to encode complex objects into flat relations. It can serve as basis for an implementation method of complex objects databases in terms of relational databases, which works well for queries expressed both with fixpoints and with bounded fixpoint. We also define a complex object logical calculus with fixpoints and prove that its range-restricted fragment is equivalent to $\mathcal{NRA}$ with bounded fixpoints.

Keywords: nested relational algebra, conservative extension, complex objects, fixpoints, indexes.

1 Introduction

Several query languages for databases with complex objects have been studied in recent years [AGVW89, AK89, AB88, BNTW95, dB92, GV91, GF88, GG91, GG92, HS89, HS91, PG92]. A natural way of designing such a language is to extend first-order logic to a logic for hereditary finite sets, and consider only *domain-independent* queries, like in the case of first-order logic. Abiteboul and Beeri follow this path in [AB88], define *safe* queries, and show that the resulting language, which they call *the calculus*, can express *powerset*, hence, unlike first-order logic, it can express intractable queries. They design an algebraic language equivalent to *the calculus*, called *the algebra*, in which *powerset* is one of the primitives. Searching for a tractable sublanguage, they observe that by removing the *powerset* from the algebra one gets such a language, and that the same expressive power can be obtained by enforcing stricter safety rules in the calculus (the resulting language is called *the strictly safe calculus*).

*This work was done while the author was at the University of Pennsylvania, and was partially supported by NSF Grant CCR-90-57570

Another way of designing query languages for complex objects is followed by Buneman et al. [BNTW95]. Based on an investigation of the primitives associated with the types occurring in complex objects, they develop a hierarchy of languages, one of which has the same expressive power as the algebra without *powerset* of [AB88]. Following established tradition, they call this language the *nested relational algebra* ($\mathcal{NRA}$), and show that, like in first-order logic, all queries expressible in $\mathcal{NRA}$ are in $PTIME$.

Paredaens and Van Gucht in [PG92] and Wong in [Won93] show that $\mathcal{NRA}$is a *conservative extension* of first-order logic. That means that a query mapping flat relations to flat relations is expressible in $\mathcal{NRA}$ if and only if it is expressible in first-order logic. This result is surprising, because such queries in $\mathcal{NRA}$ may have intermediate results which are not flat relations. In consequence one cannot express more queries in $\mathcal{NRA}$ by using intermediate results of a higher set complexity. For example, the transitive closure of a flat relation cannot be computed in $\mathcal{NRA}$, because it cannot be computed in first-order logic. In contrast, Abiteboul and Beeri show [AB88] that, by using *powerset*, one can express transitive closure in the *algebra*. Hull and Su [HS91] prove even more: the expressive power of the *algebra* strictly increases as we allow intermediate types of higher set complexity.

Transitive closure can be expressed in first-order logic *with inflationary fixpoints*, a language equivalent to $DATALOG^{\neg}$ (see [AV91a] for a review). Adding fixpoints to first-order logic keeps the queries tractable, but adding them to $\mathcal{NRA}$ does not; in fact Gyssens and Van Gucht [GG92] show that fixpoints are equivalent to *powerset*. It is natural to search for an extension of $\mathcal{NRA}$ which is still tractable and which has at least the expressive power of $DATALOG^{\neg}$. Peter Buneman suggested adding a *bounded fixpoint*. When $f : \mathcal{P}(D) \to \mathcal{P}(D)$ is a function, and A is a subset of D, then the *bounded* fixpoint of f is the fixpoint of the function $g(x) \stackrel{\text{def}}{=} f(x) \cap A$. In this paper we study the language $\mathcal{NRA}$ extended with bounded fixpoints. We consider both *inflationary* and *partial* semantics for the (bounded) fixpoints, and we denote them with $bfix_i$ and $bfix_p$. An immediate consequence of their definition is that queries expressed with $bfix_i$ are in $PTIME$ and queries expressed with $bfix_p$ are in $PSPACE$.

Our main result consists in proving that $\mathcal{NRA} + bfix_i$ is a conservative extension of first-order logic with inflationary fixpoints. Similarly, $\mathcal{NRA} + bfix_p$ is a conservative extension of first-order logic with partial fixpoints. These results holds for *arbitrary* databases (with or without order): in the presence of order, they would be immediate consequences of Immerman and Vardi's results, that first-order logic with inflationary (partial) fixpoints can express all $PTIME$ ($PSPACE$) queries over ordered databases [Imm86, Var82]. For the proof we use a different technique than those used by Paredaens and Van Gucht [PG92] and Wong [Won93], because it is not clear how to adapt their reduction methods for the bounded fixpoint. Our approach is of a semantic nature, and consists of a two steps translation of $\mathcal{NRA} + bfix$ into the relational algebra with fixpoints. First, we encode all complex object into flat relations, using indexes from some infinite set I, and translate queries from $\mathcal{NRA} + bfix$ into queries in the relational algebra with fixpoints and indexes. Second, we eliminate the indexes from the translated queries. The index set I is not ordered, but has some algebraic structure: the purpose of the second step is to rephrase queries such as to avoid using this structure. By composing the two steps we obtain a translation of $\mathcal{NRA} + bfix$ into the relational algebra with fixpoints (without indexes).

An immediate consequence of the conservativity property is the fact that there are $PTIME$ queries which are not expressible in $\mathcal{NRA} + bfix$, for example *parity*. However, as in the flat case [Var82, Imm86], we can prove that any $PTIME$, or $PSPACE$ query over *ordered* databases is expressible in $\mathcal{NRA} + bfix_i$, or $\mathcal{NRA} + bfix_p$ respectively.

The translation used in the proof is of independent interest, because it can be used to implement complex objects databases in terms of relational databases. First the complex objects database is *flattened*, that is encoded as a relational database with indexes. Next any complex objects query is translated into a relational query on the encodings. The technique is robust enough to work for databases with *external functions*, and it can even work for queries with (unbounded) fixpoints, except that for the latter we need the algebraic structure on the indexes.

Two other tractable query languages with fixpoints have been proposed for complex objects. First, Gyssens and Van Gucht discuss the *restricted fixpoint closure of the nested algebra* [GG91]: here the fixpoint construct has some simple syntactic restrictions, namely it may not mention *nest* and *unnest*. Using our conservativity result for bounded fixpoints it has been recently proven that the restricted fixpoint has the same expressive power as the bounded fixpoint [GSG95]. In consequence all properties for the bounded fixpoints reported in this paper carry over to the restricted fixpoints. Second, Grumbach and Vianu discuss *range-restricted* $CALC + IFP$ [GV91], which relies on a carefully designed set of range-restriction rules for the fixpoint. Its relationship to the bounded fixpoints is open. As a step towards a comparison, we present in Section 6 a range-restricted, logic-based calculus with fixpoints which we show to be equivalent to $\mathcal{NRA} + \mathit{bfix}$: our range-restriction rules differ from those in [GV91].

Abiteboul and Bidoit [AB86] consider a query language for complex objects, VERSO, where each instance is required to have a scalar key: our indexes are in the same spirit as the scalar keys in VERSO. The idea of encoding nested relations into flat relations with indexes was first introduced by Van Gucht and Fischer [GF88]. Van den Bussche [dB92] uses indexes with *index inventions* for proving a conservativity result for the nested algebra (without fixpoints): by contrast, we do not use index inventions here.

In Section 2, we review the basic definitions and properties of the nested relational algebra, $\mathcal{NRA}$, and the nested relational calculus, $\mathcal{NRC}$, following [BNTW95]: we define $\mathcal{NRC}$ in Subsection 2.1 and $\mathcal{NRA}$ in Subsection 2.2, show their equivalence in Subsection 2.3, and review some of their properties in Subsection 2.4. In Section 3 we introduce fixpoints and bounded fixpoints: the definitions and basic properties are given in Subsection 3.1, the equivalence between various forms of the fixpoint constructs are proven in Subsection 3.4, and the expressive power of fixpoints and bounded fixpoints is investigated in Subsection 3.5. Section 4 states our main results. The proof of the conservativity theorem (Theorem 4.1) is given in Section 5. We start by presenting the relational algebra with indexes, then show how to encode complex objects into flat relations with indexes in Subsection 5.1, show how to translate queries over complex objects into queries over flat relations with indexes in Subsection 5.2, and finally show how to eliminate the indexes in Subsection 5.3. In Section 6 we present a logic-based language with fixpoints, whose range-restricted fragment is equivalent to the nested relational algebra with bounded fixpoints.

2 The Nested Relational Algebra

In this section we review the Nested Relational Algebra $\mathcal{NRA}$ and the Nested Relational Calculus $\mathcal{NRC}$ as presented in [BNTW95], slightly adapted to our needs. We also present a different technique for proving the equivalence of $\mathcal{NRA}$ and $\mathcal{NRC}$.

Both $\mathcal{NRC}$ and $\mathcal{NRA}$ are strongly typed languages. Their **types**, called *complex object types*, are constructed from a set of base types, like $\mathbb{N}$, *string*, etc., by the grammar:

$$t ::= \mathbb{D} \mid unit \mid t \times t \mid \{t\}$$

$\mathbb{D}$ stands for one of the base types, *unit* contains only one value, the empty tuple $()$, $s \times t$ is the product type and contains all pairs (x, y) where $x \in s$ and $y \in t$, and $\{t\}$ contains all finite sets $\{x_1, \ldots, x_n\}$, where $x_1, \ldots, x_n \in t$. We feel free to omit parentheses and write $t_1 \times \ldots \times t_n$ for $(\ldots((t_1 \times t_2) \times t_3) \ldots \times t_n)$.

We say that a type is a **scalar type** if it is a product of base types. We say that it is a **flat relation type** if it is a product of types of the form $\{s\}$ with s a scalar type. For example $\{\mathbb{D} \times \mathbb{D}\}$ and $\{\mathbb{D}\} \times \{\mathbb{D} \times \mathbb{D} \times \mathbb{D}\}$ are flat relation types. For a given set of base types, we consider a set of *constants*, written $c : t_c$, and *functions*, $p : d_p \to c_p$; in this notation t_c is the type of the constant c, while the types d_p, c_p are the domain and the codomain of the function p. Examples of constants and external functions are: *cardinality* at type s ($card_s : \{s\} \to \mathbb{N}$), *addition* ($add : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$), the constant *zero* ($0 : \mathbb{N}$).

The languages $\mathcal{NRC}$ and $\mathcal{NRC}$, which we define shortly, will thus depend on (1) the set of base types, and (2) the set of constants and external functions; we denote by Σ the set of base types, constants, and external functions, on which the languages are parameterized.

2.1 The Nested Relational Calculus, $\mathcal{NRC}$

The Nested Relational Calculus, $\mathcal{NRC}$, is a language with variables. The name "calculus" should be understood in the spirit of the "lambda-calculus", i.e. a notation for values and functions, and not in the spirit of the "relational calculus", since it is not a logic-based language. For each type t, assume an infinite set of variables to be given. We write x^t to emphasize that the variable x has type t. The expressions of the Nested Relational Calculus $\mathcal{NRC}$ fall into two distinct syntactic categories: *terms* and *functions*. A term e will have some type t, and we write $t : e$, while some function f will be characterized by two types: its domain s and its codomain t, written $f : s \to t$.

$\mathcal{NRC}(\Sigma)$ is defined by the rules in Figure 1. We abbreviate it with $\mathcal{NRC}$ when Σ contains only base types and constants (i.e. no external functions).

The meaning of these expressions is the following: (e_1, e_2) denotes a pair; $\pi_1^{t_1,t_2}, \pi_2^{t_1,t_2}$ are the projections, $\pi_1^{t_1,t_2}(x, y) \stackrel{\text{def}}{=} x$, $\pi_2^{t_1,t_2}(x, y) \stackrel{\text{def}}{=} y$; $\emptyset$ is the empty set; $\{e\}$ denotes the singleton set; $e_1 \cup e_2$ is the union of e_1 and e_2; $ext(f)(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} f(x_1) \cup \ldots \cup f(x_n)$.

The lambda expression $\lambda x^s.e$ denotes a function in which x^s is the input variable. As usual, we distinguish between *free* and *bound* variables: x^s becomes bound in $\lambda x^s.e$. The term $f(e)$ denotes function application.

Note that there are only two things we can do with some lambda expression: apply it immediately, like in $f(e)$, or "extend" it, like in $ext(f)$. It is the second construct ($ext(f)$) for which lambda expressions were introduced into the language [BNTW95].

Again following [BNTW95], we use the type $\{unit\}$ as a *boolean* type, by considering $\emptyset$ to stand for *false* and the singleton set $\{()\}$ to stand for *true*. Then, the expression $e_1 = e_2$ returns *true* (i.e. $\{()\}$) iff e_1 and e_2 are equal. In the same spirit, $not(e)$ returns $\{()\}$ when $e = \emptyset$, and $\emptyset$ when $e = \{()\}$. Note that the equality is a primitive in the language only for elements of the base types; we shall see that it can be lifted at all types.

$$\frac{}{x^t : t}\text{(for every variable } x^t \text{ of type } t\text{)}$$

$$\frac{}{c : t_c}\text{(for every constant } c \text{ in } \Sigma\text{)} \qquad \frac{e : d_p}{p(e) : c_p}\text{(for every function } p : d_p \to c_p \text{ in } \Sigma\text{)}$$

$$\frac{e_1 : t_1 \quad e_2 : t_2}{(e_1, e_2) : (t_1, t_2)} \qquad \frac{e : t_1 \times t_2}{\pi_1^{t_1,t_2}(e) : t_1} \qquad \frac{e : t_1 \times t_2}{\pi_2^{t_1,t_2}(e) : t_2} \qquad \frac{}{() : \mathit{unit}}$$

$$\frac{}{\emptyset : \{t\}} \qquad \frac{e : t}{\{e\} : \{t\}} \qquad \frac{e_1 : \{t\} \quad e_2 : \{t\}}{e_1 \cup e_2 : \{t\}} \qquad \frac{f : s \to \{t\}}{\mathit{ext}(f) : \{s\} \to \{t\}}$$

$$\frac{e : t}{\lambda x^s.e : s \to t} \qquad \frac{f : s \to t \quad e : s}{f(e) : t}$$

$$\frac{e_1 : \mathbb{D} \quad e_2 : \mathbb{D}}{e_1 = e_2 : \{\mathit{unit}\}} \qquad \frac{e : \{\mathit{unit}\}}{\mathit{not}(e) : \{\mathit{unit}\}}$$

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

We define a **query** to be a closed function $f : s \to t$ (i.e. with no free variables). This is a minor deviation from the traditional notion of a query [AHV95], where the type is required to be $\{s_1\} \times \ldots \times \{s_k\} \to \{t\}$, and allows us to state our main result in a slightly stronger form.

We shall adopt abbreviations in our language. We drop the type superscripts, e.g. writing π_1 instead of $\pi_1^{t_1,t_2}$. Also, we consider generalized projections $\pi_i^{t_1,\ldots,t_n} : t_1 \times \ldots \times t_n \to t_i$, $i = 1, n$, which are expressible in $\mathcal{NRC}(\Sigma)$. We allow for a more liberal notation in lambda abstractions, like in the function $\mathit{swap} : s \times t \to t \times s$ which we write as $\mathit{swap} = \lambda(x^s, y^t).(y^t, x^s)$, instead of $\lambda z^{(s,t)}.(\pi_2(z^{(s,t)}), \pi_1(z^{(s,t)}))$.

Traditional database operators, like product, set difference, selection, and even database projections are not among the primitives of $\mathcal{NRC}(\Sigma)$. In the rest of this subsection we illustrate how they can be expressed, mostly using the versatile $\mathit{ext}(f)$ construct.

Example 2.1

1. *For some $f : s \to t$, define $\mathit{map}(f) : \{s\} \to \{t\}$ by $\mathit{map}(f) = \mathit{ext}(\lambda(x^s).\{f(x^s)\})$. The meaning of $\mathit{map}(f)$ is: $\mathit{map}(f)(\{x_1, \ldots, x_n\}) \stackrel{\text{def}}{=} \{f(x_1), \ldots, f(x_n)\}$.*

2. *The database projections $\Pi_1 : \{s \times t\} \to \{s\}$ and $\Pi_2 : \{s \times t\} \to \{t\}$ are defined by $\Pi_1 = \mathit{map}(\pi_1)$ and $\Pi_2 = \mathit{map}(\pi_2)$. Again, we use abbreviations here. The complete expression is:*

$$\Pi_1 = \mathit{map}(\lambda x^{s \times t}.\pi_1(x^{s \times t}))$$

3. *The "pair with" function $\rho_2 : s \times \{t\} \to \{s \times t\}$, with the meaning:*

$$\rho_2(x, \{y_1, \ldots, y_n\}) \stackrel{\text{def}}{=} \{(x, y_1), \ldots, (x, y_n)\}$$

is defined by: $\rho_2 = \lambda(x^s, Y^{\{t\}}).(\mathit{ext}(\lambda y^t.(x^s, y^t))(Y^{\{t\}}))$. Note the essential role of the free variable x^s in the function $\lambda y^t.(x^s, y^t)$.

4. *For $x : \{s\}, y : \{t\}$, the cartesian product, which we write $x \bowtie y : \{s \times t\}$ (instead of $x \times y$), is defined by* $\bowtie \stackrel{\text{def}}{=} \lambda(x^{\{s\}}, y^{\{t\}}).ext(\lambda u^s.\rho_2(u^s, y^{\{t\}}))(x^{\{s\}})$.

5. *Let $p : s \to \{unit\}$ be some predicate in the language. Then, we can define the selection over the predicate p, $\sigma_p : \{s\} \to \{s\}$, as follows. First, define $q : s \to \{unit \times s\}$ by $q = \lambda x.p(x) \bowtie \{x\}$, with the meaning $q(x) = \{((), x)\}$ when $p(x)$ is "true", and $q(x) = \emptyset$ when $p(x)$ is "false". Then $\sigma_p = ext(\lambda x.\Pi_2(q(x)))$.*

6. *All boolean operations at type $\{unit\}$ are easily definable. For example $x \vee y \stackrel{\text{def}}{=} x \cup y$, $x \wedge y \stackrel{\text{def}}{=} not(not(x) \cup not(y))$.*

It has been previously recognized [GG91, BNTW95] that only one non-monotone operator suffices to express all other non-monotone operators in $\mathcal{NRC}(\Sigma)$, e.g. it would have sufficed to include only equality ($=$) at all types in the language. But equality at arbitrary types turns out to be more difficult to handle in the proof of the conservativity theorem, hence we adopt here a presentation in which it is only defined at base types. To lift it at arbitrary types we need the additional non-monotone operator $not : \{unit\} \to \{unit\}$. Thus, our first goal is to prove that the predicates $member, inclusion, =$ are definable at all types t. The next lemma provides the induction step.

Lemma 2.2 *Suppose that an equality predicate ($=$) can be defined in $\mathcal{NRC}(\Sigma)$ at type t, i.e. $e_1 = e_2 : \{unit\}$ whenever $e_1, e_2 : t$, with the obvious meaning. Then the following predicates are definable in $\mathcal{NRC}(\Sigma)$ too: $member : t \times \{t\} \to \{unit\}$. $member(x, y)$ returns true iff $x \in y$; $inclusion : \{t\} \times \{t\} \to \{unit\}$. $inclusion(x, y)$ returns true iff $x \subseteq y$; equality at type $\{t\}$: $=: \{t\} \times \{t\} \to \{unit\}$.*

Proof. We provide the required expressions below:

$$\begin{aligned}
member &\stackrel{\text{def}}{=} \lambda(x^t, Y^{\{t\}}).ext(\lambda(y^t).x^t = y^t)(Y^{\{t\}}) \\
inclusion &\stackrel{\text{def}}{=} \lambda(X^{\{t\}}, Y^{\{t\}}).not(ext(\lambda x^t.not(member(x^t, y^{\{t\}})))(X^{\{t\}})) \\
(X^{\{t\}} = Y^{\{t\}}) &\stackrel{\text{def}}{=} inclusion(X^{\{t\}}, Y^{\{t\}}) \wedge inclusion(Y^{\{t\}}, X^{\{t\}})
\end{aligned}$$

□

Proposition 2.3 *Equality, $member, inclusion$ are definable in $\mathcal{NRC}(\Sigma)$ at all types.*

Proof. The proof is done easily by induction on the type t. If $t = \mathbb{D}$, then equality is in $\mathcal{NRC}(\Sigma)$ by definition, and if $t = unit$, then equality is constant $true$ (expressed as $not(\emptyset)$). For $t = t_1 \times t_2$, define $x^{t_1 \times t_2} = y^{t_1 \times t_2}$ to be $(\pi_1(x^{t_1 \times t_2}) = \pi_1(y^{t_1 \times t_2})) \wedge (\pi_2(x^{t_1 \times t_2}) = \pi_2(y^{t_1 \times t_2}))$. For $t = \{t_1\}$, use the previous lemma. □

Proposition 2.4 *Set intersection, difference, nest, unnest, and if-then-else are definable in $\mathcal{NRC}(\Sigma)$. Their types are: $\cap : \{t\} \times \{t\} \to \{t\}$, $- : \{t\} \times \{t\} \to \{t\}$, $unnest : \{t \times \{t'\}\} \to \{t \times t'\}$, $nest : \{t \times t'\} \to \{t \times \{t'\}\}$, $if - then - else\ : \{unit\} \times \{t\} \times \{t\} \to \{t\}$.*

$$\frac{p \in \Sigma}{p : d_p \to c_p} \qquad \frac{c \in \Sigma}{c : unit \to t_c} \qquad \frac{}{id_s : s \to s} \qquad \frac{f : r \to s \quad g : s \to t}{g \circ f : r \to t}$$

$$\frac{}{\pi_i : s_1 \times s_2 \to s_i}\ (i = 1, 2) \qquad \frac{f_i : s \to t_i\ (i = 1, 2)}{(f_1, f_2) : s \to t_1 \times t_2} \qquad \frac{}{\iota_s : s \to unit}$$

$$\frac{}{\emptyset : unit \to \{s\}} \qquad \frac{}{\eta : s \to \{s\}} \qquad \frac{}{\cup : \{s\} \times \{s\} \to \{s\}}$$

$$\frac{f : r \times s \to \{t\}}{ext_2(f) : r \times \{s\} \to \{t\}}$$

$$\frac{}{not : \{unit\} \to \{unit\}} \qquad \frac{}{eq_{\mathbb{D}} : \mathbb{D} \times \mathbb{D} \to \{unit\}}\ (\mathbb{D} \text{ a base type})$$

Figure 2: The language $\mathcal{NRA}(\Sigma)$.

Following [HS91], we define the **set height of a type** s to be: $sh(\mathbb{D}) = sh(unit) = 0$, $sh(s \times t) = max(sh(s), sh(t))$, $sh(\{s\}) = 1 + sh(s)$. For some $k \geq 0$, we define the restriction of $\mathcal{NRC}(\Sigma)$ to set height $\leq k$, $\mathcal{NRC}_k(\Sigma)$, in the same way as $\mathcal{NRC}(\Sigma)$, but by restricting all types to have set height $\leq k$. Thus $\mathcal{NRC}_1(\Sigma)$ contains all $\mathcal{NRC}(\Sigma)$ expressions in which only types of set height ≤ 1 are mentioned. Obviously, $\mathcal{NRC}(\Sigma) = \bigcup_{k \geq 0}(\mathcal{NRC}_k(\Sigma))$.

2.2 The Nested Relational Algebra, $\mathcal{NRA}$

It is inconvenient to prove properties about queries expressed in $\mathcal{NRC}(\Sigma)$, because they are not inductively defined. Subexpressions of some query are not necessarily queries themselves, because they may have free variables, or may be terms (as opposed to functions). Following [BNTW95], we introduce an equivalent query language, $\mathcal{NRA}(\Sigma)$, in which *every* expression is a query. $\mathcal{NRA}(\Sigma)$ is variable-free, and all expressions are functions $f : s \to t$. Contrary to the tradition, $\mathcal{NRA}(\Sigma)$ is not an algebra of *complex objects*, but an algebra of *functions* over complex objects. The equivalence of $\mathcal{NRC}(\Sigma)$ and $\mathcal{NRA}(\Sigma)$ is is shown in [BNTW95]. Here we show an alternative proof technique, which we later extend to show equivalence of various extensions of $\mathcal{NRC}(\Sigma)$ and $\mathcal{NRA}(\Sigma)$ with fixpoint constructs.

To maintain the same expressive power in the absence of free variables forces us to modify the definition of *ext* in $\mathcal{NRA}(\Sigma)$. Indeed, suppose $g : s \to \{t\}$ in $\mathcal{NRC}(\Sigma)$ has a single free variable, and let r be its type. Then, g can be viewed as a closed function $f : r \times s \to \{t\}$. Now, in $\mathcal{NRC}(\Sigma)$ we can compute $ext(g) : \{s\} \to \{t\}$, which continues to have the same free variable, so it can be viewed as a closed function $h : r \times \{s\} \to \{t\}$. To obtain h directly from f, we introduce in $\mathcal{NRA}(\Sigma)$ the construct $ext_2(f)$, with the syntax:

$$\frac{f : r \times s \to \{t\}}{ext_2(f) : r \times \{s\} \to \{t\}}$$

and the semantics $ext_2(f)(x, \{y_1, \ldots, y_n\}) = f(x, y_1) \cup \ldots \cup f(x, y_n)$.

The language $\mathcal{NRA}(\Sigma)$ is defined in Figure 2. The semantics of most of these functions is straightforward, so we briefly describe here only the less obvious ones: c is the "constant function" associated to some constant symbol in Σ, such that $c(()) = c$; $id_s(x) = x$ is the identity function; $g \circ f(x) = g(f(x))$ is function composition; $(f_1, f_2)(x) = (f_1(x), f_2(x))$ pairs two functions; $\iota_s(x) = ()$ returns the empty tuple for every input; $\emptyset(()) = \{\}$ returns the empty set; and $\eta(x) = \{x\}$ returns a singleton set.

A **query** in $\mathcal{NRA}(\Sigma)$ is simply a function $f : s \to t$ constructed according to the rules above. As for $\mathcal{NRC}(\Sigma)$, we define $\mathcal{NRA}_k(\Sigma)$, for $k \geq 0$, to be the restriction of $\mathcal{NRA}(\Sigma)$ to those queries which only mention types of set height $\leq k$.

A more elegant alternative is to replace ext_2 with two nicer constructs:

$$\frac{f : s \to \{t\}}{ext(f) : \{s\} \to \{t\}} \qquad \frac{}{\rho_2 : s \times \{t\} \to \{s \times t\}}$$

where $ext(f)$ has the same meaning as in $\mathcal{NRC}(\Sigma)$, and $\rho_2(x, \{y_1, \ldots, y_n\}) = \{(x, y_1), \ldots, (x, y_n)\}$ (see Example 2.1). Indeed, for $f : r \times s \to \{t\}$, we can define $ext_2(f) : r \times \{s\} \to \{t\}$ as $ext(f) \circ \rho_2$, while for the converse we define $\rho_2 = ext_2(\eta)$. Another alternative is to further replace the ext construct with the following two:

$$\frac{f : s \to t}{map(f) : \{s\} \to \{t\}} \qquad \frac{}{\mu : \{\{s\}\} \to \{s\}}$$

with the semantics: $map(f)(\{x_1, \ldots, x_n\}) = \{f(x_1), \ldots, f(x_n)\}$ and $\mu(\{x_1, \ldots, x_n\}) = x_1 \cup \ldots \cup x_n$. Indeed $map(f) = ext(\eta \circ f)$, $\mu = ext(id)$, and, for the converse, $ext(f) = \mu \circ map(f)$. We can write:

$$ext_2 = ext + \rho_2 = map + \mu + \rho_2$$

and this gives us three different presentations of the same language $\mathcal{NRA}(\Sigma)$. However, they differ when we impose restrictions on the set height, because the conversions described above use larger set height as intermediate results. This is why for $\mathcal{NRA}_k(\Sigma)$ we assume the ext_2 presentation, which is the only one for which we prove $\mathcal{NRA}_k(\Sigma) = \mathcal{NRC}_k(\Sigma)$ in Proposition 2.5 below.

Finally, note that $\cup$ can be replaced with a simpler operation:

$$\frac{}{doubleton : s \times s \to \{s\}}$$

with the semantics $doubleton(x, y) \stackrel{\text{def}}{=} \{x, y\} = \{x\} \cup \{y\}$; indeed, $\cup = \mu \circ doubleton$ (but, again, an intermediate type with a larger set height is used). We shall exploit this in Section 5.

2.3 Equivalence of $\mathcal{NRC}$ and $\mathcal{NRA}$

Continuing our review of $\mathcal{NRC}(\Sigma)$ and $\mathcal{NRA}(\Sigma)$ we prove their equivalence, using a different technique from [BNTW95].

Proposition 2.5 *For all $k \geq 0$, $\mathcal{NRC}_k(\Sigma) = \mathcal{NRA}_k(\Sigma)$. As a consequence, $\mathcal{NRC}(\Sigma) = \mathcal{NRA}(\Sigma)$.*

Proof. The proposition says that any query is expressible in $\mathcal{NRC}(\Sigma)$ iff it is expressible in $\mathcal{NRA}(\Sigma)$. Note that $\mathcal{NRC}(\Sigma)$ also contains open terms (expressions with variables), or subexpressions with free variables, which do not have any correspondent in $\mathcal{NRA}(\Sigma)$.

1. $\mathcal{NRA}_k(\Sigma) \subseteq \mathcal{NRC}_k(\Sigma)$. This is inclusion is shown by induction on the structure of some query $f \in \mathcal{NRA}(\Sigma)$. We only show some of the cases. An external function, $p : d_p \to c_p$, from $\mathcal{NRA}(\Sigma)$ is translated into $\lambda x^{d_p}.p(x^{d_p})$ in $\mathcal{NRC}(\Sigma)$, the identity, id_s, is translated into $\lambda x^s.x^s$. For composition, suppose $f : r \to s$ is translated into f', and $g : s \to t$ into g'. Then we translate $g \circ f : r \to t$ into $\lambda x^r.g'(f'(x^r))$ in $\mathcal{NRC}(\Sigma)$. For ext_2, suppose $f : r \times s \to \{t\}$ is translated into f'. Then translate $ext_2(f) : r \times \{s\} \to \{t\}$ into $\lambda(x^r, Y^{\{s\}}).ext(\lambda y^s.f'(x^r, y^s))(Y^{\{s\}})$. The other cases are trivial.

2. $\mathcal{NRC}_k(\Sigma) \subseteq \mathcal{NRA}_k(\Sigma)$. Here is where our method differs from that in [BNTW95]. We prove simultaneously by induction on the structure of a term expression $e : t$ and of a function expression $f : s \to t$ in $\mathcal{NRC}(\Sigma)$ the following statement: for every set $x^{s_1}, \ldots, x^{s_n}$ of variables containing all free variables of e (respectively f), the query $\lambda(x^{s_1}, \ldots, x^{s_n}).e$ (respectively $\lambda(x^{s_1}, \ldots, x^{s_n}, x^s).f(x^s)$) is expressible in $\mathcal{NRA}(\Sigma)$. Again, we only show some relevant cases. Suppose $e = x^t$ (i.e. e is a variable). Then it must be the case that x^t is one of the variables x^{s_i}, and we define the corresponding function in $\mathcal{NRA}(\Sigma)$ to be the projection $\pi_i^{s_1,\ldots,s_n} : s_1 \times \ldots \times s_n \to s_i$. For the case $e = (e_1, e_2)$, let f_1, f_2 be the translations, into $\mathcal{NRA}(\Sigma)$, of $\lambda(x^{s_1}, \ldots, x^{s_n}).e_1$ and $\lambda(x^{s_1}, \ldots, x^{s_n}).e_2$ respectively. Then, the translation of $\lambda(x^{s_1}, \ldots, x^{s_n}).(e_1, e_2)$ is (f_1, f_2). A more interesting case is $ext(f)$, where $f : s \to \{t\}$. Suppose $f' : (s_1 \times \ldots \times s_n) \times s \to \{t\}$ is the translation of $\lambda(x^{s_1}, \ldots, x^{s_n}, x^s).f(x^s)$. Then $\lambda(x^{s_1}, \ldots, x^{s_n}, X^{\{s\}}).ext(f)(X^{\{s\}})$ is expressible in $\mathcal{NRA}(\Sigma)$, namely as $ext_2(f')$. The lambda abstraction case is trivial: suppose $f : s_1 \times \ldots \times s_n \to t$ is the translation of $\lambda(x^{s_1}, \ldots, x^{s_n}).e$. Then the translation of $\lambda(x^{s_1}, \ldots, x^{s_n}, x^s).(\lambda x^s.e)(x^s)$ is also f. Also, function application $(f(e))$ in $\mathcal{NRC}(\Sigma)$ translates into function composition in $\mathcal{NRA}(\Sigma)$, etc.

$\square$

As a consequence, in the rest of the paper we shall switch back and forth between $\mathcal{NRA}(\Sigma)$ and $\mathcal{NRC}(\Sigma)$. We shall favor the syntax of $\mathcal{NRC}(\Sigma)$ in examples. When proving properties of the language, we shall favor $\mathcal{NRA}(\Sigma)$, because here the queries have an inductive structure. Following established tradition we shall refer to the language as *Nested Relational Algebra*, and denote it with $\mathcal{NRA}(\Sigma)$.

2.4 Properties of $\mathcal{NRA}$

We enumerate below some known results about the nested relational algebra, relevant to this paper.

Proposition 2.6 *[BTBN91] $\mathcal{NRA}_1$ has, essentially, the same expressive power as the relational algebra.*

The only difference between $\mathcal{NRA}_1$ and the relational algebra is that in $\mathcal{NRA}_1$ we can express "queries" with scalar inputs and/or scalar outputs. Over traditional queries (mapping flat relations to flat relations), the $\mathcal{NRA}_1$ and the relational algebra coincide.

Proposition 2.7 *[BNTW95]* $\mathcal{NRA}$ *has essentially the same expressive power as Schek and Scholl's* NF^2 *relational algebra [SS86], as Thomas and Fischer's algebra [TF86], and as Paredaens and Van Gucht's* nested algebra *[PG88, PG92].*

Proposition 2.8 *[BNTW95] All queries in* $\mathcal{NRA}$ *are in PTIME (computable in polynomial time on a Turing Machine).*

Theorem 2.9 *[Won93, PG92]* $\forall k \geq 1$*, if all function symbols in* Σ *have set height* $\leq k$*, then* $\mathcal{NRA}(\Sigma)$ *is a conservative extension of* $\mathcal{NRA}_k(\Sigma)$

Theorem 2.9 is a nontrivial result which implies that, over flat types, one cannot compute more in $\mathcal{NRA}$ than what is already computable in the relational algebra. As classical examples, the following *PTIME* queries are not expressible in $\mathcal{NRA}$: transitive closure $tc : \{\mathbb{D} \times \mathbb{D}\} \to \{\mathbb{D} \times \mathbb{D}\}$, and parity $parity : \{\mathbb{D}\} \to \{unit\}$, because they are not expressible in the relational algebra. The condition imposed on the external functions in Σ is obviously necessary. Indeed, suppose that Σ contains as an external function the powerset over binary relations $powerset : \{\mathbb{D} \times \mathbb{D}\} \to \{\{\mathbb{D} \times \mathbb{D}\}\}$ (a function of set height 2). Then, using the technique from [AB88], one can express transitive closure $tc : \{\mathbb{D} \times \mathbb{D}\} \to \{\mathbb{D} \times \mathbb{D}\}$ in $\mathcal{NRA}_2(\Sigma)$, but one cannot express it in $\mathcal{NRA}_1(\Sigma)$, because *powerset* cannot be invoked.

3 Fixpoints

In order to get more expressive languages, first-order logic has been extended with various forms of *fixpoints*. Two such extensions are well known: *inflationary fixpoints*, and *partial fixpoints*. The corresponding languages are denoted with *FO+IFP* and *FO+PFP* respectively. They "capture" the complexity classes *PTIME* and *PSPACE* respectively, in the following sense: all queries expressed in *FO+IFP* are in *PTIME*, and *FO+IFP* can express all *PTIME* queries over *ordered*[1] databases. A similar relationship holds between *FO+PFP* and *PSPACE*. Moreover, Abiteboul and Vianu [AV91b] show that the two forms of fixpoints collapse if and only if *PTIME* = *PSPACE*.

When applied to complex objects however, similar extensions of $\mathcal{NRA}$ with fixpoints do not enjoy the same nice properties: the inflationary and the partial fixpoint collapse, and they can express intractable queries, outside of *PTIME* or *PSPACE*. In fact both forms of fixpoints are equivalent to an extension of $\mathcal{NRA}$ with *powerset*, which is know to express precisely the *elementary queries* [HS91].

To compensate for this shortcoming we consider an alternative fixpoint, called *bounded fixpoint* and denoted *bfix*, suggested to us by Peter Buneman. We consider both inflationary and partial semantics. We show that the bounded fixpoint over complex objects enjoys similar properties to those of the fixpoint extensions of first order logic. More precisely: (1) all queries in $\mathcal{NRA} + bfix$ are in *PTIME* and *PSPACE* respectively, (2) over ordered databases, $\mathcal{NRA} + bfix$ can express *all* queries in *PTIME* or *PSPACE* respectively. Finally, the technically hard result is an extension of the conservativity property in Theorem 2.9: (3) $\mathcal{NRA} + bfix$ is a conservative extensions of *FO+IFP* and *FO+PFP* respectively.

[1] No query language capable of expressing *exactly* the *PTIME* queries on unordered databases is currently known.

Gyssens and Van Gucht [GG91] consider extending a language similar to $\mathcal{NRA}$ with a *restricted least fixpoint*, by allowing the least fixpoint operator to be applied only on functions in which *nest* and *unnest* do not occur. They prove that the *restricted-lfp* cannot express *powerset*, and, in the process, show that *restricted-lfp* is in *PTIME*. The relationship to our bounded fixpoint is studied in [GSG95]: based on our conservativity result for bounded fixpoints, the bounded and the restricted fixpoints are shown to be equivalent. Hence all our results for bounded fixpoints carry over to restricted fixpoints as well.

3.1 Definitions

Fixpoints cannot be defined at all types, but only at set types and products of set types. So we define a **product of sets type** (**PS type**) to be either a set type $\{t\}$, or the product $s \times t$ of two PS types s and t. In short, a PS-type has the form $\{t_1\} \times \ldots \times \{t_m\}$. Examples of PS types are $\{\mathbb{D}\}$, $\{\mathbb{D} \times \{\mathbb{D}\}\} \times \{\mathbb{D}\} \times \{\mathbb{D} \times \mathbb{D}\}$ etc., while $\mathbb{D}$ and $\mathbb{D} \times \{\mathbb{D}\}$ are not PS types. Note that the *flat relation types* (Section 2) are precisely the PS types of set height 1. For some PS type t, we extend the notations $\cup, \cap, \emptyset$ and write $\cup, \cap : t \times t \to t$ and $\emptyset : unit \to t$, for the union and intersection of the components, and for the tuple of empty sets.

Consider extending the Nested Relational Calculus, $\mathcal{NRC}(\Sigma)$ (i.e. the language *with* variables), with the following fixpoint constructions, in which t is a PS type:

$$\frac{f : t \to t}{\mathit{fix}(f) : t} \qquad \frac{f : t \to t \quad e : t}{\mathit{bfix}(f, e) : t}$$

We consider two semantics: the *partial*, and the *inflationary* semantics. In order to distinguish them, we write $\mathit{fix}_p(f)$ and $\mathit{fix}_i(f)$ (respectively $\mathit{bfix}_p(f,e)$ and $\mathit{bfix}_i(f,e)$). For the semantics of $\mathit{fix}_p(f)$, define the sequence $y_0 = \emptyset$ $(\in t)$, $y_{n+1} = f(y_n)$ for $n \geq 0$; if there is some m for which $y_m = y_{m+1}$, then we define the **partial fixpoint** of f to be $\mathit{fix}_p(f) = y_m$. Else, $\mathit{fix}_p(f) =$ undefined. The **inflationary fixpoint** is defined to be the partial fixpoint of the function $g(y) \stackrel{\text{def}}{=} y \cup f(y)$. Equivalently, define $y_0 = \emptyset$, $y_{n+1} = y_n \cup f(y_n)$. Then $\mathit{fix}_i(f) = \bigcup_{n \geq 0} y_n$, if y_n is defined for every $n \geq 0$ and the set $\bigcup_{n\geq 0} y_n$ is finite, and $\mathit{fix}_i(f)$ is undefined otherwise. The **bounded** partial (inflationary) fixpoint of f and e, $\mathit{bfix}(f, e)$, is defined to be the partial (inflationary) fixpoint of the function $h(y) \stackrel{\text{def}}{=} f(y) \cap e$. To distinguish between them, we shall sometimes refer to *fix* as the *unbounded* fixpoint.

To define fixpoints in the variable-free $\mathcal{NRA}(\Sigma)$, we need to account for the free variables in f and/or in e. Hence, we define in $\mathcal{NRA}(\Sigma)$, for some PS-type t:

$$\frac{f : s \times t \to t}{\mathit{fix}(f) : s \to t} \qquad \frac{f : s \times t \to t \quad g : s \to t}{\mathit{bfix}(f, g) : s \to t}$$

with the semantics: $\mathit{fix}_p(f)(x) = y_m$ where $y_0 = \emptyset$, $y_{n+1} = f(x, y_n)$, and m is the smallest (or any) number such that $y_{m+1} = y_m$. Similarly, $\mathit{fix}_i(f)$ is defined to be the partial fixpoint of $f(x,y) \cup y$, and the bounded fixpoints are defined to be the fixpoints of $f(x,y) \cap g(x)$.

Proposition 3.1 *Proposition 2.5 can be extended to fixpoints:*

1. $\mathcal{NRC}(\Sigma) + \mathit{fix}_p = \mathcal{NRA}(\Sigma) + \mathit{fix}_p$.

2. $\mathcal{NRC}(\Sigma) + \mathit{fix}_i = \mathcal{NRA}(\Sigma) + \mathit{fix}_i$.

3. $\mathcal{NRC}(\Sigma) + \mathit{bfix}_p = \mathcal{NRA}(\Sigma) + \mathit{bfix}_p$.

4. $\mathcal{NRC}(\Sigma) + \mathit{bfix}_i = \mathcal{NRA}(\Sigma) + \mathit{bfix}_i$.

More, the above equalities also hold if we restrict to types of set height $\leq k$.

When Σ has no external functions, then all queries in $\mathcal{NRC} + \mathit{fix}_i$ and $\mathcal{NRC} + \mathit{bfix}_i$ are total: but queries expressed with partial fixpoints may be partially defined. When we do have external functions, then some queries in $\mathcal{NRC}(\Sigma) + \mathit{fix}_i$ may be partial too, see Example 3.5.

In the sequel we shall prove the following properties about fixpoints and bounded fixpoints:

1. Over flat relations, and when Σ has no external functions, *fix* and *bfix* coincide: $\mathcal{NRA}_1 + \mathit{fix}_i = \mathcal{NRA}_1 + \mathit{bfix}_i$ and $\mathcal{NRA}_1 + \mathit{fix}_p = \mathcal{NRA}_1 + \mathit{bfix}_p$, (Subsection 3.3). Both conditions are necessary.
2. $\mathcal{NRA}(\Sigma) + \mathit{fix}_i = \mathcal{NRA}(\Sigma) + \mathit{fix}_p$ (Subsection 3.4). When Σ has no external functions, this result was known, because both languages express precisely the *elementary queries* [GG92, HS91, AB88]: that argument does not apply however in the presence of external functions, because both languages can express non-elementary queries.
3. $\mathcal{NRA} + \mathit{bfix}_i \subseteq \mathit{PTIME}$ and $\mathcal{NRA} + \mathit{bfix}_p \subseteq \mathit{PSPACE}$. Conversely all queries in *PTIME* or *PSPACE* over ordered databases can be expressed in $\mathcal{NRA} + \mathit{bfix}_i$ or $\mathcal{NRA} + \mathit{bfix}_p$ respectively (Subsection 3.5). The first part also holds for external functions.
4. The class of queries in $\mathcal{NRA} + \mathit{bfix}_i$ and $\mathcal{NRA} + \mathit{bfix}_p$ whose inputs and outputs are flat relation types coincides with *FO+IFP* and *FO+PFP* (Section 4). This is the conservativity property of *bfix* and constitutes our main result. A similar property can be stated in the presence of external functions.

Notice that, when $t = \{s_1\} \times \ldots \times \{s_m\}$, then one can think of the fixpoint $\mathit{fix}(f)$ (and its variants) as being a simultaneous fixpoint of m functions, namely $\pi_1 \circ f, \ldots, \pi_m \circ f$. We are forced to consider simultaneous fixpoints, because during the translation of complex objects into flat relations in the proof of our conservativity result, single fixpoints are translated into simultaneous fixpoints. For the case of first order logic, Gurevich and Shelah [GS86] show that simultaneous fixpoints are no more expressive than single fixpoints ($m = 1$). We extend their proof to the case when external functions are in the language in Subsection 3.4.

Before proving these properties, we start by showing a number of examples of queries with fixpoints.

3.2 Examples

The first example suggests why the bounded fixpoint and the unbounded fixpoint have the same expressive power over flat relations. We give the general result in Proposition 3.6.

Example 3.2 *The transitive closure can be computed using the bounded fixpoint. Consider some input relation* $x \in \{s \times s\}$*, and let* $f : \{s \times s\} \to \{s \times s\}$ *and* $e : \{s \times s\}$ *be defined by:* $f = \lambda y.x \cup (x \circ y)$ *(where* $x \circ y$ *is* relation composition, *which can be easily defined in* $\mathcal{NRA}$*),* $e = (\Pi_1(x) \cup \Pi_2(x)) \bowtie (\Pi_1(x) \cup \Pi_2(x))$*. Both* f *and* e *have* x *as a free variable. Then the transitive closure is given by* $tc = \lambda x.bfix(f, e)(x)$*.*

The following two examples show that, even in the absence of external functions, the unbounded fixpoints are strictly more expressive than the bounded fixpoints, if we allow types of set height ≥ 2.

Example 3.3 *We present three ways to compute the powerset using the (unbounded) fixpoint. Consider some input* $x : \{s\}$*, and the following three functions* $f_i : \{\{s\}\} \to \{\{s\}\}$*,* $i = 1, 2, 3$*:* $f_1(Y) \stackrel{\text{def}}{=} \{\emptyset\} \cup map(\eta)(x) \cup map(\lambda(y_1, y_2).y_1 \cup y_2)(Y \bowtie Y)$*,* $f_2(Y) \stackrel{\text{def}}{=} \{\emptyset\} \cup map(\lambda(e, y).ins(e, y))(x \bowtie Y)$*, and* $f_3(Y) \stackrel{\text{def}}{=} \{x\} \cup map(\lambda(e, y).\ del(e, y))(x \bowtie Y)$ *(where* $ins, del : s \times \{s\} \to \{s\}$ *are:* $ins(e, x) \stackrel{\text{def}}{=} \{e\} \cup x$*,* $del(e, x) \stackrel{\text{def}}{=} x - \{e\}$*). Then* $\lambda x.fix(f_i)(x) : \{s\} \to \{\{s\}\}$ *computes the powerset, for all* $i = 1, 2, 3$*. A method similar to* $fix(f_1)$ *for computing powerset can be found in [GG92], while a method similar to* $fix(f_2)$ *can be found in [AB88].*

Example 3.4 *[GG91] We express the function parity* $: \{s\} \to \{unit\}$*, with* $parity(x) = true$ *iff* $card(x)$ *is even, by first computing, using one of the above methods, the set* $p : \{\{s\} \times \{unit\}\}$*, defined by* $p = \{(y, even(y)) \mid y \subseteq x\}$*. When* $s = \mathbb{D}$*, then parity is a function with flat input and output types, which can be computed using unbounded fixpoints, but which, by Corollary 4.2, cannot be computed using bounded fixpoints.*

In fact, the unbounded fixpoints can express exactly the elementary functions, when no external functions are in Σ. However, when Σ contains external functions, then the fixpoints can express even more than elementary functions, even in $\mathcal{NRA}_1(\Sigma) + fix$, as the next example proves. Moreover, this example also shows that fix_i, in conjunctions with certain external functions, can express *partial* queries.

Example 3.5 *Consider* $\mathbb{N}$ *(the set of natural numbers) as a base type, and suppose* Σ *contains the constant* $0 \in \mathbb{N}$ *and the successor function* $succ : \mathbb{N} \to \mathbb{N}$*. Then, for any recursive (i.e. Turing computable) function* $f : \mathbb{N}^k \to \mathbb{N}$*, the function* $\lambda x.\{f(x)\}$ *can be expressed in* $\mathcal{NRA}_1(\Sigma) + fix$*. This can be proven by structural induction. We only show the case in which* f *is obtained by minimization,* $f(x) = \mu y.p(x, y)$*, i.e.* $f(x)$ *is the smallest* y *such that* $p(x, y)$ *is true, where* $p : \mathbb{N} \times \mathbb{N} \to \{unit\}$ *is some predicate expressible in* $\mathcal{NRA}_1(\Sigma) + fix$*. Then, for some input* $x \in \mathbb{N}$*, let* $g : \{\mathbb{N}\} \to \{\mathbb{N}\}$*,* $g(Y) = if\ \exists y \in Y.p(x, y)\ then\ Y\ else\ \{0\} \cup map(succ)(Y)$*. Clearly,* g *can be expressed in* $\mathcal{NRA}_1(\Sigma) + fix$*. Then* $fix(g) = \{0, 1, \ldots, f(x)\}$*, or is undefined, when* $f(x)$ *is undefined (because, recall,* $\{\mathbb{N}\}$ *only contains the finite subsets of* $\mathbb{N}$*). Finally, it is easy to select the largest element of* $fix(g)$*, in order to obtain* $\{f(x)\}$*.*

3.3 Fixpoints and Bounded Fixpoints on Flat Relations

We show here that on flat relations the bounded fixpoint coincides with the (unbounded) fixpoint, provided that the language has no external functions.

Proposition 3.6 $\mathcal{NRA}_1 + fix_p = \mathcal{NRA}_1 + bfix_p$ *and* $\mathcal{NRA}_1 + fix_i = \mathcal{NRA}_1 + bfix_i$*.*

Proof. Consider some fixpoint in $\mathcal{NRA}_1$, $\mathit{fix}_p(f)$, where $f : s \times t \to t$. Recall that t is a PS-type, that is of the form $t = \{s_1\} \times \ldots \times \{s_m\}$, with $s_1, \ldots, s_n$ scalar types. To keep the notation simple, assume $n = 1$, and let $s_1 = \mathbb{D}_1 \times \ldots \times \mathbb{D}_m$ be a product of m base types. The idea is that we can express $\mathit{fix}_p(f)$ as a bounded fixpoint because we can compute a "bound" for it at type $\{s_1\}$, as the product of the active domain of the input. More precisely, let $g_i : s \to \{\mathbb{D}_i\}$ be the function collecting all values of type $\mathbb{D}_i$ mention in the input together with all constants of type $\mathbb{D}_i$ in Σ, $i = 1, m$. That is $g_i(x)$ is the "active domain" of x at base type $\mathbb{D}_i$. Then $\mathit{fix}_p(f) = \mathit{bfix}_p(f, (g_1 \bowtie \ldots \bowtie g_m))$. □

Examples 3.3 and 3.4 show that, at higher types, the unbounded fixpoint is strictly more expressive than the bounded one. Example 3.5 shows that it is strictly more expressive even at flat types, in the presence of certain external functions.

3.4 Fixpoints and Bounded Fixpoints on Complex Objects

Here we prove that the partial and inflationary fixpoints have the same expressive power in $\mathcal{NRA}(\Sigma)$, even in the presence of external functions. This happens because we can go through larger set heights: by contrast, for the case of *FO*, Abiteboul and Vianu [AV91b] show that the inflationary and partial fixpoint does not collapse, unless *PTIME* = *PSPACE*. As a side result we obtain that, by restricting the fixpoints to a single set (instead of allowing fixpoints simultaneously on m sets), we do not loose expressive power. This was known for fixpoints in first order logic [GS86], but the presence of external functions complicate the matter. Our side result provides us with a simple proof, if one is willing to increase the set height. We give a separate proof that the result still holds while keeping the same set height.

According to the definition, the fixpoints can be computed simultaneously on several sets. That is, the PS-type t could be a product of, say, m set types: $t = \{t_1\} \times \ldots \times \{t_m\}$, and $f : t \to t$. Then we say that the fixpoint $\mathit{fix}(f)$ is taken over m sets. Let fix^1_p, fix^1_i, bfix^1_p, bfix^1_i stand for the fixpoint constructions over only one set, i.e. in which m is restricted to 1. We shall prove simultaneously that fix and fix^1 have the same expressive power, and that fix_i and fix_p have the same expressive power, even in the presence of external functions. For the latter to hold, we need a notation for the undefined object in a language with fix_i, since fix_p can express partial queries. For each type s, let $\Omega_s : s$ be a constant name denoting the undefined expression, and $\Omega = \{\Omega_s \mid s \in \mathit{Types}\}$. All operations in $\mathcal{NRA}(\Sigma \cup \Omega)$ are assumed to be strict, e.g. $x^s \cup \Omega_s = \Omega_s$, with one exception: we consider a non-strict primitive *if-then-else* in the language. When defined as in Proposition 2.4, the conditional results as strict, that is ($\mathit{if\ true\ then}\ e_1\ \mathit{else}\ \Omega_s) = \Omega_s$. In $\mathcal{NRA}(\Sigma \cup \Omega)$ however, we assume it as a non-strict primitive, i.e. ($\mathit{if\ true\ then}\ e_1\ \mathit{else}\ \Omega_s) = e_1$. This is related to the *v-operator* of [GG92], used for a similar purpose.

Lemma 3.7 $\mathcal{NRA}_k(\Sigma) + \mathit{fix}^1_i \subseteq \mathcal{NRA}_k(\Sigma) + \mathit{fix}_p \subseteq \mathcal{NRA}_{k+1}(\Sigma \cup \Omega) + \mathit{fix}^1_i$.

Proof. The last inclusion is the only nontrivial one, so let $\mathit{fix}_p(f)$ be in $\mathcal{NRA}_k(\Sigma) + \mathit{fix}_p$, where $f : t \to t$. Define $g : \{t\} \to \{t\}$ to be: $g(Y) = \{\phi\} \cup \mathit{map}(f)(Y)$. The intuition behind g is that, if

$$Y = \{\emptyset, f(\emptyset), f^{(2)}(\emptyset), \ldots, f^{(n)}(\emptyset)\}$$

then

$$g(Y) = \{\emptyset, f(\emptyset), f^{(2)}(\emptyset), \ldots, f^{(n)}(\emptyset), f^{(n+1)}(\emptyset)\}$$

Let $e = \mathit{fix}_i^1(g)$. Then $\mathit{fix}_p(f) = \mathit{if}\ (\exists y \in e.(f(y) = y))\ \mathit{then}\ y\ \mathit{else}\ \Omega_t$. $\square$

This lemma implies immediately:

Proposition 3.8 *By extending $\mathcal{NRA}(\Sigma)$ with* any *of fix_i, fix_p, fix_i^1 or fix_p^1 we get, essentially, the same expressive power. Therefore, we shall abbreviate with $\mathcal{NRA}(\Sigma) + \mathit{fix}$ any of them.*

To be precise, $\mathcal{NRA}(\Sigma) + \mathit{fix}_p$ can indeed express undefined queries, which $\mathcal{NRA}(\Sigma) + \mathit{fix}_i$ sometimes cannot (it depends on Σ), so we needed to add a notation for the undefined term to make up the difference. There are other ways to state their relationships without referring to Ω. For example, from the proof of Lemma 3.7 we derive also that all total functions expressible in $\mathcal{NRA}(\Sigma) + \mathit{fix}_p$ are expressible in $\mathcal{NRA}(\Sigma) + \mathit{fix}_i^1$ as well, i.e. without the need for the terms Ω_t. Alternatively, let us extend the definition of the partial fixpoint to a "total" fixpoint, such that $\mathit{fix}_t(f)(x) \stackrel{\text{def}}{=} y$ when $\mathit{fix}_p(f)(x)$ is defined and equal to y, and $\mathit{fix}_t(f)(x) \stackrel{\text{def}}{=} \emptyset$ otherwise. Then we get that $\mathcal{NRA}(\Sigma) + \mathit{fix}_t = \mathcal{NRA}(\Sigma) + \mathit{fix}_i$.

In particular, it follows from the proof of Lemma 3.7 that $\mathcal{NRA}_k(\Sigma) + \mathit{fix}_p \subseteq \mathcal{NRA}_{k+1}(\Sigma) + \mathit{fix}_p^1$. Thus fix and fix^1 are equivalent over $\mathcal{NRA}$, in all possible interpretations (inflationary/partial, bounded/unbounded). We conclude this subsection by strengthening this result to the case of an imposed set height.

Proposition 3.9 *For any $k \geq 0$, the following hold:*

1. $\mathcal{NRA}_k(\Sigma) + \mathit{fix}_p^1 = \mathcal{NRA}_k(\Sigma) + \mathit{fix}_p$
2. $\mathcal{NRA}_k(\Sigma) + \mathit{fix}_i^1 = \mathcal{NRA}_k(\Sigma) + \mathit{fix}_i$
3. $\mathcal{NRA}_k(\Sigma) + \mathit{bfix}_p^1 = \mathcal{NRA}_k(\Sigma) + \mathit{bfix}_p$
4. $\mathcal{NRA}_k(\Sigma) + \mathit{bfix}_i^1 = \mathcal{NRA}_k(\Sigma) + \mathit{bfix}_i$

Proof. (Sketch) We give the proof for the most general case, the (unbounded) partial fixpoint: the others follow immediately. The idea is to encode some m-tuple $(y_1, \ldots, y_m) \in t$, where $t = \{t_1\} \times \ldots \times \{t_m\}$, by some value $u \in t'$, where $t' = \{(t_1 \times (s \times s)) \times \ldots \times (t_m \times (s \times s))\}$; the type s will be chosen later, but it will be subject to the restriction $sh(s) \leq \max(sh(t_1), \ldots, sh(t_m))$. To encode some m-tuple, we need a set $w \in \{s\}$, containing at least 2 elements. Then the type $\{s \times s\}$ will play the role of booleans, by defining $\mathit{true} = \{(\alpha, \beta) \mid \alpha, \beta \in w, \alpha \neq \beta\}$, and $\mathit{false} = \{(\alpha, \alpha) \mid \alpha \in w\}$. To illustrate the encoding, assume $m = 3$. Then, consider any three *nonempty* sets $z_i \in \{t_i\}$, and take $u = ((y_1 \bowtie \mathit{true}) \bowtie (z_2 \bowtie \mathit{false}) \bowtie (z_3 \bowtie \mathit{false})) \cup ((z_1 \bowtie \mathit{false}) \bowtie (y_2 \bowtie \mathit{true}) \bowtie (z_3 \bowtie \mathit{false})) \cup ((z_1 \bowtie \mathit{false}) \bowtie (z_2 \bowtie \mathit{false}) \bowtie (y_3 \bowtie \mathit{true}))$. The decode function $\mathit{decode} : t' \to t$ is straightforwardly expressible in $\mathcal{NRA}$, while the encode function $\mathit{encode} : t \to t'$ will depend on the nonempty sets w, $z_1, \ldots, z_m$, with $\mathit{card}(w) \geq 2$. Then $f : t \to t$ can be simulated by some $g : t' \to t'$, where $g = \mathit{encode} \circ f \circ \mathit{decode}$, namely $\mathit{fix}_p(f) = \mathit{decode} \circ (\mathit{fix}_p^1(g)) \circ \mathit{encode}$.

The problem is that it is not obvious how to construct the m nonempty sets $z_1, \ldots, z_m$, nor how to choose the type s of the proper set height and to find at least two different elements in s, in order to construct the set w. The presence of external functions make things more complicated, as seen in the following example. Consider two base types, $\mathbb{N}$ and *string*, and two external functions $p : \mathbb{N} \to \mathbb{N}$, $p(n) = \mathit{if}\ \mathit{odd}(n)\ \mathit{then}\ \frac{3n+1}{2}\ \mathit{else}\ \frac{n}{2}$, and

$g : \mathbb{N} \to \{string\}$, $g(n) = \mathit{if}\ n = 100\ \mathit{then}\ \{\text{“Hello”}\}\ \mathit{else}\ \emptyset$. Note that “Hello” is *not* itself a constant in Σ. Let $x : \{\mathbb{N}\}$, and consider $f : \{\mathbb{N}\} \times \{string\} \to \{\mathbb{N}\} \times \{string\}$, $f(y_1, y_2) = (x \cup y_1 \cup map(p)(y_1), y_2 \cup ext(q)(y_1))$. Then f may need arbitrarily many iterations before generating a string, or may reach a fixpoint before generating any string. So in this case, it is impossible to construct some nonempty set $z_2 \in \{string\}$ from the beginning.

First we show how to choose s. Consider the following $2^m + 1$ m-tuples of type $\{t_1\} \times \ldots \times \{t_m\}$ (we abbreviate $\emptyset$ for the m-tuple $(\emptyset, \ldots, \emptyset)$):

$$\emptyset, f(\emptyset), f^{(2)}(\emptyset), f^{(3)}(\emptyset), \ldots, f^{2^m}(\emptyset)$$

We abbreviate $(y_1, \ldots, y_m)$ each of the $2^m + 1$ m-tuple. Then, one of the following must hold: (1) either two of these $2^m + 1$ m-tuples are equal, in which case the fixpoint of f is easily extracted, or (2) for some i, the union of all y_i's has at least two elements. It is easy to check in $\mathcal{NRA}$ which of these cases holds. In the second case, choose $s = t_i$, and take w to be the union of all y_i's.

Finally, we show how to choose the sets $z_1, \ldots, z_m$. For each $i = 1, m$, let k_i be the first stage at which $y_i \neq \emptyset$ in the sequence $\emptyset, f(\emptyset), f^{(2)}(\emptyset), \ldots$. We write $k_i = \infty$ if y_i remains always empty. Note that after iteration k_i, y_i may become empty again. First assume $k_1 < k_2 < \ldots < k_m$, i.e. y_1 becomes nonempty first, next y_2, etc. Then $\mathit{fix}_p(f)$ will be simulated by a series of $\mathit{fix}_p^1(g_i)$, $i = 0, \ldots, m$, where $g_i : \{(t_1 \times (s \times s)) \times \ldots \times (t_i \times (s \times s))\} \to \{(t_1 \times (s \times s)) \times \ldots \times (t_i \times (s \times s))\}$ only has access to i nonempty sets, $z_1, \ldots, z_i$. Essentially g_i decodes its input to obtain i sets $y_1, \ldots, y_i$, assigns $y_{i+1} = \emptyset, \ldots, y_m = \emptyset$, and computes $f(y_1, \ldots, y_m)$. If the result still has $y_{i+1}, \ldots, y_m$ empty, then g_i returns the encoding of the components $y_1, \ldots, y_i$ of this result. Else, it returns its input unchanged. The idea is that $\mathit{fix}_p(g_i)$ computes $\mathit{fix}_p(f)$ from stage k_i to stage $k_{i+1} - 1$: here, $\mathit{fix}_p(g_{i+1})$ takes over. In fact, the functions g_i have to be more complicated (even their types have to be changed), because in computing $\mathit{fix}_p(g_i)$, we do not want to start with an i-tuple of empty sets $(\emptyset, \ldots, \emptyset)$, but with the result of $\mathit{fix}_p(g_{i-1})$, on which we apply f once (to go from stage $k_i - 1$ to stage k_i, and to construct in the process the nonempty set z_i). Thus $\mathit{fix}_p(g_{i-1})$ will occur textually in the body of g_i. We omit the details.

The above simulation of $\mathit{fix}_p(f)$ in $\mathcal{NRA} + \mathit{fix}_p^1$ works only when $k_1 < k_2 < \ldots < k_m$. In a similar way we can also check, in $\mathcal{NRA} + \mathit{fix}_p^1$, whether indeed $k_1 < \ldots < k_m$ holds. Now we write in $\mathcal{NRA} + \mathit{fix}_p^1$ some expression, which makes all possible simulations of $\mathit{fix}_p(f)$, corresponding to all possible order relations of $k_1, \ldots, k_m$ (it is obvious how to extend the simulation for an order relation in which some of the $<$ are replaced by $\leq$). Finally, we only retain the result corresponding to the correct order. □

3.5 Fixpoints and Complexity Classes

For the bounded fixpoints, we have immediately:

Proposition 3.10 *$\mathcal{NRA}(\Sigma) + \mathit{bfix}_p \subseteq$ PSPACE and $\mathcal{NRA}(\Sigma) + \mathit{bfix}_i \subseteq$ PTIME, provided that all functions in Σ are in PSPACE or PTIME, respectively.*

Proof. (Sketch) Buneman et al. prove in [BNTW95] that every query f in $\mathcal{NRA}$ is computable in *PTIME*, by induction on the structure of f. To show that $\mathcal{NRA}(\Sigma) + \mathit{bfix}_i \subseteq$ *PTIME*, we simply extend their induction to our two new cases: (1) when the query is an external function, $p : d_p \to c_p$, and (2) when the query is $\mathit{bfix}_i(f, g)$. The first case is handled by our assumption (p is in *PTIME*). For the second case, we apply induction hypothesis to g first: it follows that, on input x, $g(x)$ can be computed in *PTIME*, and has a size which is polynomially bounded in x. Hence, to compute the fixpoint of $h(y) \stackrel{\text{def}}{=} f(x, y) \cap g(x)$ we only need to do polynomially many iterations of h, and each can be computed in *PTIME*, by induction hypothesis on f. □

Theorem 3.11 *$\mathcal{NRA} + \mathit{bfix}_p + \mathit{order} = \mathit{PSPACE}$ and $\mathcal{NRA} + \mathit{bfix}_i + \mathit{order} = \mathit{PTIME}$.*

Proof. (Sketch) The proof is an adaptation of previously known techniques [Imm86, GV91, ST]. Let Q be a generic, *PTIME* computable query. Let T be the *PTIME* Turing machine computing Q, assuming the encoding of complex objects in [ST]. Let P be a polynomial, which binds the number of steps of T: that is T computes $Q(x)$ in at most $P(\mathit{size}(x))$ steps, for every input x. We have to simulate T in $\mathcal{NRA} + \mathit{bfix}_i + \mathit{order}$, and for this, we start by constructing, from x, a set z large enough to encode all time stamps up to $P(\mathit{size}(x))$: that is z has cardinality $\geq P(\mathit{size}(x))$, and is totally ordered. The order on z is obtained by lifting the order relation on the base types. The type of z depends on that of x, and on the polynomial P, and z can be computed in $\mathcal{NRA}$ from x, see [ST]. Next, we need two ordered sets u, v, which do not depend on x, and which we assimilate with $u = \{1, 2, \ldots, p\}$ and $v = \{0, 1, \ldots, q\}$, where p is the number of letters in T's tape alphabet, while q is the number of states in T. We describe configurations of T as sets of quadruples (t, c, a, s) with $t, c \in z$, $a \in u$, and $s \in v$, meaning that "at step t the tape cell c contains the letter a". If in addition $s > 0$, then "at step t, T's head is on cell c, and T is in state s". Of course, we rely on the observation that at most $P(\mathit{size}(x))$ cells may be written during T's computation. Next we observe that one can compute the set of configurations incrementally, for $t = 0, 1, 2, \ldots$, as a fixpoint in $\mathcal{NRA} + \mathit{fix}_i + \mathit{order}$. Indeed, the initial configuration (for $t = 0$) is obtained from an encoding of x, and it is easy to describe in $\mathcal{NRA} + \mathit{bfix}_i$ a successor function f, s.t. if y is some set $y \subseteq z \bowtie z \bowtie u \bowtie v$ containing all configurations with time stamps $\leq t$, then $f(y)$ returns the set of all configurations with time stamps $\leq t + 1$. Here bfix_i is needed to compute addition on the ordered set z. Finally we simulate the computation of T by taking the fixpoint of f. So far, all we have described is standard. The only novelty is that we have to replace the last fixpoint with a bounded fixpoint. This is easily done, because $e \stackrel{\text{def}}{=} z \bowtie z \bowtie u \bowtie v$ is a bound for f, i.e. $\mathit{fix}_i(f) = \mathit{bfix}_i(f, e)$. This proves that $\mathit{PTIME} \subseteq \mathcal{NRA} + \mathit{bfix}_i + \mathit{order}$. The inclusion $\mathit{PSPACE} \subseteq \mathcal{NRA} + \mathit{bfix}_p + \mathit{order}$ is proven similarly. □

Notice that we only state and prove this theorem in absence of external functions. Currently there is no commonly agreed upon definition of *PTIME* (or *PSPACE*) queries with external functions: see [Suc] for a discussion of computability of queries with external functions.

We conclude this subsection with a discussion on the complexity of the (unbounded) fixpoint. Consider a new primitive $\mathit{powerset} : \{t\} \to \{\{t\}\}$, for each type t, and let $\mathcal{NRA}(\Sigma) + \mathit{powerset}$ denote $\mathcal{NRA}(\Sigma)$ extended with *powerset*. It is known [GG92] that, without external functions, $\mathcal{NRA} + \mathit{fix} = \mathcal{NRA} + \mathit{powerset}$ (Example 3.3 shows one inclusion). Hull and Su [HS91] show that the latter language can express all *elementary recursive* queries. More precisely, let $\mathit{exp}(0, n) = n$, $\mathit{exp}(k+1, n) = 2^{\mathit{exp}(k,n)}$: then the elementary recursive queries are defined to be those whose time (or, equivalently, space) complexity is in $\bigcup_{k \geq 0, c \geq 0} O(\mathit{exp}(k, n^c))$. Grumbach and Vianu refine this result for the case of fixpoints, showing a correspondence between the number k in $\mathit{exp}(k, n^c)$ and the set height: for $k \geq 1$, $\mathcal{NRA}_{k+1} + \mathit{fix}_i$ coincides with all queries whose time complexity is $O(\mathit{exp}(k, n^c))$, for some $c \geq 0$, and $\mathcal{NRA}_{k+1} + \mathit{fix}_p$ coincides with all queries whose space complexity is

$O(exp(k, n^c))$, for some $c \geq 0$. Finally, as our Example 3.5 shows, this picture becomes more complex, in the presence of external functions. Namely $\mathcal{NRA}(\Sigma) + \mathit{fix} \supset \mathcal{NRA}(\Sigma) + \mathit{powerset}$, because the former may express more than just elementary queries.

4 The Conservative Extension Theorem for Bounded Fixpoints

The main result of this paper is the following theorem (an extension of Theorem 2.9), and in its corollary, establishing that bounded fixpoints over complex objects are conservative extensions of fixpoints over flat relations.

Theorem 4.1 (Conservative Extension Theorem) *If all function symbols in Σ have set height ≤ 1, then $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ is a conservative extension of $\mathcal{NRA}_1(\Sigma) + \mathit{bfix}$, for both inflationary and partial fixpoint semantics: that is, a query $f : s \to t$ where s, t have set height ≤ 1 is in $\mathcal{NRA}_1(\Sigma) + \mathit{bfix}$ if and only if it is in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$. Moreover, $\mathcal{NRA}_1(\Sigma) + \mathit{bfix}$ is a conservative extension of the relational algebra with bounded fixpoints, $\mathcal{RA}(\Sigma) + \mathit{bfix}$.*

The proof is given in Section 5.

When Σ has no external functions (but it may have constants), then the bounded fixpoints and the unbounded fixpoint coincide on flat relations, Proposition 3.6. Hence we get:

Corollary 4.2

1. *$\mathcal{NRA} + \mathit{bfix}_p$ is a conservative extension of $\mathcal{RA} + \mathit{fix}_p$, which is equivalent to FO+PFP, and also equivalent to the while - queries [CH82], and to DATALOG$^{*\neg}$.*
2. *$\mathcal{NRA} + \mathit{bfix}_i$ is a conservative extension of $\mathcal{RA} + \mathit{fix}_i$. The latter is equivalent to FO+IFP, to FO+LFP, and to DATALOG$^{\neg}$.*

As a negative consequence, $\mathcal{NRA} + \mathit{bfix}_i$ and $\mathcal{NRA} + \mathit{bfix}_p$ are strictly included in *PTIME* and *PSPACE* respectively: the function $\mathit{parity} : \{s\} \to \{\mathit{unit}\}$ (see Example 3.4), is not expressible with bounded fixpoints. On the other hand, no database query language capable of capturing *exactly* the *PTIME* queries is currently known, so bfix_i and bfix_p seem to offer a reasonable extension to higher types of the fixpoints in the relational algebra (or, equivalently, first-order logic).

The technique used in the proof of Theorem 4.1 is of independent interest. It consists of two steps: first queries in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ are *flattened*, that is expressed in some extension of $\mathcal{RA}(\Sigma) + \mathit{bfix}$ with "indexes". In the second step the indexes are eliminated. Moreover, the first step also works for the (unbounded) fixpoint:

Fact 4.3 *Every query in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ can be encoded by some query in $\mathcal{RA}(\Sigma) + \mathit{bfix}$. Every query in $\mathcal{NRA}(\Sigma) + \mathit{fix}$ can be encoded by some query in an extension of $\mathcal{RA}(\Sigma) + \mathit{fix}$ with "indexes".*

From that we derive a simple implementation method of complex objects databases in terms of relational databases. First, the complex objects database is encoded as a relational database with indexes. Next, any query in, say, $\mathcal{NRA}$, with no restriction on its input or output types, is translated into a relational query (i.e. in $\mathcal{RA}$), and executed on the encoding of the original database: the result is an encoding (with indexes) of a complex object, and can be easily decoded. The technique extends also to queries with bounded fixpoints: a query in $\mathcal{NRA} + \mathit{bfix}$ will be translated into one in $\mathcal{RA} + \mathit{fix}$. It also works for queries with unbounded fixpoints, but here the encoded query may use "indexes": that is, a query in $\mathcal{NRA} + \mathit{fix}$ is translated into one in $\mathcal{RA} + \mathit{fix}$ "with indexes". We give more details in Subsection 5.2.

5 Proof of the Conservativity Extension Theorem

In this section, we present the technique used for proving the conservativity result (Theorem 4.1). It consists of two steps. In the first step we encode $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ into the relational algebra over the same signature Σ plus indexes: $\mathcal{RA}(\Sigma \cup I) + \mathit{bfix}$. In the second step we get rid of the indexes.

For more clarity, we shall describe this technique for the case when all function symbols in Σ have set height 0. That is, Σ may contain functions like $\mathit{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ or $\mathit{zero} : \mathit{unit} \to \mathbb{N}$, but not functions like $\mathit{card} : \{\mathbb{D}\} \to \mathbb{N}$ or $\mathit{sum} : \{\mathbb{N}\} \to \mathbb{N}$. Finally, in Subsection 5.4 we show how to extend these techniques to the case when Σ contains functions of set height ≤ 1.

We define the **the relational algebra** $\mathcal{RA}(\Sigma)$ to be a minor extension of the traditional relational algebra [Ull89]: its types are the **flat relation types** of Section 2, which can be described by the grammar $t ::= \mathit{unit} \mid \{s\} \mid t \times t$, where s is a scalar type. So the flat relation types are essentially of the form $\{s_1\} \times \ldots \times \{s_m\}$, with $s_1, \ldots, s_m$ scalar types: when $m = 0$ we get unit. For example $\{\mathbb{D} \times \mathbb{D}\} \times \{\mathbb{D}\}$ is a flat relation type, but $\mathbb{D} \times \{\mathbb{D}\}$ is not. Throughout this section we will denote scalar types with $s, s', s_1, \ldots$, and all other types with $t, t', t_1, \ldots$. The operations of $\mathcal{RA}(\Sigma)$ are the following:

1. Set union $\cup : \{s\} \times \{s\} \to \{s\}$.
2. Set difference $- : \{s\} \times \{s\} \to \{s\}$.
3. Cartesian product $\bowtie$: $\{s\} \times \{s'\} \to \{s \times s'\}$.
4. All functions of the form $\mathit{map}(\varphi) : \{s\} \to \{s'\}$ with φ in $\mathcal{NRA}_0(\Sigma)$. These include: (1) database projections $\Pi_i : \{s_1 \times s_2\} \to \{s_i\}$, $\Pi_i = \mathit{map}(\pi_i)$, and (2) arithmetic operations applied componentwise, e.g. $f \stackrel{\text{def}}{=} \mathit{map}((\mathit{add}, (\pi_1, \pi_2)))$, with type $f : \{\mathbb{N} \times \mathbb{N}\} \to \{\mathbb{N} \times (\mathbb{N} \times \mathbb{N})\}$, which adds two columns into a third.
5. All selection functions $\sigma_p : \{s\} \to \{s\}$, where $p : s \to \{\mathit{unit}\}$ is in $\mathcal{NRA}_0(\Sigma)$. Recall that σ_p was expressible in $\mathcal{NRA}(\Sigma)$, see Example 2.1. For example the selection $\sigma_p : \{(\mathbb{D} \times \mathbb{D}) \times s\} \to \{(\mathbb{D} \times \mathbb{D}) \times s\}$ associated to $p : (\mathbb{D} \times \mathbb{D}) \times s \to \{\mathit{unit}\}$ $p \stackrel{\text{def}}{=} \mathit{eq} \circ \pi_1$ selects exactly those tuples whose values on the first and second column are equal, traditionally written $\sigma_{1=2}$.
6. $\emptyset : \mathit{unit} \to \{s\}$, $\mathit{null} : \mathit{unit} \to \{\mathit{unit}\}$, and $\iota_{\{s\}} : \{s\} \to \mathit{unit}$. Here null has the meaning $\mathit{null}(()) = \{()\}$.

7. The identity $id_t : t \to t$, composition of functions $g \circ f : t \to t''$ (for $f : t \to t'$, $g : t' \to t''$), projections $\pi_i : t_1 \times t_2 \to t_i$ $(i = 1, 2)$, and pairing of functions $(f_1, f_2) : t \to t_1 \times t_2$, (for $f_i : t \to t_i$).

Also, we define $\mathcal{RA}(\Sigma) + \mathit{fix}$ and $\mathcal{RA}(\Sigma) + \mathit{bfix}$ to be the extensions obtained from $\mathcal{RA}(\Sigma)$ by adding *fix* or *bfix* respectively.

Most operators correspond to standard relational algebra ones, except that we favor a presentation which will make it easier for us to translate $\mathcal{NRA}$ into $\mathcal{RA}$. The only exception is *null*, which is the only non-strict operator: it always returns $\{()\}$. Formally, let us call a query f of type $\{s_1\} \times \ldots \times \{s_m\} \to \{s\}$ **strict** if $f(\emptyset, \ldots, \emptyset) = \emptyset$. Then *null* is non-strict, and it is easy to prove that every query in $\mathcal{RA}(\Sigma)$ (extended with *bfix* or *fix* respectively) expressed without *null* is strict: we call this sub-language the **strict fragment** of $\mathcal{RA}(\Sigma)$ (+*bfix*, or +*fix*). For an example of a non-strict query, suppose $c : s$ is a constant in Σ, and consider the query $f : \{s_1\} \to \{s\}$, such that $f(x) \stackrel{\text{def}}{=} \{c\}$ for every x: we expressed f as $f = \mathit{map}(c) \circ \mathit{null} \circ !_{\{s_1\}}$. In some sense *null* is almost superfluous, namely for any non-strict query f in $\mathcal{RA}(\Sigma)$, if we define $f'(\emptyset, \ldots, \emptyset) \stackrel{\text{def}}{=} \emptyset$ and $f'(x_1, \ldots, x_m) \stackrel{\text{def}}{=} f(x_1, \ldots, x_m)$ for $(x_1, \ldots, x_m) \neq (\emptyset, \ldots, \emptyset)$, then f' can be expressed in $\mathcal{RA}(\Sigma)$ without *null*. We found it more convenient for the proof of the conservativity property to keep *null* in the language, instead of reasoning on a by-case basis. Finally, note that we can express $\mathit{not} : \{\mathit{unit}\} \to \{\mathit{unit}\}$ (see Subsection 2.1) as $\mathit{not}(x) = \mathit{null} - x$. Formally: $\mathit{not} = - \circ (\mathit{null} \circ \iota_{\{\mathit{unit}\}}, id_{\{\mathit{unit}\}})$.

We introduce now the indexes needed to encode complex objects in terms of flat relations.

Definition 5.1 *An* **index** *set is a base type I together with an injective function $\mathit{pair} : I \times I \to I$ (so I is infinite), and two distinct constants $\mathit{left}, \mathit{right} : \mathit{unit} \to I$. We write $\Sigma \cup I$ for the signature Σ extended with I, left, right and pair.*

Our plan for proving the conservativity theorem is as follows. In step 1 we describe the translation of $\mathcal{NRA}(\Sigma)$ (+*fix* or +*bfix*) into $\mathcal{RA}(\Sigma \cup I)$ (+*fix* or +*bfix*). Step 1 consists of three parts:

1. A translation of types: each type t in $\mathcal{NRA}(\Sigma)$ is transformed into some type $\mathit{FLAT}(t)$ in $\mathcal{RA}(\Sigma \cup I)$, Subsection 5.1.

2. For each $\mathcal{NRA}(\Sigma)$ type t, an encoding relation $\sim_t \subseteq t \times \mathit{FLAT}(t)$: when $x \sim_t r$ holds, then, intuitively, r is an encoding of x. That is, we encode complex objects, x, as tuples of flat relations, r. The relation $\sim_t$ is one-to-many: some complex object x may be encoded by several r's. Also, not every r is the encoding of some complex object x; but when it is, then it is the encoding of at most one complex object. The decoding function, $\mathit{decode} : \mathit{FLAT}(t) \to t$, is expressible in $\mathcal{NRA}$, and even in $\mathcal{RA}$, when t is a flat relation type. We describe this part formally in Subsection 5.1.

3. A translation of functions: each function $f : t \to t'$ in $\mathcal{NRA}(\Sigma)$ (+*bfix*, or +*fix*), is mapped into some function $\mathit{FLAT}(f) : \mathit{FLAT}(t) \to \mathit{FLAT}(t')$, such that a *soundness* property holds: $x \sim_t r \Longrightarrow f(x) \sim_{t'} \mathit{FLAT}(f)(r)$. We describe this in Subsection 5.2.

Step 2 consists of "eliminating indexes". Formally we show that every query in $\mathcal{RA}(\Sigma \cup I) + \mathit{bfix}$, which is obtained as result of a translation as in item 3 above, can be rewritten into an equivalent query which does not use the functions *pair*, *left*, *right*. We describe this step in Subsection 5.3.

Finally, we comment on some important difference between the two query rewritings in the two steps. In step 1, the rewriting $f \rightsquigarrow FLAT(f)$ is uniform on types: given $f : t \to t'$ with t, t' complex objects types, the flat relation types of f's translation, $FLAT(t)$ and $FLAT(t')$, are independent on f. Moreover, the translation of a composed query, $f = f'' \circ f'$, can be done by translating the two sub-queries separately: $FLAT(f) = FLAT(f'') \circ FLAT(f')$. This step works for unbounded fixpoints too, which gives us a translation from $\mathcal{NRA}(\Sigma) + \mathit{fix}$ into $\mathcal{RA}(\Sigma \cup I) + \mathit{fix}$. In step 2, we start with a query $g : t \to t'$ with indexes, corresponding to an encoding, and rewrite it into some alternative encoding, without indexes. Here the type of the resulting query depends not only on t and t', but on g itself. Moreover, the translation of a composed query, $g = g'' \circ g'$, proceeds from its inputs to its outputs: we translate g' first and use results from this translation when translating g''. This step only works for the bounded fixpoint, and fails for the unbounded one.

5.1 The Translation of Types and the Encoding Relation

The central part in the flattening of types is the flattening of some set type, i.e. $FLAT(\{t\})$. For illustration, assume we already constructed $FLAT(t)$, which is a flat relation type, say $FLAT(t) = \{s\}$ (in general it will be a product of several set types). Then $FLAT(\{t\})$ should be some flat relation type capable of encoding elements of type $\{\{s\}\}$. A good candidate is $\{I \times s\}$: a set $\{r_1, \ldots, r_n\}$ where $r_1, \ldots, r_n \in \{s\}$ can be encoded as $(\{i_1\} \bowtie r_1) \cup \ldots \cup (\{i_n\} \bowtie r_n)$, where $i_1, \ldots, i_n$ are n distinct indexes from I. To be more precise, the type $\{I \times s\}$ encodes exactly those functions $\rho : I \to \{s\}$ that have **finite support**, i.e. in which $\rho(i) \neq \emptyset$ only holds for finitely many $i \in I$: therefore we denote $\{I \times s\}$ with $[I \to \{s\}]$. In general, we define a function $\rho : I \to \{s_1\} \times \ldots \times \{s_m\}$ to be with finite support, if the set $\{i \mid i \in I, \exists k = 1, \ldots m, \pi_k(\rho(i)) \neq \emptyset\}$ is finite. Returning to our example, define a binary *encoding* relation $\simeq \subseteq \{\{s\}\} \times [I \to \{s\}]$ by:

$$\{r_1, \ldots, r_n\} \simeq \rho \quad \text{iff} \quad \{r_1, \ldots, r_n\} = CoDom(\rho) \tag{1}$$

Note that we use the same letter ρ both for the function $\rho : I \to \{s\}$ and for its encoding $\rho \in [I \to s]$. However, there is a problem: it will always be the case that $\emptyset \in CoDom(\rho)$, hence whenever $\{r_1, \ldots, r_n\} \simeq \rho$, it will be the case that $\emptyset \in \{r_1, \ldots, r_n\}$. Alternatively, we could change Equation 1 to $\{r_1, \ldots, r_n\} = CoDom(\rho) - \{\emptyset\}$, but then we cannot encode sets containing the empty set. In short, this encoding cannot distinguish between $\{r_1, r_2\}$ and $\{r_1, r_2, \emptyset\}$. For this reason, we go from functions with finite support $\rho : I \to \{s\}$ to **partial, finite functions** (i.e. with finite domain), in notation $\varrho : I \Rightarrow \{s\}$. Any function ρ with finite support can be viewed as a partial function with finite domain ϱ, by defining $\varrho(i) = \rho(i)$ when $\rho(i) \neq \emptyset$, and $\varrho(i) =$ undefined otherwise. In order to encode functions ϱ which have $\emptyset$ in their codomain, we explicitly keep their domain, namely we encode a partial, finite function ϱ by a pair (δ, ρ) of type $\{I\} \times \{I \times s\}$, satisfying the constraint:

$$\Pi_1(\rho) \quad \subseteq \quad \delta \tag{2}$$

So now we encode some set $\{r_1, \ldots, r_n\}$ by a pair $\varrho = (\delta, \rho)$ in which, as before, $\rho = (\{i_1\} \bowtie r_1) \cup \ldots \cup (\{i_n\} \bowtie r_n)$, and $\delta = \{i_1, \ldots i_n\}$. Note that when, say, $r_1 = \emptyset$, then i_1 does not occur in ρ, but it will occur in δ. As a function with finite domain, we denote ϱ by $\varrho = \{i_1 \mapsto r_1, \ldots, i_n \mapsto r_n\}$. We emphasize that $r_1, \ldots, r_n$ are not necessarily distinct, i.e. that several indexes may be used to encode the same element. This assumption is necessary in order for our translation of queries to work, because, in general, a query may introduce several indexes for the same element: we cannot choose a unique index in $\mathcal{RA}$, because this is not a generic query.

We denote the type $\{I\} \times \{I \times s\}$ with $[I \Rightarrow \{s\}]$. As before, we use the same letter ϱ to denote both the function $\varrho : I \Rightarrow \{s\}$ and its encoding $\varrho = (\delta, \rho) \in [I \Rightarrow \{s\}]$ satisfying the constraint 2. We define the *encoding* relation $\cong \in \{\{s\}\} \times [I \Rightarrow s]$ by:

$$\{r_1, \ldots, r_n\} \cong \varrho \quad \text{iff} \quad CoDom(\varrho) = \{r_1, \ldots, r_n\} \tag{3}$$

Formally, we define for some flat relation type t the flat relation type $[I \to t]$ by induction on t: $[I \to unit] \stackrel{\text{def}}{=} unit$, $[I \to \{s\}] \stackrel{\text{def}}{=} [I \times s]$, and $[I \to (t \times t')] \stackrel{\text{def}}{=} [I \to t] \times [I \to t']$. Essentially we have $[I \to (\{s_1\} \times \ldots \times \{s_m\})] = \{I \times s_1\} \times \ldots \times \{I \times s_m\}$. Also, we define $[I \Rightarrow t] \stackrel{\text{def}}{=} \{I\} \times [I \to t]$, and define the relations $\simeq$ and $\cong$ by Equations 1 and 3.

Example 5.2 *Consider* $r_1 = \{a, b\}, r_2 = \emptyset, r_3 = \{a, c\}$ *of type* $\{\mathbb{D}\}$*. Then the set* $r = \{r_1, r_2, r_3\}$ *could be encoded by the partial, finite function* $\varrho = \{i_1 \mapsto r_1, i_2 \mapsto r_2, i_3 \mapsto r_3, i_4 \mapsto r_1\}$*. Viewed as a complex object,* $\varrho = (\{i_1, i_2, i_3, i_4\}, \{(i_1, a), (i_1, b), (i_4, a), (i_4, b), (i_3, a), (i_3, c)\})$*, and its type is* $[I \Rightarrow \{\mathbb{D}\}]$*, which is just* $\{I\} \times \{I \times \mathbb{D}\}$*. Thus,* $r \cong \varrho$*. Certainly a more economical encoding of* r *would only use three indexes instead of four, but having a minim number of indexes is not imposed by the encoding.*

Example 5.3 *Let* $r_1 = (\{a, b\}, \emptyset), r_2 = (\emptyset, \{c\}), r_3 = (\emptyset, \emptyset), r_4 = (\{a, c\}, \{b\})$ *be of type* $t = \{\mathbb{D}\} \times \{\mathbb{D}\}$*, and* $r = \{r_1, r_2, r_3, r_4\}$*. Then,* $[I \Rightarrow t] = \{I\} \times \{I \times \mathbb{D}\} \times \{I \times \mathbb{D}\}$*, and a possible encoding for* r *is* $\varrho = \{i_1 \mapsto r_1, i_2 \mapsto r_2, i_3 \mapsto r_3\}$*, i.e.* $\varrho = (\{i_1, i_2, i_3, i_4\}, \{(i_1, a), (i_1, b), (i_4, a), (i_4, c)\}, \{(i_2, c), (i_4, b)\})$*.*

Recall that the strict fragment of $\mathcal{RA}(\Sigma)$ (+*bfix*, or +*fix*) consists of all queries expressed without *null*.

Lemma 5.4 (The Strict Map Lemma) *Let* $f : t \to t'$ *be some function in the strict fragment of* $\mathcal{RA}(\Sigma \cup I)$ *(+bfix, or +fix). Then there is some function* $\overline{map}(f) : [I \to t] \to [I \to t']$ *in the strict fragment of* $\mathcal{RA}(\Sigma \cup I)$ *(+bfix, or +fix) which "simulates"* $map(f)$*, i.e. for all* $\rho \in [I \to t]$*, if* $\rho' = \overline{map}(f)(\rho)$*, then for all* $i \in I$*,* $f(\rho(i)) = \rho'(i)$*. In other words, the following holds:* $\{r_1, \ldots, r_n\} \simeq \rho \Longrightarrow \{f(r_1), \ldots, f(r_n)\} \simeq \overline{map}(f)(\rho)$*, and, more, the same indexes are used to encode* r_α *and* $f(r_\alpha)$*, for all* $\alpha = 1, \ldots, n$*.*

Proof. We define $\overline{map}(f)$ by induction on the structure of f, following the definition of $\mathcal{RA}$ at the beginning of Section 5.

1. $\overline{map}(\cup) \stackrel{\text{def}}{=} \cup$. Here $t = \{s\} \times \{s\}$. Let $\rho \in [I \to (\{s\} \times \{s\})] = [I \to \{s\}] \times [I \to \{s\}]$ be an encoding of $\{(r_1, r'_1), \ldots, (r_n, r'_n)\}$ given by $\rho = \{i_1 \mapsto (r_1, r'_1), \ldots, i_n \mapsto (r_n, r'_n)\}$. That is ρ is represented as two flat relations, $\rho = (\{i_1\} \bowtie r_1 \cup \ldots \cup \{i_n\} \bowtie r_n, \{i_1\} \bowtie r'_1 \cup \ldots \cup \{i_n\} \bowtie r'_n)$. Then $\rho' = \overline{map}(\cup)(\rho)$ is their union, $\rho' = \{i_1\} \bowtie r_1 \cup \ldots \cup \{i_n\} \bowtie r_n \cup \{i_1\} \bowtie r'_1 \cup \ldots \cup \{i_n\} \bowtie r'_n$, which corresponds to the encoding $\{i_1 \mapsto r_1 \cup r'_1, \ldots, i_n \mapsto r_n \cup r'_n\}$. In short, $\forall \alpha = 1, \ldots, n$, $\cup(\rho(i_\alpha)) = \cup(r_{i_\alpha}, r'_{i_\alpha}) = r_{i_\alpha} \cup r'_{i_\alpha}$ and is equal to $\rho'(i_\alpha)$. For $i \notin \{i_1, \ldots, i_n\}$, we have both $\cup(\rho(i))$ and $\rho'(i)$ equal to $\emptyset$.
2. $\overline{map}(-) \stackrel{\text{def}}{=} -$. The proof is similar to $\cup$.
3. For $\bowtie$: $\{s\} \times \{s'\} \to \{s \times s'\}$, we define $\overline{map}(\bowtie) : \{I \times s\} \times \{I \times s'\} \to \{I \times (s \times s')\}$ by $\overline{map}(\bowtie)(\rho, \rho') \stackrel{\text{def}}{=} \{(i, x, x') \mid (i, x) \in \rho, (i, x') \in \rho'\}$.

4. For $f = map(\varphi)$, $f : \{s\} \to \{s'\}$, we define $\overline{map}(f) \stackrel{\text{def}}{=} map((\pi_1, \varphi \circ \pi_2))$. That is, for $\rho \in \{I \times s\}$, ρ' is defined to be $\{(i, \varphi(x)) \mid (i,x) \in \rho\}$.

5. $\overline{map}(\sigma_p) \stackrel{\text{def}}{=} \sigma_{p \circ \pi_2}$. That is, for $\rho \in \{I \times s\}$, ρ' will be $\{(i,x) \mid (i,x) \in \rho, p(x) = true\}$.

6. $\overline{map}(\emptyset) \stackrel{\text{def}}{=} \emptyset$, $\overline{map}(\iota_{\{s\}}) \stackrel{\text{def}}{=} \iota_{\{I \times s\}}$. Some caution is necessary in these simple cases. By definition, $\emptyset : unit \to \{s\}$. Hence $\overline{map}(\emptyset) : [I \to unit] \to [I \to \{s\}]$. Here $[I \to unit] = unit$, and for any $\rho \in [I \to unit]$, and for all $i \in I$, $\rho(i) = ()^2$, hence $f(\rho(i)) = \emptyset$ (recall, here $f = \emptyset$). By construction of $\overline{map}(\emptyset)$, we get $\rho'(i) = \emptyset$ too.

7. For the last group, take $\overline{map}(id_t) \stackrel{\text{def}}{=} id_{[I \to t]}$, $\overline{map}(g \circ f) \stackrel{\text{def}}{=} \overline{map}(g) \circ \overline{map}(f)$, $\overline{map}(\pi_i) \stackrel{\text{def}}{=} \pi_i$ and $\overline{map}(f_1, f_2) \stackrel{\text{def}}{=} (\overline{map}(f_1), \overline{map}(f_2))$.

Finally, we define $\overline{map}(bfix(f,g)) \stackrel{\text{def}}{=} bfix(\overline{map}(f), \overline{map}(g))$, and $\overline{map}(fix(f)) \stackrel{\text{def}}{=} fix(\overline{map}(f))$. We sketch the correctness proof for $bfix_p$ (i.e. for the partial fixpoint semantics). Assume the types to be $f : t \times \{s\} \to \{s\}$ and $g : \{s\} \to \{s\}$. Let ρ be some input for $\overline{map}(bfix(f,g))$, that is $\rho \in [I \to t]$ say $\rho = \{i_1 \mapsto x_1, \ldots, i_n \mapsto x_n\}$. For each $\alpha = 1, \ldots, n$, $bfix(f,g)(x_\alpha)$ is the "limit" of the sequence y_α^k, $k \geq 0$, where $y_\alpha^0 = \emptyset$ and $y_\alpha^{k+1} = f(x_\alpha, y_\alpha^k) \cap g(x_\alpha)$. Define ρ^k to be $\{i_1 \mapsto y_1^k, \ldots, i_n \mapsto y_n^k\}$, for $k \geq 0$. Obviously $\rho^0 = \emptyset$, and by using induction hypothesis on f and g, we can show that $\rho^{k+1} = \overline{map}(f)(\rho, \rho^k) \cap \overline{map}(g)(\rho)$. For the intersection to work, it is essential that in $\overline{map}(f)$ and $\overline{map}(g)$ the corresponding elements, x_α and y_α^k, are encoded by the same index, i_α. This proves that $bfix(\overline{map}(f), \overline{map}(g))(\rho)$, which is the "limit" of ρ^k, is the function $\rho' = \{i_1 \mapsto bfix(f,g)(x_1), \ldots, i_n \mapsto bfix(f,g)(x_n)\}$. □

All functions $f : \{s_1\} \times \ldots \times \{s_m\} \to \{s\}$ in the strict fragment of $\mathcal{RA}$ and its extensions have the property $f(\emptyset, \ldots, \emptyset) = \emptyset$. Since $\emptyset \in r$ whenever $r \simeq \rho$, it follows that $\emptyset \in map(f)(r)$, so $map(f)(r)$ can indeed be encoded. By contrast, $map(null)(\{()\}) = \{\{()\}\}$ does not contain $\emptyset$, hence it cannot be encoded, and the strict map lemma is not true for the whole language $\mathcal{RA}$. We extend it next:

Lemma 5.5 (The Map Lemma) *Let $f : t \to t'$ be some function in $\mathcal{RA}(\Sigma)$ (+bfix, or +fix). Then there exists a function $\overline{\overline{map}}(f) : [I \Rightarrow t] \to [I \Rightarrow t']$ in $\mathcal{RA}(\Sigma)$ (+bfix, or +fix) which "simulates" $map(f)$ in the following strong way: for all $\varrho \in [I \Rightarrow t]$, if $\varrho' = \overline{\overline{map}}(f)(\varrho)$ then $Dom(\varrho') = Dom(\varrho)$, and $\forall i \in Dom(\varrho)$, $f(\varrho(i)) = \rho'(i)$. In other words, the following holds: $\{r_1, \ldots, r_n\} \cong \varrho \Longrightarrow \{f(r_1), \ldots, f(r_n)\} \cong \overline{\overline{map}}(f)(\varrho)$, and moreover, the same index is used to encode r_α and $f(r_\alpha)$, for $\alpha = 1, \ldots, n$.*

Proof. The proof proceeds again by induction on f. For all cases we define $\overline{\overline{map}}(f) : \{I\} \times [I \to t] \to \{I\} \times [I \to t']$ to be $(\pi_1, \overline{map}(f) \circ \pi_2)$, except for the case in which $f = null$. Here the types are $f : unit \to \{unit\}$, and $\overline{\overline{map}}(f) : [I \Rightarrow unit] \to [I \Rightarrow \{unit\}]$, which is the same as $\overline{\overline{map}}(f) : \{I\} \times unit \to \{I\} \times \{I \times unit\}$. We take $\overline{\overline{map}}(null)(\delta, \rho) \stackrel{\text{def}}{=} (\delta, \delta \bowtie !(\delta))$. Not surprisingly, $\overline{\overline{map}}(null)$ is strict: indeed, $\overline{\overline{map}}(f)$ is strict for any f. □

Note that the algebraic structure of I (i.e. the functions *pair*, *left*, *right*) is not necessary for the Map Lemma. The lemma is of independent interest, and allows us to prove certain properties for $\mathcal{RA}$. For example, we

[2] There is no contradiction with our assumption that ρ must have finite support. In this case the type of $\rho(i)$ has no sets at all, hence its support is empty, because there is no i for which $\rho(i)$ returns some non-empty set.

can show that, whenever $f : \{s\} \to \{s'\}$ is in $\mathcal{RA}$, then so is $g \stackrel{\mathrm{def}}{=} ext(\lambda x.f(\{x\}))$, of type $g : \{s\} \to \{s'\}$. The meaning of g is: $g(\{x_1, \ldots, x_n\}) = f(\{x_1\}) \cup \ldots \cup f(\{x_n\})$. To see this, take $I = s$, and compute first the following encoding of $\{\{x_1\}, \ldots, \{x_n\}\}$: $\varrho = \{x_1 \mapsto \{x_1\}, \ldots, x_n \mapsto \{x_n\}\}$. The type of ϱ is $[s \Rightarrow \{s\}] = \{s\} \times \{s \times s\}$, and it is easy to see that ϱ can be expressed in $\mathcal{RA}$. Next, we apply the Map Lemma to f, and compute $\varrho' = \overline{\overline{map}}(f)(\varrho)$. From here it is easy to compute g, because ϱ' is just $\{x_1 \mapsto f(\{x_1\}), \ldots, x_n \mapsto f(\{x_n\})\}$.

Finally, we can now define the translation of the types, and define the encoding relation.

Definition 5.6 *FLAT maps a complex object type t into a flat type FLAT(t), and is defined by:*

$$\begin{aligned}
FLAT(unit) &\stackrel{\mathrm{def}}{=} unit \\
FLAT(\mathbb{D}) &\stackrel{\mathrm{def}}{=} \{\mathbb{D}\} \\
FLAT(t \times t') &\stackrel{\mathrm{def}}{=} FLAT(t) \times FLAT(t') \\
FLAT(\{t\}) &\stackrel{\mathrm{def}}{=} [I \Rightarrow FLAT(t)]
\end{aligned}$$

The encoding relation $\sim_t \subseteq t \times FLAT(t)$ *is defined by:*

$$\begin{aligned}
() \sim_{unit} () & \\
x \sim_{\mathbb{D}} \{x\} & \\
(x, x') \sim_{t \times t'} (r, r') &\iff x \sim_t r \wedge x' \sim_{t'} r' \\
\{x_1, \ldots, x_n\} \sim_{\{t\}} \varrho &\iff \exists r_1 \in FLAT(t), \ldots, \exists r_n \in FLAT(t) \\
& \qquad x_1 \sim_t r_1 \wedge \ldots \wedge x_n \sim_t r_n \wedge \{r_1, \ldots, r_n\} \cong \varrho
\end{aligned}$$

For the last rule, we emphasize that $x_1, \ldots, x_n$ are not necessarily distinct, i.e. we may have several r_j's encoding the same x_i (and, in turn, each r_j may have several indexes assigned).

Objects of a scalar type t, like $t = \mathbb{D}$ or $t = \mathbb{D} \times \mathbb{D}$ have only one encoding, namely $(x_1, \ldots, x_m)$ is encoded by $(\{x_1\}, \ldots, \{x_m\})$. Once a set constructor is mentioned in t, there are infinitely many possible encodings for the same object, corresponding to the various choices of indexes. Conversely, not every relation of type $FLAT(t)$ represents a valid encoding, for two reasons: only singleton sets are valid encodings in $FLAT(\mathbb{D})$, and only partial functions (i.e. relations satisfying the constraint in Equation 2) are valid encodings in $FLAT(\{t\})$.

5.2 Flattening of Functions

For the translation of functions over complex objects into "flat functions" (i.e. over flat relations) we need the algebraic structure of I. The results in this subsection hold only under the assumption that $left \neq right$ and that $pair : I \times I \to I$ is injective. We start by describing the translation of $\mathcal{NRA}(\Sigma)$, i.e. without fixpoints. The fixpoints require some special care and will be treated next.

Proposition 5.7 (Flattening of functions) *For every function $f : t \to t'$ in $\mathcal{NRA}(\Sigma)$, there is some function $FLAT(f) : FLAT(t) \to FLAT(t')$ in $\mathcal{RA}(\Sigma \cup I)$, such that the following* **soundness condition** *is satisfied:*

$$x \sim_t r \Longrightarrow f(x) \sim_{t'} FLAT(f)(r)$$

We say that $FLAT(f)$ is obtained by **flattening** *f.*

Proof. The construction of $FLAT(f)$ proceeds by induction on the structure of f. We consider the presentation of $\mathcal{NRA}$, with *map*, μ and ρ_2 instead of ext_2, and with *doubleton* instead of $\cup$ (see the comments at the end of Subsection 2.2). The basic principle is to have $FLAT(f)(x)$ reuse as much as possible the indexes present in x. When f "creates" new sets, then $FLAT(f)$ has to "create" new indexes. This happens in three cases: when f is μ (flatten), η (singleton) and *doubleton*. For η we need one index constant: pick *left*. For $doubleton(x, y) = \{x, y\}$ we need two index constants: we use *left* and *right*. Finally, in μ we flatten a set of sets: here use the function *pair* to assign a new index to each pair of indexes. We illustrate some of the details next:

- $f = p$, where $p \in \Sigma$. Recall that we assume Σ to contain only scalar functions. For illustration, suppose p has type $p : d_p \to c_p$, with $d_p = \mathbb{D} \times \mathbb{D}$ and $c_p = \mathbb{D}$. Then $FLAT(d_p) = \{\mathbb{D}\} \times \{\mathbb{D}\}$, $FLAT(c_p) = \{\mathbb{D}\}$, and we define $FLAT(p)(\{x\}, \{y\}) = \{p(x, y)\}$ by $map(p) \circ \bowtie$. Also, $FLAT(eq_{\mathbb{D}})$ is translated with the aid of a selection over the predicate $eq_{\mathbb{D}}$.
- $FLAT(id_t) = id_{FLAT(t)}$, $FLAT(g \circ f) = FLAT(g) \circ FLAT(f)$, $FLAT(\pi_i) = \pi_i$, $FLAT((f_1, f_2)) = (FLAT(f_1), FLAT(f_2))$.
- $FLAT(map(f)) = \overline{\overline{map}}(FLAT(f))$. Here is where the map lemma is used.
- The case $FLAT(\mu)$ illustrates the need for the function *pair*. Recall $\mu : \{\{t\}\} \to \{t\}$ is the flattening operation. Intuitively, $FLAT(\mu)$ has to take some set of sets of values, with two levels of indexes, say $\{i_1 \mapsto \{i_{11} \mapsto x_{11}, i_{12} \mapsto x_{12}, \ldots\}, i_2 \mapsto \{i_{21} \mapsto x_{21}, i_{22} \mapsto x_{22}, \ldots\}, \ldots\}$, and to assign a unique index to each element $x_{11}, x_{12}, \ldots, x_{21}, \ldots$: namely we assign the index $pair(i_\alpha, i_{\alpha\beta})$ to each element $x_{\alpha\beta}$. The injectivity of *pair* guarantees that different elements will be assigned different indexes (but, of course, the same element may be assigned several indexes). Formally, let $t' = FLAT(t)$, and recall that the type of $FLAT(\mu)$ is $[I \Rightarrow [I \Rightarrow t']] \to [I \Rightarrow t']$. It is easy to see that $[I \to [I \to t]] = [I \times I \to t]$: for example, when $t = \{s\}$, then $[I \to [I \to \{s\}]] = [I \to \{I \times s\}] = \{I \times (I \times s)\} = \{I \times I \times s\} = [I \times I \to s]$. Then we have $[I \Rightarrow [I \Rightarrow t]] = \{I\} \times [I \to (\{I\} \times [I \to t])] = \{I\} \times (\{I \times I\} \times [I \to [I \to t]]) = (\{I\} \times \{I \times I\} \times [I \times I \to t])$. The first step in $FLAT(\mu)$ will be the projection, π_2, mapping this type into $\{I \times I\} \times [I \times I \to t]$. From here we define a mapping φ into $\{I\} \times [I \times t]$ using *pair*. To see how, assume for illustration that $t = \{s\}$: the general case $t = \{s_1\} \times \ldots \times \{s_m\}$ is handled similarly. Then $[I \times I \to \{s\}] = \{(I \times I) \times s\}$, and we take $\varphi = (map(pair), map((pair \circ \pi_1, \pi_2)))$: informally, $\varphi(\delta, \rho) = (\{pair(i_1, i_2) \mid (i_1, i_2) \in \delta\}, \{(pair(i_1, i_2), x) \mid ((i_1, i_2), x) \in \rho\})$. Finally, take $FLAT(\mu) \stackrel{\text{def}}{=} \varphi \circ \pi_2$. We use the fact that *pair* is injective when proving the soundness condition.
- $FLAT(\emptyset) = (\emptyset, \ldots, \emptyset)$.
- $FLAT(\eta)$ essentially assigns the index *left* to its unique element (it could also assign *right*). Next, for $FLAT(doubleton)$ we assign *left* to its first element and *right* to its second and use the fact that $left \neq right$ in order to prove the soundness condition. $FLAT(\rho_2)$ is, essentially, a product ($\bowtie$).

The soundness condition is easily checked in all cases.

□

To illustrate Proposition 5.7, consider $\cup$ and $\cap$. As derived operation, $\cup$ is *not* translated into $\cup$, but into $FLAT(\cup) = FLAT(\mu \circ \mathit{doubleton}) = FLAT(\mu) \circ FLAT(\mathit{doubleton})$. That is, the union $x \cup y$ of two sets $x = \{v_1, \ldots, v_m\}, y = \{w_1, \ldots, w_m\}$, encoded as $\{i_1 \mapsto v_1, \ldots, i_m \mapsto v_m\}$, $\{j_1 \mapsto w_1, \ldots, j_n \mapsto w_n\}$, will be encoded as $\{\mathit{pair}(\mathit{left}, i_1) \mapsto v_1, \ldots, \mathit{pair}(\mathit{left}, i_m) \mapsto v_m, \mathit{pair}(\mathit{right}, j_1) \mapsto w_1, \ldots, \mathit{pair}(\mathit{right}, j_n) \mapsto w_m\}$. Note how the encoding ensures that the indexes of the two encodings of x and y are distinct, before taking their union. But this also means that an element occurring both in x and in y will end up having two distinct indexes in the encoding of $x \cup y$. Things become even more complicated in the case of $\cap$: it is expressed in terms of other operators in $\mathcal{NRA}(\Sigma)$ (Proposition 2.4), and its translation is not $\cap$, but a rather complex expression.

Our next goal is to translate fixpoints too. Here we face the following problem. Let $\mathit{fix}(f)$ be the fixpoint to be translated, and suppose it reaches some fixpoint after n iterations, say $y_n = f(x, y_n)$, where $y_0 = \emptyset$ and $y_{k+1} = f(x, y_k)$. A candidate for the translation is $\mathit{fix}(f')$, with $f' = FLAT(f)$. This translation sounds promising, because if we define $r_0 = \emptyset$ and $r_{k+1} = f'(q, r_k)$ (with q an encoding of x), then it is easy to see that $y_k \sim r_k$, for all $k \geq 0$. But, in general, $\mathit{fix}(f')$ will not reach a fixpoint after the same number of iterations, that is $r_n \neq f'(r_n)$: it may be the case that $r_n, r_{n+1}, r_{n+2}, \ldots$ are all distinct encodings of the same complex object y_n, because f' may continuously rename indexes and or artificially assign new indexes to elements, and never reach a fixpoint. We describe next how to get around this problem. We will show first for the bounded fixpoint, which is easier because we can use the bound to normalize the choice of indexes. Then we deal with the unbounded fixpoint.

We write $\varrho \sqsubseteq \varrho'$ for $\varrho, \varrho' \in [I \to t]$, when ϱ, as a partial function, is a restriction of ϱ'. This implies that $\varrho \subseteq \varrho'$ (component-wise inclusion of relations), but also that $\forall i \in \mathit{Dom}(\varrho)$, $\varrho(i) = \varrho'(i)$. This is stronger than $\varrho \subseteq \varrho'$, because the latter only implies $\varrho(i) \subseteq \varrho'(i)$.

Lemma 5.8 (Bounded additiveness of the encoding) *Let $x, x', x'' \in \{t\}$ and $r, r', r'' \in FLAT(\{t\})$, such that:*

- $x \sim_{\{t\}} r$, $x' \sim_{\{t\}} r'$, $x'' \sim_{\{t\}} r''$, *and,*
- $x \subseteq x''$, $x' \subseteq x''$, $r \sqsubseteq r''$, $r' \sqsubseteq r''$ *(we say that x, x', and r, r' respectively, are* bounded*).*

Then $x \cup x' \sim_{\{t\}} r \cup r'$.

Proof. Recall that $FLAT(\{t\}) = [I \Rightarrow FLAT(t)]$. Then $x = \{x_1, \ldots, x_m\}$, $x' = \{x'_1, \ldots, x'_{m'}\}$, $x'' = \{x''_1, \ldots, x''_{m''}\}$, and $\mathit{CoDom}(r) = \{r_1, \ldots, r_m\}$, $\mathit{CoDom}(r') = \{r'_1, \ldots, r'_{m'}\}$, $\mathit{CoDom}(r'') = \{r''_1, \ldots, r''_{m''}\}$, such that $x_1 \sim r_1, \ldots, x_m \sim r_m$, etc. It suffices to prove that $\mathit{CoDom}(r \cup r') = \mathit{CoDom}(r) \cup \mathit{CoDom}(r')$. Let $\delta = \mathit{Dom}(r), \delta' = \mathit{Dom}(r')$. Obviously $\mathit{Dom}(r \cup r') = \delta \cup \delta'$. So let $i \in \delta \cup \delta'$. There are three cases: (1) $i \in \delta, i \notin \delta'$, then $(r \cup r')(i) = r(i)$. (2) $i \in \delta', i \notin \delta$ is similar. (3) $i \in \delta \cap \delta'$. This is the interesting case which fails unless we assume boundedness. With this assumption we have $r(i) = r''(i)$ and $r'(i) = r''(i)$ and, hence $r(i) = r'(i) = (r \cup r')(i)$. Finally, $\mathit{CoDom}(r \cup r') = \mathit{CoDom}(r) \cup \mathit{CoDom}(r')$ follows from the following three equalities: $(r \cup r')(\delta - \delta') = r(\delta - \delta')$, $(r \cup r')(\delta' - \delta) = r(\delta' - \delta)$ and $(r \cup r')(\delta \cap \delta') = r(\delta \cap \delta') = r'(\delta \cap \delta')$.

□

Lemma 5.9 (Strong intersection) *For every type t there is some expression:*

$$intrs : FLAT(\{t\}) \times FLAT(\{t\}) \to FLAT(\{t\})$$

in $\mathcal{RA}(I)$ such that $x \sim r$, $x' \sim r'$ implies (1) $x \cap x' \sim intrs(r, r')$ and (2) $intrs(r, r') \sqsubseteq r'$. Moreover, $intrs(r, r')$ is the largest element (with respect to the relation $\sqsubseteq$) satisfying (1) and (2).

Proof. (Sketch) Intuitively $intrs(r, r')$ has to encode the intersection with indexes already used in r', and, moreover, has to use *all* indexes from r', whenever some element in $x \cap x'$ is encoded with more than one index in r'. Formally, recall that $member : t \times \{t\} \to \{unit\}$ can be defined in $\mathcal{NRA}$ (see Proposition 2.3), and, hence, by Proposition 5.7, is translated into $FLAT(member) : FLAT(t) \times FLAT(\{t\}) \to \{unit\}$. We define $intrs(r, r')$ as the restriction of r' to those indexes i for which $FLAT(member)(r'(i), r)$ is true (this can be expressed in $\mathcal{RA}$): obviously $intrs(r, r')$ satisfies (1) and (2). To prove that $intrs(r, r')$ is the largest such element, consider some q satisfying (1) $x \cap x' \sim q$ and (2) $q \sqsubseteq r'$. Then $\forall i \in Dom(q)$, (1) implies $\exists y \in x$ such that $y \sim q(i)$, and (2) implies $q(i) = r'(i)$. Hence $FLAT(member)(r'(i), r)$ is true, so $intrs(r, r')(i)$ is defined and is equal to $r'(i)$, and to $q(i)$. So $q \sqsubseteq intrs(r, r')$.

□

Now we can extend Proposition 5.7 to bounded fixpoints:

Proposition 5.10 *All functions from $\mathcal{NRA}(\Sigma) + bfix$ can be flattened into $\mathcal{RA}(\Sigma \cup I) + bfix$ such that the soundness property holds (see Proposition 5.7).*

Proof. We only have to extend the induction proof for Proposition 5.7 to functions of the form $bfix(f, g) : t \to \{t'\}$, where $f : t \times \{t'\} \to \{t'\}$ and $g : t \to \{t'\}$. Let $\varphi = FLAT(f)$ and $\gamma = FLAT(g)$, of types $\varphi : FLAT(t) \times FLAT(\{t'\}) \to FLAT(\{t'\})$, and $\gamma : FLAT(t) \to FLAT(\{t'\})$. Also let $\psi(r, r') = \text{intrs}(\varphi(r, r'), \gamma(r))$. Define $FLAT(bfix(f, g)) = bfix(\psi, \gamma)$. We check the soundness property as follows. For the partial bounded fixpoint, consider some $x \sim r$, define $x'_0 = \emptyset$, $r'_0 = \emptyset$ (more precisely r'_0 is a tuple of empty sets), and $x'_{n+1} = f(x, x'_n) \cap g(x)$, $r'_{n+1} = \psi(r, r'_n) \cap \gamma(r)$. By induction on n, we prove that $x'_n \sim r'_n$. Indeed:

$$\begin{array}{rcl} x'_{n+1} & = & f(x, x'_n) \cap g(x) \\ & \sim & intrs(\varphi(r, r'_n), \gamma(r)) \text{ by induction and (1) of Lemma 5.9} \\ & = & intrs(\varphi(r, r'_n), \gamma(r)) \cap \gamma(r) \text{ by (2) of Lemma 5.9} \\ & = & r'_{n+1} \end{array}$$

Obviously $r'_n = r'_{n+1}$ implies $x'_n = x'_{n+1}$, but we also have to prove the converse: in general, if $x'_n \sim r'_n$ and $x'_n = x'_{n+1} \sim r'_{n+1}$, there is no reason why $r'_n = r'_{n+1}$. So suppose $f(x, x'_{n-1}) \cap g(x) = f(x, x'_n) \cap g(x)$. Then $intrs(\varphi(r, r'_{n-1}), \gamma(r))$ and $intrs(\varphi(r, r'_n), \gamma(r))$ are both the largest subset of $\gamma(r)$ encoding $f(x, x'_n) \cap g(x)$, hence they must be equal, which implies $r'_n = r'_{n+1}$. Thus, we have proven $bfix(f, g)(x) \sim bfix(\varphi, \gamma)(r)$.

For the inflationary fixpoint we proceed similarly by defining $x'_{n+1} = x'_n \cup (f(x, x'_n) \cap g(x))$ and $r'_{n+1} = r'_n \cup (\psi(r, r'_n) \cap \gamma(r))$, and use Lemma 5.8 in the induction step of the proof of $x'_n \sim r'_n$.

Note how the bounding function g has helped us in constructing the translation.

□

The unbounded fixpoint is slightly more difficult to flatten, because there is no bound which can help us choosing the indexes in a consistent way. In defining $FLAT(\mathit{fix}(f))$ as $\mathit{fix}(\varphi)$, we must make sure that φ chooses the indexes in a systematic way.

Proposition 5.11 *All functions from $\mathcal{NRA}(\Sigma) + \mathit{fix}$ can be flattened into $\mathcal{RA}(\Sigma \cup I) + \mathit{fix}$ such that the soundness property holds (see Proposition 5.7).*

Proof. (Sketch) We will define $FLAT(\mathit{fix}(f))$, where $f : t \times \{t'\} \to \{t'\}$ as $\mathit{fix}(\varphi)$, for some $\varphi : FLAT(t) \times FLAT(\{t'\}) \to FLAT(\{t'\})$. $\varphi(r, r')$ will essentially compute $q = FLAT(f)(r, r')$, and then will change some indexes in r'' such that r'' uses the same indexes and the same encodings as r', for all elements common in the decoding of r' and q. This will ensure that if r' and q encode the same element, then they are actually equal. We start by splitting q into $= q_1 \cup q_2$, where q_1 contains the encodings of all elements *common* with r', and q_2 contains the rest (more precisely $q_1 = \mathit{intrs}(r', q)$ and $q_2 = q - q_1$). To change the indexes in q, we start by changing the indexes in q_1 with those occurring in r', i.e. by replacing q_1 with $q'_1 = \mathit{intrs}(q, r')$. Now we would like to define $\varphi(r) = q'_1 \cup q_2$, but the problem is that q_2 may now have common indexes with q'_1, and we have to rename these indexes before computing the union. So consider some $i_0 \in Dom(q'_1) \cap Dom(q_2)$. Since the element encoded $q'_1(i_0)$ was encoded in q_1 already, there must be at least one index $i_1 \in Dom(q_1)$ such that $q'_1(i_0)$ and $q_1(i_1)$ encode the same element (more precisely: $FLAT(eq_t(q'_1(i_0), q_1(i_1)))$ is true). There may be several such i_1's: if at least one is not used in q'_1, then we can replace the index i_0 in the encoding of $q_2(i_0)$ with such an i_1 (in fact with all of them, because in the absence of an order relation we cannot pick a particular one). Otherwise, for each i_1, there is at least one $i_2 \in Dom(q_1)$ such that $q'_1(i_1)$ and $q_1(i_2)$ encode the same element. In this way we construct a tree rooted at i_0 and whose branches are $i_0 \to i_1 \to i_2 \ldots$. One can see that there are no cycles, because $i_k = i_l$ would imply that $q_1(i_k)$ and $q_1(i_l)$ encode the same element (to be precise: $FLAT(eq(q_1(i_k), q_1(i_l)))$ is true), hence $q'_1(i_{k-1})$ and $q'_1(i_{l-1})$ encode the same element, hence $i_{k-1} = i_{l-1}$; since i_0 is distinct from all other i_k's (it is the only one $\in Dom(q_2)$, hence $\notin Dom(q_1)$), cycles are impossible. Hence it suffices to find the leaves i_k of the tree, and re-encode $q_2(i_0)$ with all such indexes i_k. □

We argued at the end of Section 4 that our encoding technique can be used to implement complex object databases in terms of relational databases. Proposition 5.11 states formally, and proves half of the statement in Fact 4.3: every query in $\mathcal{NRA}(\Sigma) + \mathit{fix}$ can be encoded in $\mathcal{RA}(\Sigma \cup I) + \mathit{fix}$. For unbounded fixpoints, this is all we can do: we cannot get around the functions *left, right, pair* in the translated query. For bounded fixpoints however we may go a step further, and eliminate the need for *left, right, pair*: we show this in the next subsection.

5.3 Elimination of Indexes for Encodings of Flat Relations

Our second step in the proof of Theorem 4.1 consists in eliminating the indexes from the encodings in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$. Let $f : t \to t'$ be some function in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$, where t, t' are flat relation types. Proposition 5.10 tells us that f can be "simulated" in $\mathcal{RA}(\Sigma \cup I) + \mathit{bfix}$, by $FLAT(f) : FLAT(t) \to FLAT(t')$. It is important for this subsection that Proposition 5.10 holds for *any* index I, as long as $\mathit{left} \neq \mathit{right}$ and *pair* is injective, because in this subsection we will choose a particular index I.

Keeping in mind Theorem 4.1, we start by composing $FLAT(f)$ with $\mathit{decode} : FLAT(t') \to t'$ (which is expressible in $\mathcal{RA}$, because t' is a flat relation type), to get some function $g : FLAT(t) \to t'$. By Proposition 5.7, $x \sim_t r$ implies $f(x) \sim_{t'} FLAT(f)(r)$. It follows that $x \sim_t r$ implies $f(x) = g(r)$. Now we have to construct some function $h : t \to t'$ in $\mathcal{RA}(\Sigma) + \mathit{bfix}$ (without indexes) which, on some input x (1) "chooses" some indexes to encode x in $FLAT(t)$, and (2) performs all computations done by g, but without using $I, \mathit{left}, \mathit{right}, \mathit{pair}$. The idea in (1) is to use the elements in x as indexes for themselves. The idea in (2) is to use tuple concatenation instead of *pair*, and use any constant, say the empty tuple, $()$, instead of *left* and *right*. Hence, we end up having indexes of different types in different intermediate results. The difficult task here is to reconcile the types.

Throughout this subsection we shall illustrate with the following example:

Example 5.12 *Let* $t = \{s_1\} \times \{s_2\}$, *where* $s_1 = \mathbb{D} \times \mathbb{D}$ *and* $s_2 = \mathbb{D}$: *then* $FLAT(t) = [I \Rightarrow \{s_1\}] \times [I \Rightarrow \{s_2\}] = (\{I\} \times \{I \times \mathbb{D}\} \times \{I \times \mathbb{D}\}) \times (\{I\} \times \{I \times \mathbb{D}\})$.

In general, there is no function $\mathit{encode} : t \to FLAT(t)$ in $\mathcal{RA}$, with the property: $x \sim_t \mathit{encode}_I(x)$, $\forall x \in t$, simply because such a function would not be generic. A way around that is to use the elements of x as indexes themselves. In Example 5.12, if $x = (\{(a,b),(a,c)\},\{a,d\})$, then essentially we have to assign indexes to two sets: $\{(a,b),(a,c)\}$, and $\{a,d\}$. For example, a valid encoding could be $r = (\varrho_1, \varrho_2)$, where:

$$\begin{aligned}
\varrho_1 &= \{i_1 \mapsto (a,b), i'_1, \mapsto (a,c)\} \\
&= (\{i_1, i'_1\}, \{(i_1,a),(i'_1,a)\}, \{(i_1,b),(i'_1,c)\}) \\
\varrho_2 &= \{i_2 \mapsto a, i'_2 \mapsto d\} \\
&= (\{i_2, i'_2\}, \{(i_2,a),(i'_2,d)\})
\end{aligned}$$

So it suffices to pick $i_1 = (a,b), i'_1 = (a,c), i_2 = a, i'_2 = d$. For that, we will choose our index set I such as to contain the disjoint union $(\mathbb{D} \times \mathbb{D}) \uplus \mathbb{D}$. We make the notation $I_1 = \mathbb{D} \times \mathbb{D}$, $I_2 = \mathbb{D}$. Then $I_1 \uplus I_2 \subseteq I$. To encode x, we use each element as an index for itself. In Example 5.12, we obtain the encoding:

$$\begin{aligned}
r &= \mathit{encode}(x) \\
&= ((\{(a,b)_1,(a,c)_1\},\{((a,b)_1,a),((a,c)_1,c)\},\{((a,b)_1,b),((a,c)_1,c)\}),(\{a_2,d_2\},\{(a_2,a),(d_2,d)\}))
\end{aligned}$$

(To help in reading the notations, we mark with an index 1 the elements from I_1, and with 2 those from I_2.)

In general, we will have $t = \{s_1\} \times \ldots \times \{s_m\}$, and then we choose I to include a disjoint union $I_1 \uplus \ldots \uplus I_m$, where $I_1 = s_1, \ldots, I_m = s_m$. Of course, I must also contain two distinct elements *left*, *right*, and must be closed under the application of *pair*. Thus we define I to be the set of all (nested) tuples which can be constructed from the constants in $I_1 \uplus \ldots \uplus I_m \uplus \{\mathit{left}, \mathit{right}\}$. For example, I will contain tuples like (left, x_3) and $((y_2, z_1), (\mathit{right}, y_2))$, the latter being different from, say, $(y_2, ((z_1, \mathit{right}), y_2))$. That is, tuples of the same length may have different structures. Moreover, the set $\{\mathit{left}, \mathit{right}\}$ will be expressed as $\mathit{unit} \uplus \mathit{unit}$, and we denote $I_{\mathit{left}} = I_{\mathit{right}} = \mathit{unit}$.

Obviously, we can replace the function *pair* with tuple concatenation, an operation which is expressible in $\mathcal{RA}$ (without indexes). Next, we plan to eliminate I altogether, because it consists of disjoint unions of types already in $\mathcal{RA}$. The difficulty is keeping track of the types of the intermediate results. Some of them may have indexes which belong to several $\mathcal{RA}$ types. For example after *map*-ing the function *pair*, we obtain longer tuples, and after applying $\cup$ we may mix tuples of different lengths or structures. We therefore need to transform g to make sure that all its intermediate results are homogeneous relations (i.e. well-typed).

To solve this problem, we introduce the notion of a tag. A **tag** is given by the grammar:

$$\tau ::= \mathit{left} \mid \mathit{right} \mid 1 \mid \ldots \mid m \mid (\tau, \tau)$$

Then, for every tag τ we define the subset I_τ of our particular I to be: $I_{\mathit{left}}, I_{\mathit{right}}, I_1, \ldots, I_m$ are those described above, and $I_{(\tau,\tau')} \stackrel{\text{def}}{=} I_\tau \times I_{\tau'}$. In fact $I = \biguplus_{\tau \in \text{Tags}} I_\tau$. A **tagged type** $\bar{t}$ is a type t in which every occurrence of I has some tag. For example, $\bar{t} = \{I_{(2,4)} \times \mathbb{D}\} \times \{I_{(3,\mathit{left})}\}$ is a tagged type: its values are only a subset of the untagged type $t = \{I \times \mathbb{D}\} \times \{I\}$, namely those pairs (x, y) for which $y \subseteq I_{(3,\mathit{left})}$ and $\Pi_1(x) \subseteq I_{(2,4)}$. However, and most importantly, a tagged type is isomorphic to some type without indexes: $\bar{t}$ in the example above is isomorphic to $\{(s_2 \times s_4) \times \mathbb{D}\} \times \{s_3 \times \mathit{unit}\}$. Also, we define the language of **tagged expressions (or tagged functions)** of $\mathcal{RA}(\Sigma \cup I) + \mathit{bfix}$ in the same way as $\mathcal{RA}(\Sigma \cup I) + \mathit{bfix}$, but in which each I_τ is a type in itself, *left*, *right* have types $\mathit{unit} \to I_{\mathit{left}}$ and $\mathit{unit} \to I_{\mathit{right}}$ respectively, and *pair* is replaced by a family of functions $\mathit{pair}_{\tau,\tau'} : I_\tau \times I_{\tau'} \to I_{(\tau,\tau')}$. Obviously, each tagged function $\bar{g} : \bar{t} \to \bar{t}'$ corresponds to some untagged function $g : t \to t'$, obtained by simply erasing the tags.

Recall that our goal is to express some function $g : \mathit{FLAT}(t) \to t'$ by some "equivalent" function $h : t \to t'$ in $\mathcal{RA}(\Sigma) + \mathit{bfix}$ without indexes. Due to our choice of encoding, the input type of g has a canonical tagging, obtained by assigning the tags $1, 2, \ldots, m$ to each occurrence of I. In Example 5.12 we tag $\mathit{FLAT}(t)$ by $(\{I_1\} \times \{I_1 \times \mathbb{D}\} \times \{I_1 \times \mathbb{D}\}) \times (\{I_2\} \times \{I_2 \times \mathbb{D}\})$. The next lemma proves that for any function $g : \bar{t} \to t'$, the tags can be propagated from the input $\bar{t}$ to the output t', and, in the process, g can be replaced by a tagged function. Essentially, this boils down to keeping track of the structure of the indexes in the intermediate results. For example, when $g : \{I_\tau \times I_{\tau'} \times t\} \to \{I \times t\}$ is $\mathit{map}((\mathit{pair} \circ \pi_1, \pi_2))$, then we tag the output type by $\{I_{(\tau \times \tau')} \times t\}$, and tag g by $\mathit{map}((\mathit{pair}_{\tau,\tau'} \circ \pi_1, \pi_2))$. However there is a slight technical difficulty: when computing union, we may need to mix indexes of different tags. In that case, we simply keep the resulting type as a disjoint union of tagged types. We wrap up these ideas in the following lemma:

Lemma 5.13 (Tagging lemma) *Let $g : t \to t'$ be some function in $\mathcal{RA}(\Sigma \cup I) + \mathit{bfix}$, and let $\bar{t}_1, \ldots, \bar{t}_k$ be k taggings of t. Then there exists some l taggings of t', $\bar{t}'_1, \ldots, \bar{t}'_l$, and a tagged function $\bar{g} : \bar{t}_1 \times \ldots \times \bar{t}_k \to \bar{t}'_1 \times \ldots \times \bar{t}'_l$, say $\bar{g} = (\bar{g}_1, \ldots, \bar{g}_l)$, such that: $\forall r_1 \in \bar{t}_1, \ldots, \forall r_k \in \bar{t}_k$, $g(r_1 \cup \ldots \cup r_k) = \bar{g}_1(r_1, \ldots, r_k) \cup \ldots \cup \bar{g}_l(r_1, \ldots, r_k)$. Here $r_1 \cup \ldots \cup r_k$ simply denotes union when t is a set type, or component-wise union, when t is a product of set types.*

Proof. By induction on the structure of g, according to the definition of $\mathcal{RA}(\Sigma)$ given at the beginning of Section 5.

1. $g = \cup$ (union). Then $t = \{s\} \times \{s'\}$, where $s = s'$, and we are given some k taggings of t, $\bar{t}_1 = \{\bar{s}_1\} \times \{\bar{s}'_1\}, \ldots, \bar{t}_k = \{\bar{s}_k\} \times \{\bar{s}'_k\}$. We are searching for some expression $\bar{g}$ such that the "meaning" of $\bar{g}$ is $\bar{g}((r_1, r'_1), \ldots, (r_k, r'_k)) = (r_1 \cup \ldots \cup r_k) \bigcup (r'_1 \cup \ldots \cup r'_k)$: we are allowed to split the right hand side into fragments, according to their types. To help the intuition, assume that the k taggings of s are distinct from those for s'. Then none of the expressions $r_\alpha \cup r'_\beta$ are type correct. So we split the right hand side into $2k$ independent expressions, that is define $\bar{g}((r_1, r'_1), \ldots, (r_k, r'_k)) \stackrel{\text{def}}{=} (r_1, r_2, \ldots, r_k, r'_1, \ldots, r'_k)$. Thus we define $l = 2k$ and $\bar{t}'_1 = \{\bar{s}_1\}, \ldots, \bar{t}'_k = \{\bar{s}_k\}, \bar{t}'_{k+1} = \{\bar{s}'_1\}, \ldots, \bar{t}'_{2k} = \{\bar{s}'_k\}$. For uniformity, we keep the same construction also for the case when some $\bar{s}_\alpha$'s are the same as some $\bar{s}'_\beta$'s.

2. $g = -$ (difference). Consider the same notations as above: here we search for some $\bar{g}$ with the meaning $\bar{g}((r_1, r'_1), \ldots, (r_k, r'_k)) = (r_1 \cup \ldots \cup r_k) - (r'_1 \cup \ldots \cup r'_k)$. Again, we may split the right hand side into a union of differently typed expressions. First $\bar{g}$ will combine arguments of the same type, i.e. will replace r_α and r_β with $r_\alpha \cup r_\beta$ whenever $\bar{s}_\alpha = \bar{s}_\beta$, and similarly for r'_α, r'_β. Thus, we may assume w.l.o.g. that the types $\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_k$ are distinct, and that the types $\bar{s}'_1, \bar{s}'_2, \ldots, s'_{k'}$ are distinct too[3]. Here we will split the result of $\bar{g}$ into k distinctly typed expressions, i.e. define $l \stackrel{\text{def}}{=} k$, $\bar{t}'_1 \stackrel{\text{def}}{=} \{\bar{s}_1\}, \ldots, \bar{t}'_k \stackrel{\text{def}}{=} \{\bar{s}_k\}$, and $\bar{g} \stackrel{\text{def}}{=} (\bar{g}_1, \ldots, \bar{g}_k)$. For each $\alpha = 1, \ldots, k$, $\bar{g}_\alpha$ is defined as follows. We distinguish two cases. Case (1): there is some type $\bar{s}'_\beta$ equal to $\bar{s}_\alpha$. Then define $\bar{g}_\alpha$ to be $r_\alpha - r'_\beta$. Case (2): all types $\bar{s}'_\beta$ are distinct from $\bar{s}_\alpha$. Here take $\bar{g}_\alpha$ to be r_α. To see why this works, observe that $(r_1 \cup \ldots \cup r_k) - (r'_1 \cup \ldots \cup r'_k) = (r_1 - \bigcup_\beta r'_\beta) \cup \ldots \cup (r_k - \bigcup_\beta r'_\beta)$. Now recall that r_α and r_β are disjoint sets whenever $\bar{s}_\alpha \neq \bar{s}'_\beta$, and hence $r_\alpha - \bigcup_\beta r'_\beta$ is either $r_\alpha - r_\beta$ (in case (1)), or r_α (in case (2)).

3. $g = \bowtie$ (product). Here we simply join the tags. With the above notation, take $l = k^2$, $\bar{t}'_{(\alpha,\beta)} = \{\bar{s}_\alpha \times \bar{s}'_\beta\}$, and define $\bar{g}((r_1, r'_1), \ldots, (r_k, r'_k)) = ((r_1 \bowtie r'_1), \ldots, (r_1 \bowtie r'_k), (r_2 \bowtie r'_1), \ldots, (r_k \bowtie r'_k))$.

4. $g = map(\varphi)$, $g : \{s\} \to \{s'\}$ where $\varphi : s \to s'$ is some scalar function. Take $l = k$ and $\bar{t}'_\alpha = \{\bar{s}'_\alpha\}$, where $\bar{s}'_\alpha$ is obtained by propagating the tags for $\varphi : s_\alpha \to s'$, as illustrated in the example before this lemma.

5. $g = \sigma_p$ (selection). Here we keep the same tags. That is $\sigma_p : t \to t'$, with $t = t' = \{s\}$. Given k taggings of t, $\bar{t}_1, \ldots, \bar{t}_k$, we define $l \stackrel{\text{def}}{=} k$, and $\bar{t}'_1 \stackrel{\text{def}}{=} \bar{t}_1, \ldots, \bar{t}'_k \stackrel{\text{def}}{=} \bar{t}_k$. Finally, $\bar{g} = (\bar{g}_1, \ldots, \bar{g}_k)$, where each $\bar{g}_\alpha(r_1, \ldots, r_k) = \sigma_p(r_\alpha)$.

6. $\emptyset, null, \iota$ do not involve any tags.

7. id, π_1, π_2 and (g, g') are either trivial or do not involve tags at all. For composition $g' \circ g$, apply first induction hypothesis for $g : t \to t'$ and the k taggings for t to get l taggings for t', next apply induction hypothesis for g' and the l taggings for t'.

Finally, we handle $bfix(g, h) : t \to t'$, where $g : t \times t' \to t'$ and $h : t \to t'$. Recall that h is a bounding function, and that we compute the fixpoint of $\lambda(r, r').g(r, r') \cap h(r)$. Here we start with the k taggings of t and apply induction hypothesis to h, to get $\bar{h}$ and l taggings of t', $\bar{t}'_1, \ldots, \bar{t}'_l$: because the result of $bfix(g, h)$ is included in h we keep as final result to be these l possible taggings, although we may discover in the sequel that some of the l components will be empty. Next we construct the kl possible taggings for $t \times t'$, and apply induction hypothesis for g; we get $\bar{g}$ and some other m taggings for t', say $\bar{t}''_1, \ldots, \bar{t}''_m$. Since at each iteration of the

[3]As a consequence of this combination step, we may end up with $k \neq k'$, where k is the number of inputs $r_1, \ldots, r_k$, while k' is the number of inputs $r'_1, \ldots, r'_{k'}$.

fixpoint we only retain the intersection of g with h, we keep only those components of $\bar{g}$ whose tagged type $\bar{t}''_\alpha$ occurs among $\bar{t}'_1, \ldots, \bar{t}'_l$, and reorder them such as to produce a value of type $\bar{t}'_1 \times \ldots \times \bar{t}'_l$, by putting $\emptyset$ on positions corresponding to those $\bar{t}'_\beta$ which do not occur among the $\bar{t}''_\alpha$'s. The resulting function has type $\prod_{\alpha,\beta}(\bar{t}_\alpha \times \bar{t}'_\beta) \to (\bar{t}'_1 \times \ldots \times \bar{t}'_l)$. We compose it with the function $(\bar{t}_1 \times \ldots \times \bar{t}_k) \times (\bar{t}'_1 \times \ldots \times \bar{t}'_l) \to \prod_{\alpha,\beta}(\bar{t}_\alpha \times \bar{t}'_\beta)$ mapping $((r_1, \ldots, r_k), (r'_1, \ldots, r'_l))$ into $((r_1, r'_1), (r_1, r'_2), \ldots, (r_1, r'_l), (r_2, r'_1), \ldots, (r_k, r'_l))$. Call the resulting function $\bar{g}'$; its type is $(\bar{t}_1 \times \ldots \times \bar{t}_k) \times (\bar{t}'_1 \times \ldots \times \bar{t}'_l) \to (\bar{t}'_1 \times \ldots \times \bar{t}'_l)$. Finally take $\overline{\mathit{bfix}(g,h)} \stackrel{\text{def}}{=} \mathit{bfix}(\bar{g}', \bar{h})$.

□

Now we can conclude the proof of Fact 4.3. Suppose we are given some encoding of a complex object database in terms of a relational database, using indexes from a set I. Given a query $f : t_1 \to t_2$ in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$, we first flatten it to $g : t'_1 \to t'_2$, where $t'_1 = \mathit{FLAT}(t_1), t'_2 = \mathit{FLAT}(t_2), g = \mathit{FLAT}(f)$. Now we eliminate all references to *left*, *right*, *pair* from g as follows. First choose some arbitrary tagging $\bar{t}'_1$ for t'_1, for example by tagging all occurrences of I with I_1. Using Lemma 5.13, "propagate" the taggings from $\bar{t}'_1$ to t'_2. This gives us a l tagged versions of g, $\bar{g}_1, \ldots, \bar{g}_l$. Next replace repeatedly in the intermediate types of $\bar{g}_1, \ldots, \bar{g}_l$ every occurrence of $I_{(\tau,\tau')}$ with $I_\tau \times I_{\tau'}$ (and repeat, for I_τ and $I_{\tau'}$), every occurrence of I_{left} with *unit* and every occurrence of I_{right} with *unit* too. Then replace in $\bar{g}_1, \ldots, \bar{g}_l$ every occurrence of $\mathit{pair}_{(\tau,\tau')}$ with the identity function[4], and *left*, *right* with $()$. We obtain an equivalent formulation of $\bar{g}_1, \ldots, \bar{g}_l$, in which *pair* has been replaced with tuple concatenation, and *left*, *right* with the empty tuple.

5.4 Proof of the Conservativity Theorem

Proposition 5.14 *If all function symbols in Σ have set height 0, then $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ is a conservative extension of $\mathcal{RA}(\Sigma) + \mathit{bfix}$, for both inflationary and partial fixpoint semantics.*

Proof. Let $f : t \to t'$ in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$, where t, t' are flat relation types, $t = \{s_1\} \times \ldots \times \{s_m\}$. By Proposition 5.10 there is some function $\mathit{FLAT}(f) : \mathit{FLAT}(t) \to \mathit{FLAT}(t')$ which "simulates" f. By composing it with $\mathit{decode} : \mathit{FLAT}(t') \to t'$, we get $g : \mathit{FLAT}(t) \to t'$ such that $x \sim r \Longrightarrow f(x) = g(r)$. Now we choose the particular index set containing $I_1 \uplus \ldots \uplus I_m$, as described in Subsection 5.3, tag the type $\mathit{FLAT}(t)$ accordingly, and apply Lemma 5.13 to tag the function g: in applying this lemma, we have $k = 1$ (there is only one tagging of the input type), and we will get $l = 1$, because, the output type of g, t', does not contain indexes. Thus, the tagged function is $\bar{g} : \overline{\mathit{FLAT}(t)} \to t'$. Next, replace the tagged indexes I_τ in $\bar{g}$ with their corresponding types in $\mathcal{RA}(\Sigma)$, i.e. replace $I_1, \ldots, I_m$ with $s_1, \ldots, s_m$ respectively, $I_{\mathit{left}}, I_{\mathit{right}}$ with *unit*, and $I_{(\tau,\tau')}$ with $s \times s'$, whenever I_τ is replaced by s and $I_{\tau'}$ is replaced by s'. Similarly, transform $\bar{g}$ by replacing $\mathit{pair}_{\tau,\tau'}$ with the identity $\mathit{id}_{s \times s'} : s \times s' \to s \times s'$, *left* and *right* with the identity on *unit*. In this way, we obtain a function $\bar{\bar{g}} : \bar{\bar{t}} \to t'$ in $\mathcal{RA}(\Sigma) + \mathit{bfix}$, without indexes, which "simulates" f. Here $\bar{\bar{t}}$ is the type $\overline{\mathit{FLAT}(t)}$ in which $I_1, \ldots, I_m$ have been replaced with $s_1, ldots, s_m$ respectively. Finally, observe that we can express in $\mathcal{RA}$ the standard encoding $\mathit{encode} : t \to \bar{\bar{t}}$, as illustrated after Example 5.12. Then $\bar{\bar{g}} \circ \mathit{encode}$ is the function in $\mathcal{RA}(\Sigma) + \mathit{bfix}$ computing f.

□

To complete the proof of Theorem 4.1, we have to strengthen the statement in Proposition 5.14 to the

[4]Here, the identity $\mathit{id} : I_\tau \times I_{\tau'} \to I_\tau \times I_{\tau'}$ can be thought of as tuple concatenation.

case when the function $f : t \to t'$ has set height ≤ 1, rather than both types t, t' being flat relation types. The difference is that now t, t' may contain, as factors, some scalar types, and we have to show how their presence can be isolated before Proposition 5.14 can be applied. Note that, without loss of generality, we may consider that t has the form $t \times s$, where s is a scalar type and t is a flat relation type. Also it suffices to consider the case when t' is either some scalar type s' or some set type $\{s'\}$. The next lemma takes care of the first case.

Lemma 5.15 *Suppose all functions in Σ have set height 0, and let $f : t \times s \to s'$ be some function in $\mathcal{NRA}(\Sigma) + bfix$, where s, s' are scalar types and t is a PS-type (i.e. a "product of sets" type, see Subsection 3.1). Then there is some function $f_0 : s \to s'$ in $\mathcal{NRA}_0(\Sigma)$ such that $f = f_0 \circ \pi_2$.*

Proof. The lemma essentially says that the only way we can obtain some scalar value is from other scalar values. Indeed, no function construct in $\mathcal{NRA}(\Sigma) + bfix$ returns a scalar type, except the functions in Σ, and projections and pairing of scalar values. The formal proof is done straightforwardly, by induction on the structure of f. $\Box$

Finally we can prove the conservativity result of Theorem 4.1:

Theorem 4.1 *If all function symbols in Σ have set height ≤ 1, then $\mathcal{NRA}(\Sigma) + bfix$ is a conservative extension of $\mathcal{NRA}_1(\Sigma) + bfix$, for both inflationary and partial fixpoint semantics. The latter is a conservative extension of the relational algebra with bounded fixpoints $\mathcal{RA}(\Sigma) + bfix$.*

Proof. We start with the case when all functions in Σ have set height 0, and let f be some function in $\mathcal{NRA}(\Sigma) + bfix$, whose input and output types have set height ≤ 1. Without loss of generality, it suffices to consider two cases: (1) $f : t \times s \to s'$, and (2) $f : t \times s \to \{s'\}$ where s, s' are scalar types and t is a flat type. Lemma 5.15 proves that, in the first case f can be expressed in $\mathcal{NRA}_1(\Sigma)$. For the second case, define $g = ext_2(f)$, $g : t \times \{s\} \to \{s'\}$. Now g has flat types as input and output types, hence, by Proposition 5.14 we may assume that g is in $\mathcal{RA}(\Sigma) + bfix$, hence in $\mathcal{NRA}_1(\Sigma) + bfix$. Finally, observe that $f(x, y) = g(x, \{y\})$, so f can be expressed in $\mathcal{NRA}_1(\Sigma) + bfix$.

Now we describe how the techniques developed for the case when the functions in Σ have set height 0 extend to the case when they have set height ≤ 1. Without loss of generality we may assume that Σ contains two new kinds of function. Here s, s' are scalar types, t is a flat type.

Aggregate Functions having type $p : t \times s \to s'$. For example, $average : \{\mathbb{N}\} \to \mathbb{N}$ is an aggregate function (where $t = \{\mathbb{N}\}$ and $s = unit$).

Set Constructors having some type $p : t \times s \to \{s'\}$. For example, $gen : \mathbb{N} \to \{\mathbb{N}\}$, $gen(n) \stackrel{\text{def}}{=} \{1, 2, \ldots, n\}$.

The we extend the techniques described so far, as follows:

1. We extend the definition of $\mathcal{RA}(\Sigma) + bfix$. Namely instead of function of the form $map(\varphi) : \{s\} \to \{s'\}$ with φ in $\mathcal{NRA}_0(\Sigma)$ (see Item 4 in the definition of $\mathcal{RA}(\Sigma)$), we include in $\mathcal{RA}$ all functions of the form $ext_2(\varphi) : t \times \{s\} \to \{s'\}$, where $\varphi : t \times s \to \{s'\}$ is in $\mathcal{NRA}_1(\Sigma)$.

2. Extend both the Strict Map Lemma (Lemma 5.4) and the Map Lemma to the new definition of $\mathcal{RA}(\Sigma)+$ *bfix*. We only have to show how the case $ext_2(\varphi) : t \times \{s\} \to \{s'\}$ is handled. To compute $g(\rho, \rho')$, where $g = \overline{map}(ext_2(\varphi))$, $g : [I \to t] \times \{I \times s\} \to \{I \times s'\}$, first define $\psi : ([I \to t] \times \{I \times s\}) \times I \to \{I \times s'\}$ as follows: $\psi((\rho, \rho'), i) = \{i\} \bowtie ext_2(\varphi)(\rho(i), \rho'(i))$. Obviously ψ is in $\mathcal{NRA}_1(\Sigma) + bfix$, since φ is there. Next, consider $ext_2(\psi) : ([I \to t] \times \{I \times s\}) \times \{I\} \to \{I \times s'\}$. Then $g(\varrho, \varrho') = ext_2(\psi)((\rho, \rho'), Dom(\rho'))$.

3. Extend the flattening Proposition 5.10 and 5.7. Here we only have to show how to compute $FLAT(p)$, when p is (1) some aggregate function or (2) some set constructor. The first case is trivial, but for the second case, when $p : t \times s \to \{s'\}$, we need some additional structure on I. Namely we assume to have some index generators $index\text{-}gen_{s'} : \{s'\} \to \{I \times s'\}$ with the property $\forall x. x \cong index\text{-}gen_{s'}(x)$, for every type s' which is the codomain of some set constructor. Intuitively $index\text{-}gen_{s'}(\{x_1, \ldots, x_n\}) = \{(i_1, x_1), \ldots, (i_n, x_n)\}$, where $i_1, \ldots, i_n$ are distinct. Then $FLAT(p)$ is expressed in $\mathcal{RA}(\Sigma \cup I)$ with the help of $index\text{-}gen_{s'}$.

4. We extend the tagging constructs of Subsection 5.3, by considering also tags $\tau = p$, for every set constructor $p \in \Sigma$, and by taking $I_p \stackrel{\text{def}}{=} s'$, where $\{s'\}$ is the codomain of p. A tagging of the expression $index\text{-}gen_{s'}$ will be $\overline{index\text{-}gen_{s'}} : \{s'\} \to \{I_p \times s'\}$. Finally we extend the tagging Lemma 5.13; while in Lemma 5.13 the case $g = map(\varphi)$ didn't deserve too much attention, now the case $g = ext_2(\varphi)$ becomes more complicated, and we need to prove a tagging lemma for $\mathcal{NRA}_1(\Sigma) + bfix$ first. This is lengthly, but straightforward, and is omitted.

5. The above suffices to prove that $\mathcal{NRA}(\Sigma) + bfix$ is a conservative extension of $\mathcal{RA}(\Sigma) + bfix$. To prove that it is also a conservative extension of $\mathcal{NRA}_1(\Sigma) + bfix$, we need to extend Lemma 5.15 to the following. If $f : t \times s \to s'$ is some function in $\mathcal{NRA}(\Sigma) + bfix$, where t is a PS-type and s, s' are scalar types, then there is a function $h : t \times s \to t'$ in $\mathcal{NRA}(\Sigma) + bfix$, where t' is a flat relation type, and there is some function $g : t' \times s \to s'$ in $\mathcal{NRA}_1(\Sigma)$, such that $f(x, y) = g(h(x, y), y)$. In other words, we can obtain some scalar value only by making some arbitrary computation (h), obtain some flat relation (of type t'), and then by applying some "simple" function g, i.e. $\mathcal{NRA}_1(\Sigma)$, which returns the scalar. That is the last part of the computation of f obtains the scalar value from some flat relation. This is proven straightforwardly by induction on the structure of f. Now we translate h into h', with h' in $\mathcal{RA}(\Sigma) + bfix$, which is possible because its output type is a flat relation type, then conclude that $g(h'(x, y), y)$ is in $\mathcal{NRA}_1(\Sigma) + bfix$, because both g and h' are in this language.

□

6 A Logic-Based Language

In this section we present a logical calculus for complex objects, with fixpoints, $\mathcal{NRCALC}(\Sigma) + fix$, and define a range-restricted fragment of it, which we prove to be equivalent to $\mathcal{NRA}(\Sigma) + bfix$. $\mathcal{NRCALC}(\Sigma) + bfix$ is in the style of other logic-based languages for complex objects (e.g. [AB88, GV91, HS91]), but its syntax is kept closer to the algebraic language, to simplify the proof of the conversions. Its main novelty consists in the range-restriction rules, which are designed such as (1) to simplify the proofs of the closure properties (under composition, *map*, bounded fixpoints, etc) of the language, and (2) to make the bounding of fixpoints explicit. The range-restriction rules can accommodate external functions.

Grumbach and Vianu in [GV91] consider a logic-based language for complex objects, RR-CALC+IFP, which they prove to be contained in *PTIME*. The relationship between RR-CALC+IFP and $\mathcal{NRCALC}(\Sigma) + fix$ remains open.

We consider a signature Σ given, as for $\mathcal{NRA}(\Sigma)$. For every type s there is exactly one **input variable** x^s, and a denumerable set of **variables** $u^s, v^s, \ldots$. The terms (e) and formulas (φ) are typed, and are defined inductively:

terms The following are terms: () (the empty tuple), x^s, $u^s, v^s, w^s, \ldots$ (variables), (e_1, e_2), $p(e)$ (where $p \in \Sigma$ is a function symbol), $\pi_i(e)$ (the i's projection, $i = 1, 2$), $\{u^t \mid \varphi\}$ where u^t is a variable and φ is some formula (the type of this term is $\{t\}$). u^t becomes bound in $\{u^t \mid \varphi\}$.

formulas The following are formulas: $true$, $false$, $e_1 \in e_2$, $e_1 = e_2$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$ (where φ, ψ are formulas and e_1, e_2 are terms), $\exists u^s \in e.\varphi$ (where u^s is a variable, e is a term of type $\{s\}$, and φ is some formula). We abbreviate $\forall u^s \in e.\varphi$ for $\neg\exists u^s \in e.\neg\varphi$.

We extend the language with a fixpoint construction:

fixpoint $\mu w^{\{t\}}.\{u^t \mid \varphi\}$ is a term of type $\{t\}$, where $w^{\{t\}}$ and u^t are variables, and φ is a formula. Both $w^{\{t\}}$ and u^t become bound in $\mu w^{\{t\}}.\{u^t \mid \varphi\}$.

We shall assume that all bounded variables in some term e or formula φ are distinct, and distinct from the free variables; by renaming the bound variables, we can bring any term or formula to such a form. Also, we shall omit the type superscripts of the variables, when they are clear from the context.

A **query** of type $s \to t$ in $\mathcal{NRCALC}(\Sigma)$ or $\mathcal{NRCALC}(\Sigma) + \mathit{fix}$, is simply a term of type t, having only one free variable, namely x^s, the input variable of type s. Note that, although we can write some purely algebraic queries, like $(p(\pi_1(x), \pi_2(x)), p'(\pi_2(x)))$, for "real" database queries one has to make use of formulas, like in $\mathit{unnest} : \{s \times \{t\}\} \to \{s \times t\}$: $\mathit{unnest} \stackrel{\text{def}}{=} \{u \mid \exists v \in x.\exists w \in \pi_2(v).u = (\pi_1(v), w)\}$, or in the following *nest* query, $\{s \times t\} \to \{s \times \{t\}\}$, $\mathit{nest} \stackrel{\text{def}}{=} \{u \mid \exists v \in x.u = (\pi_1(v), \{w \mid (\pi_1(v), w) \in x\})\}$.

The values of terms and formulas depend on the values assigned to their free variables. Formulas can be $true$, $false$, or undefined. $\exists u \in e.\varphi$, is interpreted as follows: it is defined iff e is defined, and for all $u \in e$, φ is defined; in that case, $\exists u \in e.\varphi = true$ iff there is some $u \in e$ such that φ is true. The interpretation of the term $\{u \mid \varphi\}$ is the set of all values of u for which φ is defined and true; it is undefined, if this set is infinite[5]. For the fixpoint construction, we consider both inflationary and partial interpretations.

To avoid exponential queries, like the powerset of $x^{\{s\}}$, $\{u^{\{s\}} \mid \forall v^s \in u^{\{s\}}.v^s \in x^{\{s\}}\}$, or domain dependent queries, like $\{u^s \mid \neg u^s \in x^{\{s\}}\}$, we restrict the language, to the *range-restricted* sub-language $(\mathcal{NRCALC}(\Sigma) + \mathit{bfix})^{rr}$. For this, let M be a term or a formula: we define certain sub-term occurrences e to be range-restricted in M (and then we underline e: $\underline{e}$), and certain variables v to be range-restricted in a sub-term or sub-formula occurrences N of M (and then we mark N by: $\underbrace{N}_{v}$). The following rules define the range-restriction. Recall that the range-restriction is a property of a sub-term or sub-formula occurrence N, of some term or formula M.

[5]This will be impossible in the range-restricted version of $\mathcal{NRCALC}$, to be defined below.

Axiom

$$\frac{}{\underline{x}}\ (x \text{ is the input variable})$$

Variables Downwards

$$\frac{\underbrace{(e_1, e_2)}_{u}}{(\underbrace{e_1}_{u}, \underbrace{e_2}_{u})} \qquad \frac{\underbrace{p(e_1, \ldots, e_m)}_{u}}{p(\underbrace{e_1}_{u}, \ldots, \underbrace{e_m}_{u})} \qquad \frac{\underbrace{\pi_i(e)}_{u}}{\pi_i(\underbrace{e}_{u})}\ (i = 1, 2) \qquad \frac{\underbrace{e_1 \in e_2}_{u}}{\underbrace{e_1}_{u} \in \underbrace{e_2}_{u}} \qquad \frac{\underbrace{e_1 = e_2}_{u}}{\underbrace{e_1}_{u} = \underbrace{e_2}_{u}}$$

$$\frac{\underbrace{\varphi \wedge \psi}_{u}}{\underbrace{\varphi}_{u} \wedge \underbrace{\psi}_{u}} \qquad \frac{\underbrace{\varphi \vee \psi}_{u}}{\underbrace{\varphi}_{u} \vee \underbrace{\psi}_{u}} \qquad \frac{\underbrace{\neg\varphi}_{u}}{\neg \underbrace{\varphi}_{u}} \qquad \frac{\underbrace{\exists u \in e.\varphi}_{v}}{\exists u \in e.\ \underbrace{\varphi}_{v}} \qquad \frac{\underbrace{\{u \mid \varphi\}}_{v}}{\{u \mid \underbrace{\varphi}_{v}\}} \qquad \frac{\underbrace{\mu w.\{u \mid \varphi\}}_{v}}{\mu w.\{u \mid \underbrace{\varphi}_{v}\}}$$

Variables to Terms

$$\frac{\underbrace{u}_{u}}{\underline{u}} \qquad \frac{\{u \mid \underbrace{\varphi}_{u}\}}{\underline{\{u \mid \varphi\}}} \qquad \frac{\mu w.\{u \mid \underbrace{\varphi}_{u}\}}{\underline{\mu w.\{u \mid \varphi\}}}$$

Terms Upwards

$$\frac{(\underline{e_1}, \underline{e_2})}{\underline{(e_1, e_2)}} \qquad \frac{p(\underline{e_1}, \ldots, \underline{e_m})}{\underline{p(e_1, \ldots, e_m)}} \qquad \frac{\pi_i(\underline{e})}{\underline{\pi_i(e)}}\ (i = 1, 2) \qquad \frac{}{\underline{()}}$$

Horizontal Rules

$$\frac{\underline{e_1} = e_2}{\underline{e_1} = \underline{e_2}} \qquad \frac{e_1 = \underline{e_2}}{\underline{e_1} = \underline{e_2}} \qquad \frac{e_1 \in \underline{e_2}}{\underline{e_1} \in \underline{e_2}}$$

Terms Downwards

$$\frac{\underline{(e_1, e_2)}}{(\underline{e_1}, \underline{e_2})}$$

Terms to Variables

$$\frac{\underline{u}}{\underbrace{u}_{u}} \qquad \frac{\exists u \in \underline{e}.\varphi}{\exists u \in \underline{e}.\ \underbrace{\varphi}_{u}}$$

Variable Upwards

$$\frac{(e_1, \underbrace{e_2}_{u})}{\underbrace{(e_1, e_2)}_{u}} \quad \frac{(\underbrace{e_1}_{u}, e_2)}{\underbrace{(e_1, e_2)}_{u}} \quad \frac{p(e_1, \ldots, \underbrace{e_i}_{u}, \ldots, e_m)}{\underbrace{p(e_1, \ldots, e_m)}_{u}} \quad \frac{\pi_i(\underbrace{e}_{u})}{\underbrace{\pi_i(e)}_{u}} \quad \frac{\underbrace{e_1}_{u} \in e_2}{\underbrace{e_1 \in e_2}_{u}} \quad \frac{\underbrace{e_1}_{u} = e_2}{\underbrace{e_1 = e_2}_{u}}$$

$$\frac{e_1 = \underbrace{e_2}_{u}}{\underbrace{e_1 = e_2}_{u}} \quad \frac{}{\underbrace{false}_{u}} \quad \frac{\underbrace{\varphi}_{u} \wedge \psi}{\underbrace{\varphi \wedge \psi}_{u}} \quad \frac{\varphi \wedge \underbrace{\psi}_{u}}{\underbrace{\varphi \wedge \psi}_{u}} \quad \frac{\underbrace{\varphi}_{u} \vee \underbrace{\psi}_{u}}{\underbrace{\varphi \vee \psi}_{u}} \quad \frac{\exists u \in t. \underbrace{\varphi}_{v}}{\underbrace{\exists u \in t. \varphi}_{v}}$$

A formula φ is *range-restricted* if all sub-terms occurring as bounds for quantifiers are range-restricted (i.e. the quantifiers are of the form $\exists u \in \underline{t}.\varphi$). A term e is **range-restricted** if it satisfies the same condition, and, in addition, $\underline{e}$ is provable. Similarly, the **range-restricted** fragment $(\mathcal{NRCALC}(\Sigma) + \mathit{fix})^{rr}$ of $\mathcal{NRCALC}(\Sigma) + \mathit{fix}$ is defined to contain only the range-restricted queries.

The intuition is that u becomes range-restricted in some sub-formula φ if we can compute the set of all values u which will make φ true. Range restriction is then exported to other sub-formulas. For example when u is range-restricted in the sub-formula φ of $\varphi \wedge \psi$, then it is range-restricted also in ψ: intuitively, when we "compute" ψ, there is no sense in assigning to u other values than those bounded in φ. But a similar property does *not* hold for $\vee$: if φ binds u, this does not limit our search space when we try to satisfy $\varphi \vee \psi$. As a second example, note that there is no *variable upwards* rule for $\neg\varphi$, $\{v \mid \varphi\}$ and $\mu w.\{v \mid \varphi\}$: the intuition is that, if have a bound for the set of values of u which make φ *true*, this does not give us any bound for the values of u making $\neg\varphi$ true. The case $\{v \mid \varphi\}$ is more subtle. If u becomes bound in φ, it means that when we try to satisfy $\varphi(u, v)$ we have a bound for the values u may take. But when we compute the *term* $\{v \mid \varphi\}$, this bound is vacuous: we may still assign to u any value outside this bound, and get as the result $\emptyset$. Finally, note the rule for the fixpoint (μ): in order to show that $\mu w.\{u \mid \varphi\}$ is range-restricted, $\underline{\mu w.\{u \mid \varphi\}}$, we must be able to prove that u is range-restricted in φ, without assuming anything about the fixpoint variable w (which, in fact, may not be range-restricted). This is a different approach than in [GV91].

Under this proof system, some sub-formula φ acts as a function from sets of variables to sets of variables: from the assertion that some variables u are range-restricted in φ, i.e. $\underbrace{\varphi}_{u}$, we prove that other variables, say w are range-restricted in φ, in a sequence of upwards and downwards steps, as described below:

Input From a context outside φ, some variables u, v, etc., are asserted to be range-restricted in φ: $\underbrace{\varphi}_{u}, \underbrace{\varphi}_{v}, \ldots$. At least the input variable, x, is asserted to be range-restricted.

Downwards For each such variable u, we push the assertion $\underbrace{\varphi}_{u}$ downwards, to the sub-formulas and sub-terms of φ, until we reach sub-terms which are u itself: this sub-term becomes range-restricted, $\underline{u}$.

Upwards Starting from small, range-restricted terms, we prove that larger terms are range-restricted.

Horizontally Range-restriction is transmitted "horizontally" to other terms, in formulas like $e_1 \in \underline{e_2}$ and $e_1 = \underline{e_2}$.

Downwards Range-restriction of terms is again transmitted downwards, to terms until we prove that new variables, say w, are range-restricted.

Upwards We prove $\underbrace{M}_{w}$ upwards, for larger sub-terms and sub-formulas M.

Output We have proven $\underbrace{\varphi}_{w}$, for new variables w.

For terms e, we start with some assertions $\underbrace{e}_{u}$, and only prove that the term is range-restricted $\underline{e}$: terms do not allow us to conclude that other variables are range-restricted ($\underbrace{e}_{w}$).

Example 6.1 *nest defined above is range-restricted. The proof steps are the following:*

$$\{u \mid \exists v \in \underline{x}.u = (\pi_1(v), \{w \mid (\pi_1(v), w) \in \underline{x}\})\} \qquad \textit{(axiom)}$$

$$\{u \mid \exists v \in x.\underbrace{u = (\pi_1(v), \{w \mid (\pi_1(v), w) \in \underline{x}\})}_{v}\} \qquad \textit{(terms to variables)}$$

$$\{u \mid \exists v \in x.u = \underbrace{(\pi_1(v), \{w \mid (\pi_1(v), w) \in \underline{x}\})}_{v}\} \qquad \textit{(variables downwards)}$$

$$\{u \mid \exists v \in x.u = (\underbrace{\pi_1(v)}_{v}, \{w \mid (\pi_1(v), w) \in \underline{x}\})\} \qquad \textit{(variables downwards)}$$

$$\{u \mid \exists v \in x.u = (\pi_1(\underbrace{v}_{v}), \{w \mid (\pi_1(v), w) \in \underline{x}\})\} \qquad \textit{(variables downwards)}$$

$$\{u \mid \exists v \in x.u = (\pi_1(\underline{v}), \{w \mid (\pi_1(v), w) \in \underline{x}\})\} \qquad \textit{(variables to terms)}$$

$$\{u \mid \exists v \in x.u = (\underline{\pi_1(v)}, \{w \mid \underline{(\pi_1(v), w)} \in x\})\} \qquad \textit{(term upwards, horizontal rule)}$$

$$\{u \mid \exists v \in x.u = (\underline{\pi_1(v)}, \{w \mid (\pi_1(v), \underline{w}) \in x\})\} \qquad \textit{(terms downwards)}$$

$$\{u \mid \exists v \in x.u = (\underline{\pi_1(v)}, \{w \mid (\pi_1(v), \underbrace{w}_{w}) \in x\})\} \qquad \textit{(terms to variables)}$$

$$\{u \mid \exists v \in x.u = (\underline{\pi_1(v)}, \{w \mid \underbrace{(\pi_1(v), w) \in x}_{w}\})\} \qquad \textit{(variables upwards)}$$

$$\{u \mid \exists v \in x.u = (\underline{\pi_1(v)}, \underline{\{w \mid (\pi_1(v), w) \in x\}})\} \qquad \textit{(variables to terms)}$$

$$\{u \mid \exists v \in x.\underline{u} = (\pi_1(v), \{w \mid (\pi_1(v), w) \in x\})\} \qquad \textit{(terms upwards)}$$

$$\{u \mid \exists v \in x.\underbrace{u}_{u} = (\pi_1(v), \{w \mid (\pi_1(v), w) \in x\})\} \qquad \textit{(terms to variables)}$$

$$\{u \mid \underbrace{\exists v \in x.u = (\pi_1(v), \{w \mid (\pi_1(v), w) \in x\})}_{u}\} \qquad \textit{(variables upwards)}$$

$$\underline{\{u \mid \exists v \in x.u = (\pi_1(v), \{w \mid (\pi_1(v), w) \in x\})\}} \qquad \textit{(variables to terms)}$$

Hence we have proven $\underline{nest}$. Moreover, it has a single quantifier, $(\exists v \in x)$, and its bound is range-restricted too. This proves that nest is range-restricted.

Theorem 6.2 *$(\mathcal{NRCALC}(\Sigma) + \mathit{fix})^{rr}$ and $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ have the same expressive power.*

Proof.

1. $\mathcal{NRA}(\Sigma) + \mathit{bfix} \subseteq (\mathcal{NRCALC}(\Sigma) + \mathit{fix})^{rr}$. This is done straightforward, by converting any function $f : s \to t$ from $\mathcal{NRA}(\Sigma) + \mathit{bfix}$, into some term e_f of type t, with input variable of type s. We present the most significant cases.

   $\eta : s \to \{s\}$ is translated to $e_\eta := \{u^s \ \mid u^s = x^s\}$.

   $\mu : \{\{s\}\} \to \{s\}$ is translated to $e_\mu := \{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.u^s \in v^{\{s\}}\}$. This term *is* range-restricted, by the following judgment (we keep the type superscripts for the variables in this example, to make it more formal):

$$
\begin{gathered}
\{u^s \ \mid \exists v^{\{s\}} \in \underline{x^{\{\{s\}\}}}.u^s \in v^{\{s\}}\} \\
\{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.\underbrace{u^s \in v^{\{s\}}}_{v^{\{s\}}}\} \\
\{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.u^s \in \underbrace{v^{\{s\}}}_{v^{\{s\}}}\} \\
\{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.u^s \in \underline{v^{\{s\}}}\} \\
\{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.\underline{u^s} \in \underline{v^{\{s\}}}\} \\
\{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.\underbrace{u^s \in v^{\{s\}}}_{u^s}\} \\
\{u^s \ \mid \underbrace{\exists v^{\{s\}} \in x^{\{\{s\}\}}.u^s \in v^{\{s\}}}_{u^s}\} \\
\underline{\{u^s \ \mid \exists v^{\{s\}} \in x^{\{\{s\}\}}.u^s \in v^{\{s\}}\}}
\end{gathered}
$$

   $\mathit{map}(f) : \{s\} \to \{t\}$ is translated to $e_{\mathit{map}(f)} := \{u^s \ \mid \exists v^s \in x^{\{s\}}.u^s = e_f[v^s/x^s]\}$. By induction hypothesis, there is a range-restriction proof of e_f starting from the hypothesis $\underbrace{e_f}_{x^{\{s\}}}$. So we can prove that $e_f[v^s/x^s]$ is range-restricted, starting from the hypothesis $\underbrace{e_f[v^s/x^s]}_{v^s}$, which is easy to prove from the second *Terms-to-Variables* rule.

   $\cup : \{s\} \times \{s\} \to \{s\}$ is translated to $e_\cup := \{u \ \mid u \in \pi_1(x) \vee u \in \pi_2(x)\}$

   $\mathit{not} : \{\mathit{unit}\} \to \{\mathit{unit}\}$ is translated to $e_{\mathit{not}} := \{u \ \mid u = () \wedge \neg\exists u \in x.\mathit{true}\}$.

   $\mathit{eq} : b \times b \to \{\mathit{unit}\}$ is translated to $e_{\mathit{eq}} := \{u \ \mid u = () \wedge \pi_1(x) = \pi_2(x)\}$.

   The bounded fixpoint of $f : s \times \{t\} \to \{t\}$ and $g : s \to \{t\}$, $\mathit{bfix}(f, g) : s \to \{t\}$, is translated into $e_{\mathit{bfix}(f,g)} := \mu w^{\{t\}}.\{u^t \ \mid u^t \in e_g \wedge u^t \in e_f[(x^s, \{v^t \mid v^t \in e_g \wedge v^t \in w^{\{t\}}\})/x^{s\times\{t\}}]\}$. To prove that this term is range-restricted, start with the observation that e_g is range-restricted, and that x is always range-restricted:

$$\mu w.\{u \mid u \in \underline{e_g} \wedge u \in e_f[(\underline{x}, \{v \mid v \in \underline{e_g} \wedge v \in w\})/x]\}$$

$$\mu w.\{u \mid u \in \underline{e_g} \wedge u \in e_f[(\underline{x}, \{v \mid \underline{v} \in \underline{e_g} \wedge v \in w\})/x]\}$$

$$\mu w.\{u \mid u \in \underline{e_g} \wedge u \in e_f[(\underline{x}, \{v \mid \underbrace{v \in \underline{e_g}}_{v} \wedge v \in w\})/x]\}$$

$$\mu w.\{u \mid u \in \underline{e_g} \wedge u \in e_f[(\underline{x}, \{v \mid \underbrace{v \in e_g \wedge v \in w}_{v}\})/x]\}$$

$$\mu w.\{u \mid u \in \underline{e_g} \wedge u \in e_f[(\underline{x}, \underline{\{v \mid v \in e_g \wedge v \in w\}})/x]\}$$

$$\mu w.\{u \mid u \in \underline{e_g} \wedge u \in e_f[\underline{(x, \{v \mid v \in e_g \wedge v \in w\})}/x]\}$$

To prove that all quantifiers are bounded by range-restricted terms in $e_f[(x,u)/x]$, proceed as in the proof of $\underline{e_f}$, but replacing the axiom $\underline{x}$, with the derived fact $\underline{(x, \{v \mid v \in e_g \wedge v \in w\})}$.

2. $(\mathcal{NRCALC}(\Sigma) + \mathit{fix})^{rr} \subseteq \mathcal{NRA}(\Sigma) + \mathit{bfix}$. Subexpressions of some query in $(\mathcal{NRCALC}(\Sigma) + \mathit{fix})^{rr}$ are not necessarily queries themselves, so we need to prove a stronger assertion for terms with several free variables, and for formulas as well. Consider some term e of type t, or some formula e, with free variable(s) of type s (i.e. s is the product of the types of all free variables). Let v^s stand for the tuple of all free variables. Assume the variables in v^s to be split as $v^s = ((v')^{s'}, (v'')^{s''})$, where s' and s'' are the types of the first and second group respectively (hence $s = s' \times s''$). The splitting is done such that e (or φ) can be proven to be range-restricted under the assumption that the variables $(v')^{s'}$ are range-restricted. Then we translated e into a function $f_e : s \to t$, and φ into some predicate $P_\varphi : s \to \{\mathit{unit}\}$ in $\mathcal{NRA}(\Sigma) + \mathit{bfix}$, by induction on the structure of e and φ. All cases are straightforward, with two exceptions: the term constructions $e = \{u^r \mid \psi\}$ and $e = \mu w^{\{r\}}.\{u^r \mid \psi\}$. Consider first $e = \{u^r \mid \psi\}$. By induction, ψ translates to some predicate $P_\psi : s \times r \to \{\mathit{unit}\}$. We show below how to derive from the proof of the fact that $\{u^r \mid \psi\}$ is range-restricted a *bounding function* $b_u : s' \to \{r\}$, such that for any values of the variable u^r making ψ true, $u^r \in b_u((v')^{s'})$ holds. Then, the translation of the term $e = \{u^r \mid \psi\}$ is $f_e(v^s) = \sigma_{P_\psi}(b_u((v')^{s'}))$, where σ_{P_ψ} is the *selection* associated to the predicate P_ψ. For the term $e = \mu w^{\{r\}}.\{u^r \mid \psi\}$, ψ has both $w^{\{r\}}$ and u^r among its free variables, and we may view the term $e' = \{u^r \mid \psi\}$ as a query of type $s \times \{r\} \to \{r\}$ (some renaming of variables is needed for that). Since e' has less fixpoints than e, we assume to have some translation of e' into $\mathcal{NRA}(\Sigma) + \mathit{bfix}$, $f_{e'}$. As before, we prove that, if ψ can be proven to be range-restricted from the assumption that the variables $(v')^{s'}$ are range-restricted, then whenever ψ is true, it must be the case that $u^r \in b_u((v')^{s'})$, for some bounding function b_u. Now it is important to notice that $w^{\{r\}}$ is *not assumed* to be range-restricted (see the rules for the fixpoint), hence b_u does not depend on $w^{\{r\}}$, and can be viewed as a function $b_u : s \to \{r\}$. So, in $\mathcal{NRA}$ notation, we take $f_e := \mathit{bfix}(f_{e'}, b_u)$.

So it remains to prove the existence of the bounding functions b_u. Recall that some formula φ is proven to be range-restricted under the assumption that some variables $(v')^{s'}$ (of type s') are range-restricted. Here is the place where we do not do induction by the structure of terms or formulas, but by the length of the proof of the range-restriction. Namely, by induction on the length of the judgments of range-restriction, we derive two family of functions: $b_u^\varphi : s' \to \{r\}$ for every variable proven range-restricted in φ ($\underbrace{\varphi}_{u}$; r is the type of u), and $c_e^\varphi : s' \to \{r\}$ for each sub-term proven range-restricted

($\underline{e}$; r is the type of e), with the following properties: (1) whenever φ is true, $u \in b_u^\varphi((v')^{s'})$ holds for any range-restricted variable u, and (2) $e \in c_e^\varphi((v')^{s'})$ holds for any range-restricted sub-term.

We construct similar functions b_u^e and c_u^e for range-restricted terms e. The functions are derived straightforwardly, according to the range-restriction rules. We only illustrate two *variable-upwards* rules:

- $\dfrac{\underbrace{\varphi}_{u} \wedge \psi}{\underbrace{\varphi \wedge \psi}_{u}}$. Here take $b_u^{\varphi \wedge \psi} = b_u^\varphi$, and argue that whenever $\varphi \wedge \psi$ is true, φ must be true as well, so by induction $u \in b_u^\varphi$, hence by this definition $u \in b_u^{\varphi \wedge \psi}$.
- $\dfrac{\underbrace{\varphi}_{u} \vee \underbrace{\psi}_{v}}{\underbrace{\varphi \vee \psi}_{u}}$. Take $b_u^{\varphi \vee \psi} = b_u^\varphi \cup b_u^\psi$, and argue that, whenever $\varphi \vee \psi$ is true, either φ or ψ must be true, so one of $u \in b_u^\varphi(v')$ or $u \in b_u^\psi(v')$ must hold.

$\square$

7 Concluding remarks and further research

We have investigated the power of the nested relational algebra enriched with a bounded fixpoint. The language turned out to be still of polynomial time or space complexity - according to the interpretation of the fixpoint: inflationary or partial - and it can express all *PTIME* or *PSPACE* queries over ordered databases. The main result consists in proving that, over flat relations, the language has the same expressive power as the first-order logic with inflationary or partial fixpoints. We also define a logical calculus, having the same expressive power.

The method used in the proof of the conservativity result could be used as a compilation technique, for encoding complex objects by flat relations, and for translating queries over complex objects into queries over flat relations with indexes. The complexity of this translation however remains to be investigated: e.g. does it guarantee that a query of time complexity $O(n^2)$ over complex objects is translated into a query over flat relations with the same complexity ?

Finally, we believe that our methods can be extended to show that, for any $k \geq 1$, $\mathcal{NRA}(\Sigma) + \mathit{bfix}$ is a conservative extension of $\mathcal{NRA}_k(\Sigma) + \mathit{bfix}$, provided that all functions in Σ have set height $\leq k$.

8 Acknowledgments

I am in debt to Val Tannen for numerous discussions and comments, to Peter Buneman for his suggestions and encouragement, and to Catriel Beeri for his careful reading of an earlier version of this paper. I want to thank Victor Vianu for explaining me how to encode simultaneous fixpoints with only one fixpoint: the

proof of Proposition 3.9 is a generalization of his explanations. I also want to thank the anonymous reviewer, for his particularly clear and knowledgeable comments.

References

[AB86] S. Abiteboul and N. Bidoit. Non-first normal form relations: An algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33(3):361–371, 1986.

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

[AGVW89] 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.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley Publishing Co, 1995.

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

[AV91a] Serge Abiteboul and Victor Vianu. Expressive power of query languages. In J. Ullman, editor, *Theoretical Studies in Computer Science.* Academic Press, 1991.

[AV91b] Serge Abiteboul and Victor Vianu. Generic computation and its complexity. In *Proceedings of 23rd ACM Symposium on the Theory of Computing*, 1991.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of proramming with collection types. *Theoretical Computer Science*, 1995. To appear.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages, Naphlion, Greece*, pages 9–19. Morgan Kaufmann, August 1991. Also available as UPenn Technical Report MS-CIS-92-17.

[CH82] Ashok Chandra and David Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[dB92] Jan Van den Bussche. Complex object manipulation through identifiers: An algebraic perspective. Technical Report 92-41, University of Antwerp, Department of Mathematics and Computer Science, Universiteitsplein 1, B-2610 Antwerp, Belgium, September 1992.

[GF88] Dirk Van Gucht and Patrick C. Fischer. Multilevel nested relational structures. *Journal of Computer and System Sciences*, 36:77–105, 1988.

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

[GG92] 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.

[GS86] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32:265–180, 1986.

[GSG95] Marc Gyssens, Dan Suciu, and Dirk Van Gucht. On polynomially bounded fixpoint constructs for nested relations. In *Proceedings of DBPL'95*, Gubbio, Italy, September 1995.

[GV91] Stephane Grumbach and Victor Vianu. Tractable query languages for complex object databases. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, 1991.

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

[HS91] Richard Hull and Jianwen Su. On the expressive power of database queries with intermediate types. *Journal of Computer and System Sciences*, 43:219–267, 1991.

[Imm86] Neil Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[PG88] Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *Proceedings of 7th ACM Symposium on Principles of Database Systems,Austin, Texas*, pages 29–38, 1988.

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

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[ST] Dan Suciu and Val Tannen. A query language for NC. To appear in Journal of Computer and Systems Sciences. Preliminary version appeared in the Proceedings of the 13th ACM Symposium on Principles of Database Systems, 1994.

[Suc] Dan Suciu. Domain-independent queries on databases with external functions. To appear in Journal of Theoretical Computer Science. Preliminary version appeared in the Proceedings of the Fifth International Conference on Database Theory, 1995.

[TF86] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis and F. P. Preparata, editors, *Advances in Computing Research: The Theory of Databases*, pages 269–307, London, England, 1986. JAI Press.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems II: The New Technologies*. Computer Science Press, Rockvill, MD 20850, 1989.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of 14th ACM SIGACT Symposium on the Theory of Computing*, pages 137–146, San Francisco, California, 1982.

[Won93] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proceedings of 12th ACM Symposium on Principles of Database Systems*, pages 26–36, Washington, D. C., May 1993. See also UPenn Technical Report MS-CIS-92-59.