# The Restricted and Bounded Fixpoint Closures of the Nested Algebra are Equivalent


Marc Gyssens
Dept. WNI, University of Limburg
B-3590 Diepenbeek, Belgium
gyssens@charlie.luc.ac.be

Dan Suciu
AT&T Bell Laboratories
Murray Hill, NJ 07974, USA
suciu@research.att.com

Dirk Van Gucht
Comp. Sci. Dept., Indiana University
Bloomington, IN 47405-4101, USA
vgucht@cs.indiana.edu



**Abstract**

The nested model is an extension of the traditional, "flat" relational model in which relations can also have relation-valued entries. Its "default" query language, the nested algebra, is rather weak, unfortunately, since it is only a conservative extension of the traditional, "flat" relational algebra, and thus can only express a small fraction of the polynomial-time queries. Therefore, it was proposed to extend the nested algebra with a least-fixpoint construct, but the resulting language turned out to be too powerful: many inherently exponential queries could also be expressed. Two polynomial-time restrictions of the least-fixpoint closure of the nested algebra were proposed: the *restricted least-fixpoint closure* (by Gyssens and Van Gucht) and the *bounded fixpoint closure* (by Suciu). Here, we prove that both restrictions are equivalent in expressive power. We also exhibit a proof technique, called type substitution, by which we reduce our result to its obvious counterpart in the "flat" relational model; thus emphasizing the inherent weakness of the nested algebra.


## 1 Introduction

The nested model [12, 16] is an extension of the traditional, "flat" relational database model in which relations can have both "flat," atomic entries and structured, relation-valued entries. Since the late 1980s, various query languages have been considered in the context of the nested model [1, 5, 7, 9, 11, 14, 16]. These languages can be classified according to their expressive power [2]. The *nested algebra* [16], which extends the traditional, "flat" relational algebra with two restructuring operators, called *nest* and *unnest*, can only express a fragment of the polynomial-time queries over nested databases. Therefore, several extensions of the nested algebra were proposed one of which is its *least-fixpoint closure* [1, 11].Although many more polynomial-time queries on nested databases can be expressed efficiently in this extended language, it was shown in the aforementioned papers that some intractable queries, such as computing the powerset of a relation, can also be expressed in the least-fixpoint closure of the nested algebra. Therefore, proposals were made for extensions of the nested algebra which can only express polynomial-time queries. One such proposal is the *restricted least-fixpoint closure* of the nested algebra introduced by Gyssens and Van Gucht [10]. In the restricted least-fixpoint closure of the nested algebra, the fixpoint construct can only be applied to expressions wherein nesting and unnesting do not occur. Another proposal to extend the expressive power of the nested algebra within PTIME is to consider the *bounded-fixpoint closure* of the nested algebra introduced by Suciu [15]. In the bounded-fixpoint closure of the nested algebra, the fixpoint construct can be applied to expressions in which nesting and unnesting can occur; at each iteration step, however, the intermediate result is intersected with a relation which is constant during the iteration process. Consequently, the final result of an application of the bounded-fixpoint construct is bound by that relation.

It can easily be seen that the expressive power of both the restricted least-fixpoint closure and the bounded-fixpoint closure of the nested algebra is contained in PTIME, and that both extensions are strictly more powerful than the nested algebra. Likewise, it can easily be seen that the expressive power of the bounded-fixpoint closure of the nested algebra is at least that of the restricted least-fixpoint closure. In this paper, we show that they are equivalent.

It was known that neither the relational algebra nor its extension with fixpoints can express all PTIME queries (e.g. transitive closure cannot be expressed in the relational algebra, while parity cannot be expressed in it's extension with fixpoints). Paredaens and Van Gucht [13] prove that the nested relational algebra is a conservative extension of the relational algebra, while Suciu [15] shows that nested relational algebra with bounded fixpoints is a conservative extension of the relational algebra with fixpoints. In some sense these are both negative results, proving that not even with the help of nested relations can we express all of PTIME. The equivalence of the restricted least-fixpoint closure and the bounded-fixpoint closure of the nested algebra which we prove here further confirms that nesting and unnesting are very weak tools indeed to restructure nested databases.

This paper is organized as follows. In Section 2, a typed version of the nested model is presented. In conjunction with the introduction of the model, a notion of substitution is presented which will be used in Section 4 to encode nested databases by flat databases. In Section 3, an overview is given of expressiveness results concerning the nested algebra and some of its extensions. In particular, least-fixpoint extensions are considered. The least-fixpoint closure, the restricted least-fixpoint closure, and the bounded fixpoint closure of the nested algebra are defined. Next, in Section 4, it is shown how nested databases can be represented by flat databases. These techniques are then used in Section 5 to prove the main result of the paper, the equivalence of the restricted least-fixpoint and bounded-fixpoint closures of the nested algebra. Section 6, finally, discusses some interesting ramifications of this result.

## 2 The typed nested model

In this paper, we work essentially with the nested model as it was proposed by Thomas and Fischer [16] and used and extended in work by Gyssens, Paredaens, and Van Gucht [9, 10, 11]. (In the nested model, relation entries need not be "flat" (i.e., atomic), but can in turn be nested relations.) To simplify the proofs in this paper, however, we introduce two major variations with regard to the earlier work of Gyssens, Paredaens, and Van Gucht: (*i*) we work in an attribute-free formalism and (*ii*) we consider multiple flat types. We must emphasize though that these modifications are introduced solely to accommodate our proof techniques, and are not essential for the results in this paper to hold. The nested model modified as outlined above will be referred to as the *typed nested model*.

In our attribute-free approach, a nested relation is a mathematical relation of a certain arity (not necessarily 2) in which the entries may in turn be nested relations. Figure 1 shows a nested relation providing information about persons, their jobs, and the locations in which these jobs are executed.

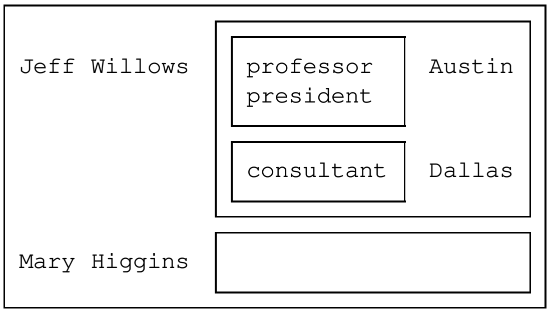


Figure 1: Example of a nested relation

While nested relations as we see them are in essence mathematical relations the entries of which may in turn be relations, our formalism must be somewhat more elaborate to take into account that likewise-numbered components of different tuples in a nested relation must have the same structure. We shall store the information necessary to take this restriction into account into the *type* of the nested relation.

More formally, we assume that we have an infinitely enumerable set of *flat value types*, denoted $\mathfrak{F}$. For each flat value type, $\mathfrak{f}$, we have an infinitely enumerable set of *flat values*, denoted $\mathcal{V}_\mathfrak{f}$. We assume that the sets $\mathcal{V}_\mathfrak{f}$, $\mathfrak{f} \in \mathfrak{F}$, are mutually disjoint. >From flat value types, *relation types* are constructed as follows:

**Definition 2.1** The set of all *relation types*, $\mathfrak{R}$, is the smallest set containing all tuples $\mathfrak{t} = [\mathfrak{t}(1), \ldots, \mathfrak{t}(n)]$, $n \geq 0$, such that, for $i = 1, \ldots, n$, $\mathfrak{t}(i) \in \mathfrak{F} \cup \mathfrak{R}$. The arity of $\mathfrak{t}$, $n$, is denoted $\alpha(\mathfrak{t})$. The set of flat types used in $\mathfrak{t}$ is inductively defined by $\text{flat}(\mathfrak{t}) = \text{flat}(\mathfrak{t}(1)) \cup \ldots \cup \text{flat}(\mathfrak{t}(n))$, where, for each flat type $\mathfrak{f}$, $\text{flat}(\mathfrak{f}) = \{\mathfrak{f}\}$.

Intuitively, a relation type describes the structure of a class of nested relations. For example, if $\mathfrak{string}$ is the string type, then the type of the nested relation in Figure 1 is $[\mathfrak{string}, [[\mathfrak{string}], \mathfrak{string}]]$.

**Definition 2.2** For each relation type $\mathfrak{t} = [\mathfrak{t}(1), \ldots, \mathfrak{t}(n)]$ in $\mathfrak{R}$, the set of *tuples of type* $\mathfrak{t}$, $\mathcal{T}_\mathfrak{t}$, equals $\mathcal{V}_{\mathfrak{t}(1)} \times \ldots \times \mathcal{V}_{\mathfrak{t}(n)}$. The set of *relations of type* $\mathfrak{t}$, $\mathcal{V}_\mathfrak{t}$, consists of all finite subsets of $\mathcal{T}_\mathfrak{t}$. Finally we define the set of *all relations*, $\mathcal{R}$, to equal $\bigcup_{\mathfrak{t} \in \mathfrak{R}} \mathcal{V}_\mathfrak{t}$.

We invite the reader to convince himself that the nested relation in Figure 1 is indeed a relation of type $[\mathfrak{string}, [[\mathfrak{string}], \mathfrak{string}]]$.

In order to be able to refer to relations, we assume the existence of an infinitely enumerable set of *relation names*. In the context of a database or a query, each relation name $R$ will have a fixed type $\mathfrak{t}$, and, whenever necessary, we will emphasize that by writing $R^\mathfrak{t}$. We shall abuse the notation and write $R^{\mathfrak{t}_1}$ and $R^{\mathfrak{t}_2}$ for two different relation names, of type $\mathfrak{t}_1$ and $\mathfrak{t}_2$, respectively. We can now finally define the following:

**Definition 2.3** A *nested database scheme*, $\mathfrak{S}$, is a finite set of relation names. The set of flat types used in $\mathfrak{S}$ is defined by $\text{flat}(\mathfrak{S}) = \bigcup_{R^\mathfrak{t} \in \mathfrak{S}} \text{flat}(\mathfrak{t})$.

A *nested database instance* $I$ over a nested database scheme $\mathfrak{S}$ is a function $I : \mathfrak{S} \to \mathcal{R}$ assigning to each relation name $R^\mathfrak{t}$ in $\mathfrak{S}$ a relation $I(R)^\mathfrak{t}$ in $\mathcal{V}_\mathfrak{t}$.

Obviously, the nested relations and databases encompass the traditional relations and relational databases; we shall refer to the latter with the adjective *flat*.

To prove the main result of the paper, we shall encode nested relations by flat relations in order to be able to apply results obtained in the flat relational model. This encoding will be achieved by substituting flat values for relation values. It is however more convenient to define substitutions the other way around. We shall define substitutions on the type and the value level.

**Definition 2.4** A *type substitution* is a set of pairs $s = \{\mathfrak{t}_1/\mathfrak{f}_1, \ldots, \mathfrak{t}_p/\mathfrak{f}_p\}$, where $\mathfrak{f}_1, \ldots, \mathfrak{f}_p$ are different flat value types and $\mathfrak{t}_1, \ldots, \mathfrak{t}_p$ are relation types. It defines a function $\overline{s} : \mathfrak{F} \cup \mathfrak{R} \to \mathfrak{F} \cup \mathfrak{R} : \mathfrak{t} \mapsto \mathfrak{t}[s]$, as follows:

1. for $i = 1, \ldots, p$, $\mathfrak{t}[s] = \mathfrak{t}_i$ if $\mathfrak{t} = \mathfrak{f}_i$;
2. $\mathfrak{t}[s] = \mathfrak{t}$ if $\mathfrak{t}$ is a flat value type not among $\mathfrak{f}_1, \ldots, \mathfrak{f}_p$; and
3. $\mathfrak{t}[s] = [\mathfrak{t}(1)[s], \ldots, \mathfrak{t}(n)[s]]$ if $\mathfrak{t} = [\mathfrak{t}(1), \ldots, \mathfrak{t}(n)]$ is a relation type.

If $\mathfrak{S} = \{R_1^{\mathfrak{t}_1}, \ldots, R_k^{\mathfrak{t}_k}\}$ is a nested database scheme, then we denote by $\mathfrak{S}[s]$ the database scheme $\{R_1^{\mathfrak{t}_1[s]}, \ldots, R_k^{\mathfrak{t}_k[s]}\}$.

For example consider the type substitution $\mathfrak{s} = \{\mathfrak{a}/[\mathfrak{string}, \mathfrak{string}]\}$. It essentially says that we are going to replace every occurence of the atomic type $\mathfrak{a}$ with $[\mathfrak{string}, \mathfrak{string}]$. E.g. the type $t = [\mathfrak{a}, [\mathfrak{string}, \mathfrak{a}]]$ will be mapped into $t[s] = [[\mathfrak{string}, \mathfrak{string}], [\mathfrak{string}, [\mathfrak{string}, \mathfrak{string}]]]$.

A *value substitution* associated to $s$ is a set $\varphi = \{\varphi_1, \ldots, \varphi_p\}$, where, for $i = 1, \ldots, p$, $\varphi_i$ is an injective function from $\mathcal{V}_{\mathfrak{f}_i}$ to $\mathcal{V}_{\mathfrak{t}_i}$. For $\mathfrak{t} \in \mathfrak{F} \cup \mathfrak{R}$, we shall write $\varphi^\mathfrak{t}$ for the injective function from $\mathcal{V}_\mathfrak{t}$ to $\mathcal{V}_{\mathfrak{t}[s]}$, defined as follows:

1. for $i = 1, \ldots, p$, $\varphi^\mathfrak{t} = \varphi_i$ if $\mathfrak{t} = \mathfrak{f}_i$;
2. $\varphi^\mathfrak{t}$ is the identity function if $\mathfrak{t}$ is a flat value type not among $\mathfrak{f}_1, \ldots, \mathfrak{f}_p$; and
3. $\varphi^\mathfrak{t} = [\varphi^{\mathfrak{t}(1)}, \ldots, \varphi^{\mathfrak{t}(n)}]$ if $\mathfrak{t} = [\mathfrak{t}(1), \ldots, \mathfrak{t}(n)]$ is a relation type.

Finally, if $\mathfrak{S}$ is a nested database scheme, then $\varphi$ extends to a function from instances over $\mathfrak{S}$ to instances over $\mathfrak{S}[s]$ by putting $\varphi(I)(R^{\mathfrak{t}[s]}) = \varphi^\mathfrak{t}(I(R^\mathfrak{t}))$ for each relation name $R^\mathfrak{t}$ in $\mathfrak{S}$.

As an example, let $\mathfrak{new}$ be a flat value type different from $\mathfrak{string}$. If the relation in Figure 1 is encoded by replacing all relation values in its second column by flat values of type $\mathfrak{new}$, then the resulting relation has type $t = [\mathfrak{string}, \mathfrak{new}]$. The type substitution $s = \{[[\mathfrak{string}], \mathfrak{string}]/\mathfrak{new}\}$ precisely captures the relationship between the type of the encoding and the type of the original relation, $t[s]$. The relationship between the entries in the encoding and the entries of the original relation is then captured by a value substitution associated to the above type substitution.

As we shall see later, these actions are but the first step in a whole process aimed at obtaining a flat encoding *without loss of information*.

The language of the nested model is the *nested algebra*, in which queries are expressed by *nested algebra programs* (*nap*s), which are sequences of *nested algebra statements* (*nas*es). A nas assigns to an appropriate relation name the result of a *nested algebra expression* (*nae*). Naes are built from the nested algebra operators, defined below.

**Definition 2.5** Let $r$ and $s$ be nested relations of types $\mathfrak{r}$ and $\mathfrak{s}$, respectively. Let $\alpha(\mathfrak{r}) = m$ and $\alpha(\mathfrak{s}) = n$.

- *Union* ($\cup$), *difference* ($-$), and *intersection* ($\cap$) are binary operators defined on relations of the same type and yield a relation of that type in the usual, set-theoretic way.
- *Product* is a binary operator such that $r \times s$ has type $[\mathfrak{r}(1), \ldots, \mathfrak{r}(m), \mathfrak{s}(1), \ldots, \mathfrak{s}(n)]$ and is defined in the usual, set theoretic way.
- (Generalized) *projection* is a unary operator such that $\pi_{[i(1),\ldots,i(k)]}(r)$, where, for $j = 1, \ldots, k$, $1 \leq i(j) \leq m$, has type $\mathfrak{t} = [\mathfrak{r}(i(1)), \ldots, \mathfrak{r}(i(n))]$ and is defined in the obvious way. Generalized projection can also be used to rearrange or duplicate the columns of a relation.
- *Selection* is a unary operator such that $\sigma_{i=j}(r)$, where $1 \leq i, j \leq m$ and $\mathfrak{r}(i) = \mathfrak{r}(j)$, also has type $\mathfrak{r}$ and is defined as the set of tuples $\{t \in r \mid t(i) = t(j)\}$.
- *Nesting* is a unary operator such that $\nu_i(r)$, where $1 \leq i \leq m$, has type $[\mathfrak{r}(1), \ldots, \mathfrak{r}(i-1), [\mathfrak{r}(i), \ldots, \mathfrak{r}(n)]]$ and is obtained by grouping the tuples of $\mathfrak{r}$ according to their first $i-1$ components.
- *Unnesting* is a unary operator such that $\mu(r)$, where $\mathfrak{r}(n)$ is a relation type, has type $[\mathfrak{r}(1), \ldots, \mathfrak{r}(n-1), \mathfrak{r}(n)(1), \ldots, \mathfrak{r}(n)(k)]$, where $k = \alpha(\mathfrak{r}(n))$, and is obtained by ungrouping the tuples in the relation-valued entries in the last column of $r$.

Let $r$ be the relation in Figure 1. Then $\mu(r)$ is the relation in Figure 2, *left* and $\nu_2\mu(r)$ is the relation consisting of the first tuple of $r$. Finally, the relation $\pi_{[1,3,2]}\mu\pi_{[1,3,2]}\mu(r)$ is the flat relation in Figure 2, *right*.

| | | |
|---|---|---|
| Jeff Willows | professor<br>president | Austin |
| Jeff Willows | consultant | Dallas |

| | | |
|---|---|---|
| Jeff Willows | professor | Austin |
| Jeff Willows | president | Austin |
| Jeff Willows | consultant | Dallas |

Figure 2: Unnesting the relation in Figure 1

>From the operators in Definition 2.5, naes and nases can be defined.

**Definition 2.6** *Nested algebra expressions* (*nae*s) are recursively defined, as follows:

- Each relation name $R^{\mathfrak{t}}$ is an nae of type $\mathfrak{t}$.
- For each relation type $\mathfrak{t}$, $\emptyset_{\mathfrak{t}}$ is an nae of type $\mathfrak{t}$ and $\{[\emptyset_{\mathfrak{t}}]\}$ is an nae of type $[\mathfrak{t}]$.
- Any of the operators in Definition 2.5 applied to naes of appropriate types yields another nae the type of which is obtained from the rules in Definition 2.5.

A *nested algebra statement* (*nas*) has the form $R^{\mathfrak{t}} \leftarrow E(R_1^{\mathfrak{t}_1}, \ldots, R_n^{\mathfrak{t}_n})$, where $R^{\mathfrak{t}}, R_1^{\mathfrak{t}_1}, \ldots, R_n^{\mathfrak{t}_n}$ are relation names and $E(R_1^{\mathfrak{t}_1}, \ldots, R_n^{\mathfrak{t}_n})$ is an nae of type $\mathfrak{t}$.[1] Given a nested database scheme $\mathfrak{S}$ containing $R_1^{\mathfrak{t}_1}, \ldots, R_n^{\mathfrak{t}_n}$, the above nas expresses a query from $\mathfrak{S}$ to $\mathfrak{S} \cup \{R^{\mathfrak{t}}\}$ in the obvious way.

Nested algebra programs (naps) are finite sequences of nases.

**Definition 2.7** Let $\mathfrak{S}_{\text{in}}$ and $\mathfrak{S}_{\text{out}}$ be nested database schemes. A *nested algebra program (nap)* from $\mathfrak{S}_{\text{in}}$ to $\mathfrak{S}_{\text{out}}$ is a sequence of nases such that (*i*) in each nas, the relation names in the right-hand side either occur in $\mathfrak{S}_{\text{in}}$ or in the left-hand side of a preceding nas, and (*ii*) the relation names in $\mathfrak{S}_{\text{out}}$ either occur in $\mathfrak{S}_{\text{in}}$ or in the left-hand side of some nas. Such a nap defines a query from $\mathfrak{S}_{\text{in}}$ to $\mathfrak{S}_{\text{out}}$ by composition of the queries defined by its constituting nases followed by a restriction to $\mathfrak{S}_{\text{out}}$.

As mentioned, the nested algebra encompasses the traditional relational algebra. Naes, nases, and naps of the relational algebra will be called *flat*.

In Definition 2.4, we explained the effect of a type substitution on a nested database scheme. Below, we explain the effect of a substitution on a program.

**Definition 2.8** Let $P$ be a nap from $\mathfrak{S}_{\text{in}}$ to $\mathfrak{S}_{\text{out}}$ and let $s$ be a type substitution. Then $P[s]$ is the program from $\mathfrak{S}_{\text{in}}[s]$ to $\mathfrak{S}_{\text{out}}[s]$ obtained by replacing every relation name $R^{\mathfrak{t}}$ in $P$ with $R^{\mathfrak{t}[s]}$.

Intuitively, if $s = \{\mathfrak{t}_1/\mathfrak{f}_1, \ldots, \mathfrak{t}_p/\mathfrak{f}_p\}$, then $P[s]$ does the same thing as $P$, by treating complex values of types $\mathfrak{t}_1, \ldots, \mathfrak{t}_p$ as if they where atomic values of types $\mathfrak{f}_1, \ldots, \mathfrak{f}_p$. This property is called *polymorphism*, and is made precise below.

**Proposition 2.9 (Polymorphism)** *Let $P$ be a nap from $\mathfrak{S}_{\text{in}}$ to $\mathfrak{S}_{\text{out}}$ and let $s$ be a type substitution. Let $I$ be an instance over $\mathfrak{S}_{\text{in}}$ and let $\varphi$ be a value substitution associated with $s$. Then $P[s](\varphi(I)) = \varphi(P(I))$.*

The proof of this proposition is tedious but straightforward and is omitted. Instead, we wish to elucidate the polymorphism property by observing the very simple fact that it is satisfied by projection. Consider again the relation in Figure 1 and the type substitution $s = \{[[\mathfrak{string}], \mathfrak{string}]/\mathfrak{new}\}$, first considered in the example following Definition 2.4. Our goal is to "flatten" the relation in Figure 1, and the first step towards that is to "pull out" the inner relations in the second column, by substituting them with some fresh values `jc_1` and `jc_2`, of some fresh type $\mathfrak{new}$. Of course, in order to "flatten" it, we need to keep separately, in an additional relation, the connection between `jc_1`, `jc_2` and the relations they replaced, but for the moment we focus on what happens to the main relation once we substitute its relations from the second column with `jc_1` and `jc_2`. Let $\varphi$ be the value substitution mapping these flat values to the two relation-valued entries in the second column of the relation in Figure 1. Consider the following one-line nap $P$ from $\mathfrak{S}_{\text{in}} = \{R^{[\mathfrak{string}, \mathfrak{new}]}\}$ to $\mathfrak{S}_{\text{out}} = \{S^{[\mathfrak{new}]}\}$: $S \leftarrow \pi_{[2]}(R)$, and consider now the nap $Q$ defined to be $P[s]$. $Q$ simply computes the second projection of the input database, and it "treats the relations in the second column as atomic values". The latter statement is made precise by observing that $Q$ is of the form $P[s]$, where the input relation to $P$ indeed has atomic values in the second column (namely of type $\mathfrak{new}$). To see how $Q$ acts on some database instance $I$, start by replacing the second column in $I(R)$ with atomic values, e.g. the relation $R$ in Figure 1 becomes the relation $R^{\mathfrak{t}_{\text{flat}}}$ in Figure 4 (further on in the paper). Intuitively, the computation of $Q$ may proceed in two ways: (1) compute $P$ on $R^{\mathfrak{t}_{\text{flat}}}$, then substitute the atomic values back with their corresponding relations, or (2) do first the substitution on $R^{\mathfrak{t}_{\text{flat}}}$ to get $R$, then compute $Q$ (which is $P[s]$) on this. The polymorphism property tells us that we get that no matter which way we choose, we end up with the same result. In our example, let $I$ be the instance over $\mathfrak{S}_{\text{in}}$ for which $I(R)$ is the relation in Figure 4. Clearly, both $\varphi(P[s](I))$ and $\varphi(P(I))$ are the instance $J$ of $\mathfrak{S}_{\text{out}}[s]$ for which $J(S)$ is the projection of the relation in Figure 1 on its second component.

The following is an important corollary to Proposition 2.9 (proof omitted):

**Proposition 2.10** *Let $s$ be a type substitution and let $P_1$ and $P_2$ be equivalent naps* (*i.e., computing the same query*). *Then $P_1[s]$ and $P_2[s]$ are also equivalent.*

[1] In the sequel, such an nae will be abbreviated to $E^{\mathfrak{t}}$.

We shall use polymorphism as a tool which alows us to avoid object inventions. Namely the key step in our main Theorem 3.6 consists in showing that some query on a database instance $I$ can be "flattened". As suggested above (and shown in detail in the sequel), any database instance $I$ can be encoded as a flat instance $I_{\text{flat}}$, essentially by "inventing" flat values like `jc_1` and `jc_2` above, to replace its nested relations: the query will be accordingly transformed into a query $P$, mapping flat relations to a flat relation. The trick to avoid "value invention" is to use the very relations they replace, in the flat encoding of $I$, instead of the new values like `jc_1` and `jc_2`. But then the query $P$ becomes another query $Q$, which no longer maps flat relations to flat relations, but which treats the inner relations as atomic values: this statement is made precise by the equality $P[s] = Q$, for some type substitution $s$. Hence although $Q$ is not quite a query from flat relations to flat relations, we can still apply the conservativity theorem [15]. Indeed, this theorem says that every query $P$ mapping flat relations to a flat relation is equivalent to some $P'$ having only flat algebra operations. Although we cannot apply this theorem directly to $Q$, we note nevertheless that, by Proposition 2.10 above, $Q$ is equivalent to $Q' = P'[s]$, where $Q'$ consists only of polymorphic instances of flat algebra operations.

# 3 Expressiveness of the nested algebra and its extensions

The nested model was initially proposed to overcome the first-normal-form restriction Codd imposed on the flat relational model [6]. The language of the nested model, the nested algebra, turned out to be very weak, however. Compared to the relational algebra, the nested algebra can do nothing more than group and ungroup data, as was shown by Paredaens and Van Gucht [13]:

**Proposition 3.1** *For every nap from a flat database scheme to a flat database scheme there exists an equivalent flat nap.*

To overcome the inherent weakness of the nested algebra, researchers have proposed several extensions of the nested algebra. One of these extensions is the *least-fixpoint* (*lfp*) *closure* of the nested algebra [1, 11]. In the lfp closure of the nested algebra, queries are expressed by *lfp nap*s, which are defined in much the same way as naps, except that *lfp statements* can occur besides nases.

**Definition 3.2** An *lfp statement* is of the form $R^{\mathfrak{t}} \leftarrow P^*$ with $R^{\mathfrak{t}}$ a relation name and $P$ an lfp nap from some appropriate nested database scheme to $\{R^{\mathfrak{t}}\}$ in which no assignments are made to relation names occurring outside $P$ other than $R^{\mathfrak{t}}$. To $R^{\mathfrak{t}}$, precisely one assignment is made in the last statement of $P$, which is a nas of the form $R^{\mathfrak{t}} \leftarrow E^{\mathfrak{t}}$, where $E^{\mathfrak{t}}$ is an nae.

In a similar way, we can define *inflationary lfp naps* as being composed of nases and inflationary lfp statements. An *inflationary lfp statement* is an lfp statement, $R^{\mathfrak{t}} \leftarrow P^*$, where $P$ is an inflationary lfp nap for which in the last line, $R^{\mathfrak{t}} \leftarrow E^{\mathfrak{t}}$, $E^{\mathfrak{t}} = R^{\mathfrak{t}} \cup F^{\mathfrak{t}}$ for some nae $F^{\mathfrak{t}}$.

Semantically, the effect of a general or inflationary lfp statement $R^{\mathfrak{t}} \leftarrow P^*$ is that $R^{\mathfrak{t}}$ is initialized as the empty relation of type $\mathfrak{t}$ and that the lfp nap $P$ is executed as many times as needed to obtain a fixed value for $R^{\mathfrak{t}}$. If such a fixpoint is not reached, then the effect of the lfp statement (whence the result of the global lfp nap in which it is contained) is considered to be undefined. Notice that, by definition, lfp statements have no side effects.

Consistent with earlier practice, we call lfp naps part of the lfp closure of the relational algebra *flat*. Also, we extend Definition 2.8 to apply also lfp naps.

Figure 3 shows two examples of simple lfp naps. The lfp nap `transitive-closure` in Figure 3, *left*, computes the transitive closure ($T$) of a binary, flat relation ($R$). The lfp nap `powerset` in Figure 3, *right*, computes the powerset ($T^{[\mathfrak{t}]}$) of a unary relation ($R^{\mathfrak{t}}$). (To make programs more readable, type superscripts will often be omitted.)

The existence of the lfp nap `powerset` shows that, contrary to the lfp closure of the flat algebra, the lfp closure of the nested algebra allows the formulation of intractable queries. Therefore, restrictions of the lfp closure of the nested algebra were proposed in which only polynomial-time queries can be expressed. Gyssens and Van Gucht considered the *restricted least-fixpoint* (*rlfp*) *closure* of the nested algebra [10], and Suciu considered the *bounded-fixpoint* (*bfp*) *closure* of the nested algebra [15], both of which can be defined in a similar way as the lfp closure of the nested algebra: rlfp naps consist of nases and *rlfp statements*, and bfp naps consist of nases and *bfp statements*.

**Definition 3.3** An *rlfp statement* is an lfp statement in which nesting and unnesting operators are not allowed to occur.

`transitive-closure`

$$(T \leftarrow T \cup \pi_{[1,4]}\sigma_{2=3}(T \times R) \cup R)^* .$$

`powerset`

$$\begin{aligned} S \leftarrow\ & \{[\emptyset_{\mathfrak{t}}]\} \cup \pi_{[1]}\nu_{[1]}\sigma_{1=2}(R \times R); \\ T \leftarrow (U \leftarrow\ & \pi_{[1]}\nu_{[1]}(\mu_1(\sigma_{1=2}(T \times T \times T) \\ & \cup \sigma_{1=3}(T \times T \times T))); \\ T \leftarrow\ & S \cup U)^* . \end{aligned}$$

Figure 3: Two examples of lfp naps.

A *bfp statement* is an lfp statement, $R^{\mathfrak{t}} \leftarrow P^*$, where $P$ is a bfp nap for which in the last line has the form $R^{\mathfrak{t}} \leftarrow E^{\mathfrak{t}}$, with $E^{\mathfrak{t}} = F^{\mathfrak{t}} \cap S^{\mathfrak{t}}$, for some nae $F^{\mathfrak{t}}$ and some relation name $S$ occurring outside $P$.[2]

The rlfp and bfp closures of the nested algebra are obviously contained in PTIME. Notice that the rlfp and bfp closures of the flat algebra coincide with the general lfp closure of the flat algebra.

The lfp nap `transitive-closure` obviously is an rlfp nap. Modifying this program by intersecting the right-hand side with $S$ where $S$ is assigned the value $\pi_{[1]}(R) \times \pi_{[2]}(R)$ before the execution of the lfp statement yields a bfp nap for the same query.

With respect to the bfp closure of the nested algebra, Suciu [15] proves the following extension of Proposition 3.1:

**Proposition 3.4** *For every* (*inflationary*) *bfp nap from a flat database scheme to a flat database scheme there exists an equivalent flat* (*inflationary*) *lfp nap.*

By lifting Proposition 2.10 and combining it with Proposition 3.1 or Proposition 3.4, we obtain the following result (proof omitted):

**Corollary 3.5** *Let $P$ be a (bfp) nap from the flat database scheme $\mathfrak{S}_{\text{in}}$ to the flat database scheme $\mathfrak{S}_{\text{out}}$. There exists a flat (lfp) nap $P_{\text{flat}}$ from $\mathfrak{S}_{\text{in}}$ to $\mathfrak{S}_{\text{out}}$ such that, for every type substitution s, $P[s] = P_{\text{flat}}[s]$.*

Actually, Corollary 3.5 is a statement about the query expressed by $P[s]$. Informally, it says that whenever a (bfp) nap treats all relation-valued entries as if they were atomic, that (bfp) nap is equivalent to a flat (lfp) nap using only operators in the (lfp closure of the) flat algebra.

In this paper, we prove the following:

**Theorem 3.6** *Every* (*inflationary*) *bfp nap is equivalent to an* (*inflationary*) *rlfp nap, and conversely.*

The following two sections are dedicated to the proof of Theorem 3.6. Essentially, we reduce it to the equivalence of the rlfp and bfp closures of the flat algebra. Corollary 3.5 will be a key lemma in this argument. In order to be able to apply it, however, we must be able to represent a nested database "faithfully" by a flat database. This the subject of the next section.

# 4 Representing nested databases by flat databases

The technique we describe here to represent a nested database by a flat database consists of replacing in every relation $R$ every relation-valued entry with a "new" flat value. Thus $R$ will be replaced by a flat relation, say $R_{\text{flat}}$. We say that $R_{\text{flat}}$ *encodes* $R$. To recover $R$ from its encoding $R_{\text{flat}}$, we need additional information, which we call *translation tables*, capturing the mapping between the flat values in $R_{\text{flat}}$ and the relations in $R$ they replaced. As the translation tables may be nested themselves, the process may have to be repeated.

We first describe one step of the representation process formally.

**Definition 4.1** Let $\mathfrak{f}$ be a flat type, let $\mathfrak{t} = [\mathfrak{t}(1), \ldots, \mathfrak{t}(n)]$ be a relation type, and let $\varphi : \mathcal{V}_{\mathfrak{f}} \rightarrow \mathcal{V}_{\mathfrak{t}}$ be an injective function, and let $v_1, \ldots, v_m$ be different values in $\mathcal{V}_{\mathfrak{f}}$. The *translation table* of $\varphi$ over $\{v_1, \ldots, v_m\}$ is given by two relations, $D^{[\mathfrak{f}]} = \{[v_1], \ldots, [v_m]\}$, and $T^{[\mathfrak{f}, \mathfrak{t}(1), \ldots, \mathfrak{t}(n)]} = \mu(\{[v_1, \varphi(v_1)], \ldots, [v_m, \varphi(v_m)]\})$.

[2]An *inflationary* bfp nap consists of nases and inflationary bfp statements. An inflationary bfp statement is a bfp statement, $R^{\mathfrak{t}} \leftarrow P^*$, where $P$ is an inflationary bfp nap for which in the last line, $R^{\mathfrak{t}} \leftarrow E^{\mathfrak{t}}$, $E^{\mathfrak{t}} = (R^{\mathfrak{t}} \cup F^{\mathfrak{t}}) \cap S^{\mathfrak{t}}$, for some nae $F^{\mathfrak{t}}$ and some relation name $S$ occurring outside $P$.

The translation table encodes complete information about the action of $\varphi$ on the finite set $\{v_1, \ldots, v_m\}$, in the sense that we may re-compute the relation $\{[v_1, \varphi(v_1)], \ldots, [v_m, \varphi(v_m)]\}$ from $D$ and $T$ using a nap. Notice that we need $D$ for the case when, for some $i$, $1 \leq i \leq m$, $\varphi(v_i)$ is the empty relation of type $\mathfrak{t}(i)$.

**Definition 4.2** Let $\mathfrak{t} = [\mathfrak{t}(1), \ldots, \mathfrak{t}(n)]$ be a relation type, and let $s$ be type substitution, $s = \{\mathfrak{t}_1/\mathfrak{f}_1, \ldots, \mathfrak{t}_p/\mathfrak{f}_p\}$ be a type substitution for which $\mathfrak{f}_1, \ldots, \mathfrak{f}_p$ are flat types *not* in flat$(\mathfrak{t})$. A *one-level flattening scheme* of $\mathfrak{t}$ under $s$ is a nested scheme

$$\mathfrak{S}_{1\text{flat}} = \{R^{\mathfrak{t}_{\text{flat}}}, D_1^{[\mathfrak{f}_1]}, T_1^{\mathfrak{t}'_1}, \ldots, D_p^{[\mathfrak{f}_p]}, T_p^{\mathfrak{t}'_p}\}$$

such that (*i*) $\mathfrak{t}_{\text{flat}}$ is a flat type, (*ii*) $\mathfrak{t}_{\text{flat}}[s] = \mathfrak{t}$, and (*iii*), for every $i = 1, \ldots, p$, $\mathfrak{t}'_i = [\mathfrak{f}_i, \mathfrak{t}_i(1), \ldots, \mathfrak{t}_i(n_i)]$, where $n_i = \alpha(\mathfrak{t}_i)$.

We call $R^{\mathfrak{t}_{\text{flat}}}$ the *flat relation associated to* $R^{\mathfrak{t}}$, and $D_1, T_1, \ldots, D_p, T_p$ the *translation tables of* $R^{\mathfrak{t}}$.

A one-level flattening scheme of $\mathfrak{t}$ can easily be obtained by choosing as many flat types not in flat$(\mathfrak{t})$ as there are relation types among $\mathfrak{t}(1), \ldots, \mathfrak{t}(n)$ and replacing the latter by the former. For example, consider the type $\mathfrak{t} = [\mathfrak{string}, [[\mathfrak{string}], \mathfrak{string}]]$ of the relation in Figure 1. Let $s$ be the type substitution $\{[[\mathfrak{string}], \mathfrak{string}]/\mathfrak{new}\}$, considered in the examples following Definitions 2.4 and Proposition 2.9. Then

$$\mathfrak{S}_{1\text{flat}} = \{R^{[\mathfrak{string},\mathfrak{new}]}, D^{[\mathfrak{new}]}, T^{[\mathfrak{new},[\mathfrak{string}],\mathfrak{string}]}\}.$$

Notice that the scheme $\mathfrak{S}_{1\text{flat}}$ flattens $\mathfrak{t}$ only "one level". We need to apply an additional flattening step to $T$ to obtain a fully flat scheme.

**Definition 4.3** Let $\mathfrak{S}_{1\text{flat}} = \{R^{\mathfrak{t}_{\text{flat}}}, D_1^{[\mathfrak{f}_1]}, T_1^{\mathfrak{t}'_1}, \ldots, D_p^{[\mathfrak{f}_p]}, T_p^{\mathfrak{t}'_p}\}$ be a one-level flattening scheme of $\mathfrak{t}$ under the type substitution $s = \{\mathfrak{t}_1/\mathfrak{f}_1, \ldots, \mathfrak{t}_p/\mathfrak{f}_p\}$. An instance $I_{1\text{flat}}$ over $\mathfrak{S}_{1\text{flat}}$ is a *one-level flat encoding* of some relation $R^{\mathfrak{t}}$ under some substitution $\varphi = [\varphi_1, \ldots, \varphi_p]$ associated to $s$, if (*i*) $R^{\mathfrak{t}} = \varphi(I_{1\text{flat}}(R^{\mathfrak{t}_{\text{flat}}}))$ and (*ii*), for $i = 1, \ldots, p$, $I_{1\text{flat}}(D_i)$ and $I_{1\text{flat}}(T_i)$ constitute translation tables for $\varphi_i$ over the flat values occurring in $I_{1\text{flat}}(R^{\mathfrak{t}_{\text{flat}}})$.

Consider the one-level flattening scheme $\mathfrak{S}_{1\text{flat}}$ introduced as an example following Definition 4.2, and let $R$ be the nested relation of Figure 1. Then the instance $I_{1\text{flat}}$ over the scheme $\mathfrak{S}_{1\text{flat}}$ shown in Figure 4 is a one-level flat encoding of $R$ under the value substitution $\varphi$, proposed in the example following Proposition 2.9, and mapping `jc_1` and `jc_2` to the first and second relation-valued entry in the second column of $R$, respectively.

$R^{\mathfrak{t}_{\text{flat}}}$

| | | |
|---|---|---|
| Jeff | Willows | jc_1 |
| Mary | Higgins | jc_2 |

$D$

| |
|---|
| jc_1 |
| jc_2 |

$T$

| | | |
|---|---|---|
| jc_1 | professor<br>president | Austin |
| jc_1 | consultant | Dallas |

Figure 4: A one-level flat encoding of the relation in Figure 1.

**Definition 4.4** Let $\mathfrak{S} = \{R_1^{\mathfrak{t}_1}, \ldots, R_k^{\mathfrak{t}_k}\}$ be a scheme, and let $s$ be a type substitution, $s = \{\mathfrak{t}_1/\mathfrak{f}_1, \ldots, \mathfrak{t}_p/\mathfrak{f}_p\}$ be a type substitution for which $\mathfrak{f}_1, \ldots, \mathfrak{f}_p$ are flat types *not* in flat$(\mathfrak{S})$. The nested database scheme $\mathfrak{S}_{1\text{flat}}$ is a *one-level flattening scheme* of $\mathfrak{S}$ under $s$ if $\mathfrak{S}_{1\text{flat}}$ can be partitioned into $k$ disjoint sets $\mathfrak{S}_{1\text{flat},1}, \ldots, \mathfrak{S}_{1\text{flat},k}$ such that, for $i = 1, \ldots, k$, $\mathfrak{S}_{1\text{flat},i}$ is a one-level flattening scheme of the type $\mathfrak{t}_i$ under $s$.

We shall write $\mathfrak{S}_{1\text{flat}} = \mathfrak{S}^R_{1\text{flat}} \cup \mathfrak{S}^T_{1\text{flat}}$, where $\mathfrak{S}^R_{1\text{flat}}$ contains the flat versions of the relations in $\mathfrak{S}$ and $\mathfrak{S}^T_{1\text{flat}}$ contains all translation tables.

Notice that the same type substitution $s$ must be used for each of the "partial" $k$ one-level flattening schemes.

| `decode` | `encode` | `pseudo-encode` |
|---|---|---|
| $\overline{T} \leftarrow \nu_2(T) \cup (D \times \{[\emptyset_{\mathfrak{s}}]\})$; | $\overline{T} \leftarrow \nu_2(T) \cup (D \times \{[\emptyset_{\mathfrak{s}}]\})$; | $D \leftarrow \pi_{[2]}(R)$; |
| $R \leftarrow \pi_{[1,4]}(\sigma_{2=3}(R_{1\text{flat}} \times \overline{T}))$. | $R_{1\text{flat}} \leftarrow \pi_{[1,3]}(\sigma_{2=4}(R \times \overline{T}))$. | $T \leftarrow \mu(D \times D)$. |

Figure 5: The naps `decode`, `encode`, and `pseudo-encode` for our running example. The type $\mathfrak{s}$ stands for $[[\mathfrak{string}], \mathfrak{string}]$.

**Definition 4.5** Let $\mathfrak{S}_{1\text{flat}} = \mathfrak{S}_{1\text{flat},1} \cup \ldots \cup \mathfrak{S}_{1\text{flat},k}$ be a one-level flattening of the scheme $\mathfrak{S} = \{R_1^{\mathfrak{t}_1}, \ldots, R_k^{\mathfrak{t}_k}\}$ under some type substitution $s$ for which, for $i = 1, \ldots, k$, $\mathfrak{S}_{1\text{flat},i}$ is a one-level flattening scheme of the type $\mathfrak{t}_i$ under $s$. An instance $I_{1\text{flat}}$ over $\mathfrak{S}_{1\text{flat}}$ is *a one-level flat encoding* of some instance $I$ over $\mathfrak{S}$ under some value substitution $\varphi$ associated to $s$ if, for $i = 1, \ldots, k$, $I|_{\mathfrak{S}_{\text{flat},i}}$ is *a one-level flat encoding* of $I(R_i)$. We shall write $I \sim I_{1\text{flat}}$.

We now indicate the relationships between instances and their one-level flat encodings. Rather than proving these propositions, we shall illustrate them by examples.

**Proposition 4.6** *Let $\mathfrak{S}_{1\text{flat}}$ be a one-level flat encoding of some scheme $\mathfrak{S}$. Then there exists a nap* `decode` *from $\mathfrak{S}_{1\text{flat}}$ to $\mathfrak{S}$ such that, for every instance $I$ over $\mathfrak{S}$ and for every instance $I_{1\text{flat}}$ over $\mathfrak{S}_{1\text{flat}}$ with $I \sim I_{1\text{flat}}$,* $\texttt{decode}(I_{1\text{flat}}) = I$.

If $\mathfrak{S} = \{R^{\mathfrak{t}}\}$, where $\mathfrak{t}$ is the type of the relation in Figure 1, then the one-level flattening scheme $\mathfrak{S}_{1\text{flat}}$ of $\mathfrak{t}$ in the example following Definition 4.2 is also a one-level flattening scheme of $\mathfrak{S}$. The corresponding nap `decode` is shown in Figure 5, *left*.

We cannot expect the existence of a nap computing the inverse of `decode`, as there are many one-level flat encodings for the same instance. However, there does exist a nap `encode` computing the flat part if the translation tables are given.

**Proposition 4.7** *Let $\mathfrak{S}_{1\text{flat}} = \mathfrak{S}^R_{1\text{flat}} \cup \mathfrak{S}^T_{1\text{flat}}$ be a one-level flat encoding of some scheme $\mathfrak{S}$. Then there exists a nap* `encode` *from $\mathfrak{S} \times \mathfrak{S}^T_{1\text{flat}}$ to $\mathfrak{S}^R_{1\text{flat}}$ such that, for every instance $I$ over $\mathfrak{S}$ and for every instance $I_{1\text{flat}}$ over $\mathfrak{S}_{1\text{flat}}$ with $I \sim I_{1\text{flat}}$,* $\texttt{encode}(I, I_{1\text{flat}}|_{\mathfrak{S}^T_{1\text{flat}}}) = I_{1\text{flat}}|_{\mathfrak{S}^R_{1\text{flat}}}$.

Continuing with our example, the nap `encode` is shown in Figure 5, *center*.

**Definition 4.8** Let $\mathfrak{S}_f$ be a one-level flattening scheme of $\mathfrak{S}$ under the type substitution $s = \{\mathfrak{t}_1/\mathfrak{f}_1, \ldots, \mathfrak{t}_p/\mathfrak{f}_p\}$. The *one-level pseudo-flattening scheme* of $\mathfrak{S}$, $\mathfrak{S}_{1\text{pf}}$, is unique and equals $\mathfrak{S}_{1\text{flat}}[s]$.

Now let $I$ be an instance over $\mathcal{S}$. The *one-level pseudo-flat encoding* of $I$ is the unique instance $I_{1\text{pf}}$ over $\mathfrak{S}_{1\text{pf}}$, obtained as follows. Let $\mathfrak{S}_{1\text{flat}}$ be any one-level flattening scheme under $s$, and let $I_{1\text{flat}}$ be a corresponding one-level flat encoding of $I$ under some value substitution $\varphi$ associated to $s$. Then $I_{1\text{pf}} = \varphi(I_{1\text{flat}})$.

Each scheme $\mathfrak{S}$ has a unique one-level pseudo-flattening scheme $\mathfrak{S}_{1\text{pf}}$, which is obtained, essentially, by encoding each relation type with itself. Each instance $I$ over $\mathfrak{S}$ has a unique one-level pseudo-flat encoding $I_{1\text{pf}}$ over $\mathfrak{S}_{1\text{pf}}$ which is obtained by substituting the newly introduced flat values in any one-level flat encoding of $I$ by the relation-valued entries they represent.

Continuing with our example, the one-level pseudo-flattening scheme $\mathfrak{S}_{1\text{pf}}$ of $\mathfrak{S} = \{R^{\mathfrak{t}}\}$, $\mathfrak{t}$ being the type of the nested relation in Figure 1, is

$$\{R^{\mathfrak{t}}, D^{[[[\mathfrak{string}],\mathfrak{string}]]}, T^{[[[\mathfrak{string}],\mathfrak{string}],[\mathfrak{string}],\mathfrak{string}]}\}.$$

In the one-level pseudo-flat encoding of $I$, $I_{1\text{pf}}$, $I_{1\text{pf}}(R) = I(R)$, and $I_{1\text{pf}}(D)$ and $I_{1\text{pf}}(T)$ are shown in Figure 6.

**Proposition 4.9** *Let $\mathfrak{S}_{1\text{pf}}$ be a one-level pseudo-flat encoding of some scheme $\mathfrak{S}$. Then there exists a nap* `pseudo-encode` *from $\mathfrak{S}$ to $\mathfrak{S}_{1\text{pf}}$ such that, for every instance $I$ over $\mathfrak{S}$, we have that* $\texttt{pseudo-encode}(I) = I_{1\text{pf}}$.

Continuing on our example, the nap `encode` is shown in Figure 5, *right*. Finally, a flat encoding of a database is obtained by repeatedly constructing one-level flat encodings:

**Definition 4.10** Let $\mathfrak{S}$ be a nested database scheme. A flat database scheme $\mathfrak{S}_{\text{flat}}$ is a *flattening* of $\mathfrak{S}$ if either $\mathfrak{S} = \mathfrak{S}_{\text{flat}}$, or there exists a one-level flattening scheme $\mathfrak{S}_{1\text{flat}}$ of $\mathfrak{S}$ such that $\mathfrak{S}_{\text{flat}}$ is a flattening of $\mathfrak{S}_{1\text{flat}}$.

Let $I$ be an instance over $\mathfrak{S}$. An instance $I_{\text{flat}}$ over $\mathfrak{S}_{\text{flat}}$ is a *flat encoding* of $I$, denoted $I \approx I_{\text{flat}}$, if either $I_{\text{flat}} = I$ (if $\mathfrak{S}_{\text{flat}} = \mathfrak{S}$), or there exists some instance $I_{1\text{flat}}$ over some scheme $\mathfrak{S}_{1\text{flat}}$ such that $I \sim I_{1\text{flat}}$ and $I_{1\text{flat}} \approx I_{\text{flat}}$.

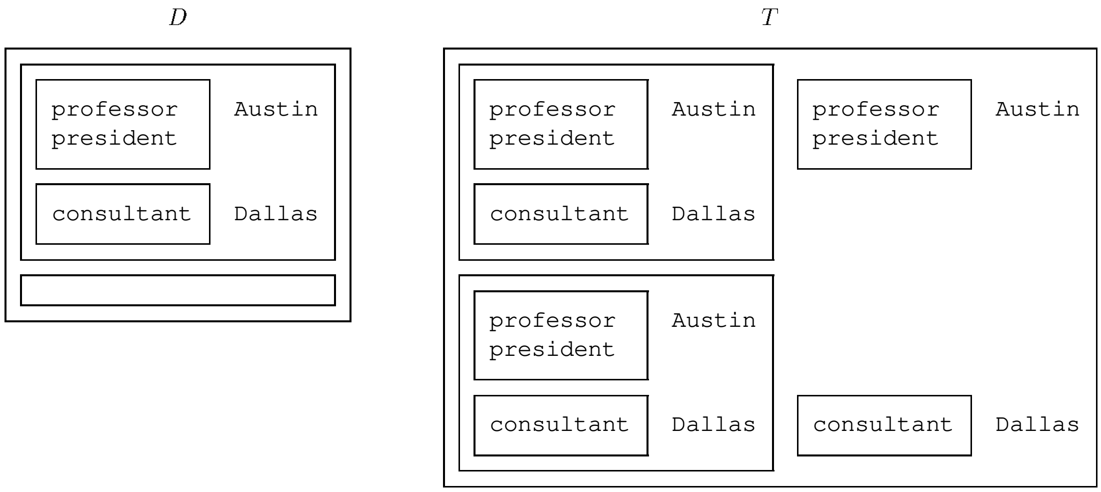


Figure 6: Translation tables of a one-level pseudo-flat encoding of the relation in Figure 1.

The union of all type substitutions involved in the flattening process of an nested database described above yields again a type substitution, say $s$. Similarly, the union of the associated value substitution yields again a value substitution, say $\varphi$. In analogy to Definition 4.8, it is now possible to define the unique *pseudo-flattening scheme* of $\mathfrak{S}$, $\mathfrak{S}_{\text{pf}}$, as $\mathfrak{S}_{\text{flat}}[s]$, and the equally unique *pseudo-flat encoding* of $I$, $I_{\text{pf}}$, as $\varphi(I_{\text{flat}})$.

# 5 The main result

In this section, we prove the main result of this paper, the equivalence of the rlfp and bfp closures of the nested algebra, essentially by reducing this equivalence to the obvious equivalence of the rlfp and bfp closures of the nested algebra, using the flattening techniques developed in the previous section. The present section consists of two lemmas and the actual theorem. The proof sketches of the lemmas explain how the techniques of Section 4 are used to deduce the result.

**Lemma 5.1** *Let $P$ be a bfp nap from $\mathfrak{S}$ to $\mathfrak{S}' = \mathfrak{S} \cup \{R^{\mathfrak{t}}\}$ consisting of only one bfp statement, $R^{\mathfrak{t}} \leftarrow Q^*$, where $Q$ is an arbitrary bfp nap whose last instruction is $R^{\mathfrak{t}} \leftarrow E^{\mathfrak{t}} \cap S^{\mathfrak{t}}$. Let $\mathfrak{S}_{\text{flat}}$ and $\mathfrak{S}'_{\text{flat}}$ be total flattenings of, respectively, $\mathfrak{S}$ and $\mathfrak{S}'$ under some common type substitution $s$ (whence $\mathfrak{S}_{\text{flat}} \subseteq \mathfrak{S}'_{\text{flat}}$). Then there exists a bfp nap $P_{\text{flat}}$ from $\mathfrak{S}_{\text{flat}}$ to $\mathfrak{S}'_{\text{flat}}$ such that, for each instance $I$ over $\mathfrak{S}$ and for each instance $I_{\text{flat}}$ over $\mathfrak{S}_{\text{flat}}$ with $I \approx I_{\text{flat}}$, we have that $P(I) \approx P_{\text{flat}}(I_{\text{flat}})$.*

**Proof.** The program $P_{\text{flat}}$ will proceed as follows on input $I_{\text{flat}}$. First, $I$ is computed from $I_{\text{flat}}$, by repeatedly applying `decode`. In the process, it will retain all translation tables. Next, $P$ will be applied to $I$. Now, by the definition of $P$, $P(I)(R^{\mathfrak{t}}) \subseteq I(S^{\mathfrak{t}})$. Therefore, we can use the translation tables in $I_{\text{flat}}$ to obtain a flat encoding of $P(I)$, by repeatedly applying `encode`. By construction, the result, $P_{\text{flat}}(I_{\text{flat}})$, satisfies $P(I) \approx P_{\text{flat}}(I_{\text{flat}})$. □

**Lemma 5.2** *Every one-line (inflationary) bfp nap is equivalent to an (inflationary) rlfp nap.*

**Proof.** Let $P$ be as in Lemma 5.1. We show there exists an rlfp nap $\tilde{P}$ equivalent to the one-line bfp nap $P$.

Let $\mathfrak{S}_{\text{flat}}$, $\mathfrak{S}'_{\text{flat}}$, $s$, and $P_{\text{flat}}$ be as in Lemma 5.1. We apply $s$ to $\mathfrak{S}_{\text{flat}}$, $\mathfrak{S}'_{\text{flat}}$, and $P_{\text{flat}}$ to obtain $\mathfrak{S}_{\text{pf}} = \mathfrak{S}_{\text{flat}}[s]$, $\mathfrak{S}'_{\text{pf}} = \mathfrak{S}'_{\text{flat}}[s]$, and $P_{\text{flat}}[s]$. By Corollary 3.5, $P_{\text{flat}}[s]$ is equivalent to an (inflationary) rlfp nap, $P_{\text{rlfp}}$.

The (inflationary) rlfp nap $\tilde{P}$ will proceed as follows on input $I$ over $\mathfrak{S}$. First, the pseudo-flat encoding $I_{\text{pf}}$ over $\mathcal{S}_{\text{pf}}$ is computed, by repeatedly applying `pseudo-encode` (a nap). Next, $P_{\text{rlfp}}$ (an (inflationary) rlfp nap) is applied to $I_{\text{pf}}$ to get $P_{\text{rlfp}}(I_{\text{pf}}) = P_{\text{flat}}[s](I_{\text{pf}})$. Let `full-decode` be the composition of the `decode` programs needed to obtain $\mathfrak{S}$ and $I$ from $\mathfrak{S}_{\text{flat}}$ and $I_{\text{flat}}$. Then finally, `full-decode`$[s]$ (a nap) is applied to get `full-decode`$[s](P_{\text{flat}}[s](I_{\text{pf}}))$.

To complete the proof, we have to argue that the above instance is indeed $P(I)$. By definition, we have $I_{\text{pf}} = \varphi(I_{\text{flat}})$, where $I_{\text{flat}}$ is a flat encoding of $I$ over $\mathfrak{S}_{\text{flat}}$, under some value substitution $\varphi$ associated to $s$. By Proposition 2.9, we have

$$\begin{aligned}\texttt{full-decode}[s](P_{\text{flat}}[s](I_{\text{pf}})) &= \texttt{full-decode}[s](P_{\text{flat}}[s](\varphi(I_{\text{flat}}))) \\ &= \texttt{full-decode}[s](\varphi(P_{\text{flat}}(I_{\text{flat}}))) \\ &= \varphi(\texttt{full-decode}(P_{\text{flat}}(I_{\text{flat}}))).\end{aligned}$$

By Lemma 5.1, we have that $P(I) \approx P_{\text{flat}}(I_{\text{flat}})$, whence

$$\texttt{full-decode}(P_{\text{flat}}(I_{\text{flat}})) = P(I),$$

which had to be shown. □

We can now return to our main theorem:

**Theorem 3.6** *Every* (*inflationary*) *bfp nap is equivalent to an* (*inflationary*) *rlfp nap, and conversely.*

Theorem 3.6 follows from applying Lemma 5.2 to each (inflationary) bfp statement in the (inflationary) bfp nap. The converse is obvious.

Theorem 3.6 further emphasizes the inherent weakness of the nest and unnest operators: even in connection with a least-fixpoint construct they do not yield additional expressive power as far as the expression of polynomial-time queries is concerned.

# 6 Applications of the main result

The main result (Theorem 3.6) can be used to derive a normal form for bfp naps.

**Proposition 6.1** *Every* (*inflationary*) *bfp nap is equivalent to an* (*inflationary*) *bfp only containing nases and bfp statements of the form $R \leftarrow Q^*$, with $Q$ a nap* (*i.e., without fixpoints*).

**Proof.** On each of the rlfp statements constructed in Lemma 5.2 to obtain the rlfp nap of Theorem 3.6, we apply either the result of Gurevich and Shelah [8] about the collapse of the FO+IFP hierarchy (in the inflationary case) or the result of Abiteboul and Vianu [3] about the collapse of the FO+PFP hierarchy (in the general case) to obtain (inflationary) rlfp statements in which the fixpoint operator only occurs at the outermost level. These (inflationary) rlfp can be translated straightforwardly into (inflationary) bfp statements (provided suitable "constant" relations are computed in a nas preceding the bfp statement). □

Suciu [15] proved that the nested algebra with bounded fixpoints expresses exactly the PTIME queries over *ordered* nested databases. Hence Proposition 6.1 also yields a simple normal form for PTIME queries over *ordered* nested databases.

In addition, Theorem 3.6 allows us to derive a new characterization of the PTIME = PSPACE problem.

**Proposition 6.2** *Inflationary bfp naps are equivalent to bfp naps if and only if* PTIME = PSPACE.

**Proof.** Suppose every bfp nap is equivalent to some inflationary bfp nap. In particular, it then follows that every flat lfp nap is equivalent to some inflationary flat lfp nap, i.e., that FO+IFP = FO+PFP. By a result of Abiteboul and Vianu [4], it then follows that PTIME = PSPACE.

Conversely, suppose that PTIME = PSPACE. By the same result in [4] it follows that every flat lfp nap is equivalent to some inflationary flat lfp nap. Now let $P$ be a bfp nap. By Theorem 3.6, $P$ is equivalent to an rlfp nap. Moreover, each rlfp statement in this rlfp nap can be obtained by type substitution from a flat lfp statement, and hence from a flat inflationary lfp statement. Thus the rlfp nap obtained in Theorem 3.6 is equivalent to an inflationary rlfp nap, which in turn is equivalent to an inflationary bfp nap. □

# References

[1] S. Abiteboul, C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. In: Proceedings International Workshop on Theory and Applications of Nested Relations and Complex Objects. (Darmstadt, April 1987. Also: INRIA Technical Report no. 864, May 1988

[2] S. Abiteboul, C. Beeri, M. Gyssens, D. Van Gucht. An Introduction to the Completeness of Languages for Complex Objects and Nested Relations. In: S. Abiteboul, P.C. Fischer, and H.-J. Schek (eds.) Nested Relations and Complex Objects in Databases. Springer-Verlag, 1989, pp. 117–138. (Lecture Notes in Computer Science no. 361)

[3] S. Abiteboul, V. Vianu. Datalog extensions for database queries and updates. Journal of Computer and System Sciences 1991; 43:62–124

[4] S. Abiteboul, V. Vianu. Generic Computation and Its Complexity. In: Proceedings ACM SIGACT Symposium on the Theory of Computing. 1991, pp. 209–219

[5] V. Breazu-Tannen, P. Buneman, L. Wong. Naturally Embedded Query Languages. In: Proceedings 4th International Conference on Database Theory (Berlin, Germany). Springer-Verlag, 1992, pp. 140–154 (Lecture Notes in Computer Science no. 646). Also: University of Pennsylvania Technical Report MS-CIS-92-47

[6] E.F. Codd. Relational Model of Data for Large Shared Data Banks. Communications of the ACM 1970; 13:377–387

[7] L.S. Colby. A Recursive Algebra for Nested Relations. Information Systems 1990; 15:567–582

[8] Y. Gurevich, S. Shelah. Fixed-Point Extensions of First-Order Logic. Annals of Pure and Applied Logic 1986; 32:265–280

[9] M. Gyssens, J. Paredaens, D. Van Gucht. A Uniform Approach toward Handling Atomic and Structured Information in the Nested Relational Database Model. Journal of the Association for Computing Machinery 1989; 36:790–825

[10] M. Gyssens, D. Van Gucht. A Comparison between Algebraic Query Languages for Flat and Nested Databases. Theoretical Computer Science 1991; 87:263–286.

[11] M. Gyssens, D. Van Gucht. The Powerset Algebra as a Natural Tool to Handle Nested Database Relations. Journal of Computer and System Sciences 1992; 45:76–103

[12] G. Jaeschke, H.-J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In: Proceedings 1st Symposium on Principles of Database Systems (Los Angeles, California). ACM Press, 1982, pp. 124–138

[13] J. Paredaens, D. Van Gucht. Converting Nested Algebra Expressions into Flat Algebra Expressions. ACM Transactions on Database Systems 1992; 17:65–93

[14] H.-J. Schek, M.H. Scholl. The Relational Model with Relation-Valued Attributes. Information Systems 1986; 11:137–147

[15] D. Suciu. Fixpoints and Bounded Fixpoints for Complex Objects. In: C. Beeri, A. Ohori, D. Shasha (eds.) Database Programming Languages (Proceedings 4th International Workshop on Database Programming Languages, Manhattan, New York, August-September 1993), Workshops on Computing, Springer-Verlag, 1994, pp. 263–281. Also: University of Pennsylvania Technical Report MS-CIS-93-32

[16] S.J. Thomas, P.C. Fischer. Nested Relational Structures. In: P.C. Kanellakis (ed.) The Theory of Databases. JAI Press, 1986, pp. 269–307