Normalizing Incomplete Databases

Leonid Libkin
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974 USA
E-mail: libkin@research.att.com

Abstract

Databases are often incomplete because of the presence of disjunctive information, due to conflicts, partial knowledge and other reasons. Queries against such databases often ask questions about various possibilities encoded by the stored data, rather than the stored data itself. Normalization, which is a mechanism for asking such queries, was presented in [LW93a]; however, it had exponential space complexity.

The main goal of this paper is to develop a general theory of answering queries against incomplete databases with disjunctive information, and use it to design practical algorithms for query evaluation. We define the semantics of such databases and prove normalization theorems for set- and bag-based complex objects. These theorems provide us with programming primitives that one needs in order to obtain the list of all possibilities encoded by a complex object with disjunctions.

We study two ways of making query evaluation faster and more space efficient. Partial normalization allows us to disregard some of the disjunctions if they do not affect a given query. We also design a new normalization algorithm that produces objects represented by an incomplete database one-by-one, rather than all at once. It has linear space complexity and allows us to speed up many classes of queries.

Algorithms presented in this paper have been implemented in existing dbpl. We present experimental results that demonstrate substantial improvement over standard algorithms, both in space and time.

1 Introduction

Information stored in databases is usually incomplete. One of the typical sources of partiality, along with null values [AKG91, IL84], is disjunctive information that occurs primarily in the areas of design and planning, as was noticed in [INV91a, INV91b]. It may also arise due to conflicts that occur when different databases are merged.

A number of approaches to querying databases with disjunctions are known in the literature. The idea of using *and-or* trees to develop a new object oriented data model with an ad hoc query facility was exploited in [INV91a, INV91b]. The query complexity in this model was analyzed in [IMV89]. Recently, a functional query language for databases with disjunctions was designed [LW93a] and implemented [GL94]. In these papers two kinds of queries have been distinguished: *structural* queries ask questions about the data stored in a database, whereas *conceptual* queries ask questions about the data encoded by the information in a database. To illustrate the difference between the structural and conceptual queries, consider the following example of an incomplete design borrowed from [GL94], see figure 1.

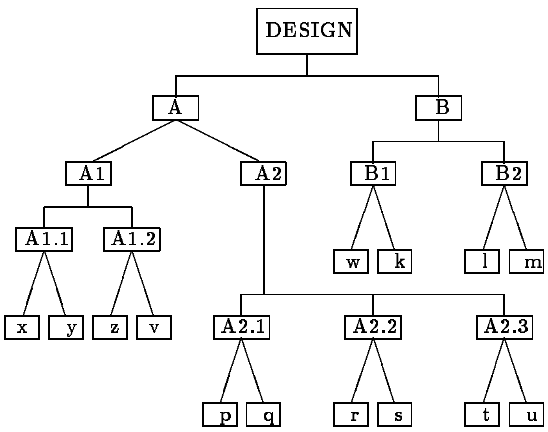

Figure 1: Incomplete design

In this figure vertical and horizontal lines represent subparts that must be included in the design, while the sloping lines represent possible choices. For example, the whole design consists of two parts: A and B. An A is either an $A1$ or an $A2$, and a B consists of a $B1$ and a $B2$, where a $B1$ is either a w or a k. Structural queries ask about the structure of a given object. For example, "what is the least expensive choice for $B2$" and "how many subparts does $A2$ have" are examples of *structural queries*.

Conceptual queries ask questions about possible completed designs. For example, "how many completed designs are there" and "is there a completed design that costs under \$100 and has reliability at least 95%" are examples of conceptual queries.

To distinguish ordinary sets from collections of disjunctive possibilities, we call the latter *or-sets*, see [INV91a, LW93a, Rou91]. We use $\langle\rangle$ to denote or-sets. In the example in figure 1, the whole design can be represented as a set $\{A, B\}$, while A is an or-set $\langle A1, A2\rangle$ and $B2$ is an or-set $\langle w, k\rangle$. Note that or-sets have two distinct representations. With respect to structural queries, or-sets behave like sets, but with respect to conceptual queries, an or-set denotes one of its elements. For example, $\langle 1, 2\rangle$ is structurally a two-element set, but conceptually it is an integer that equals either 1 or 2.

A mechanism for answering conceptual queries against complex objects with or-sets, called *normalization*, was presented in [LW93a]. Roughly speaking, it provides us with a small number of programming primitives that, when repeatedly applied to an object o, create an or-set that lists all possibilities encoded by o (like completed designs). This or-set is called the *normal form* of o. Then conceptual queries are simply structural queries on normal forms.

Normalization, as presented in [LW93a], provides the solid theoretical foundation for developing languages in which conceptual queries can be formulated. It also has led to development of a prototype [GL94]. However, there are several theoretical problems that must be addressed in order to develop *practical* methods for answering conceptual queries.

- Only sets have been considered in [INV91a, INV91b, LW93a, Rou91], but many practical languages are based on bags (multisets). In the past few years several approaches to design of *bag* languages have been proposed. Moreover, most approaches agree on what constitutes the basic set of bag operations [Alb91, GM93, LW93b, LW94]. Thus, we believe the normalization mechanism must be extended to bags.
- Normalization may cause exponential blowup in the size of objects. For objects of size n, the size of their normal forms is bounded (roughly) by $n \cdot 1.45^n$ [LW93a]. Therefore, we need better normalization tools. One possibility is to normalize *partially.* If some of the disjunctions do not affect the conceptual query that is asked, there is no need to unfold those disjunctions. The problem of partial normalization has not been addressed in the literature.
- Normalization, as presented in [LW93a], requires that the whole normal form be created before any conceptual queries could be asked. Therefore, it has exponential *space* complexity. Alternatively, one may want to produce normal form elements (e.g. completed designs) one-by-one, rather than all at once, thus making the space usage linear.

The main goal of the paper is to address these shortcomings of the normalization process. As the outcome, we shall have much better tools for querying databases with disjunctive information and much better understanding of their structure. The main contributions of this paper are listed below.

1. We rigorously define normal forms (or conceptual semantics) of objects with or-sets and prove normalization theorems giving us a small number of operations that construct normal forms. We do this for *both set and bag* semantics.
2. We prove a *partial normalization* result that tells us when the normalization process need not be completed in order to answer a conceptual query. We give a restriction on types of objects for which this can be done.
3. We design a *linear space algorithm* that produces all elements in the normal form, and suggest a new programming primitive based on it. This primitive allows us to express a number of important queries (including a class of existential conceptual queries) in a uniform fashion.
4. We consider interaction of disjunctive information with traditional forms of partial information, represented via orders on objects, and prove both normalization and partial normalization theorems in this setting.
5. We implement the new space-efficient algorithm in the system for querying databases with disjunctions [GL94]. We compare it with the standard algorithm and demonstrate substantial improvement. We show how the new programming

primitive can be used together with some heuristics to answer conceptual queries approximately, when normalization process is very expensive.

Organization. We define structural semantics and normal forms in section 2. Normalization theorems for sets and bags and partial normalization theorem are proved in section 3. The space-efficient normalization algorithm and a programming primitive based on it are presented in section 4. Normalization in the presence of partial information is studied in section 5. Experimental results are presented in section 6.

Remark. Our approach to disjunctive information as a form of partial information should not be confused with the work on disjunctive deductive databases [LMR92]. For differences between these approaches, see [INV91a, INV91b].

2 Semantics and normal forms

As we mentioned before, objects with or-sets can be treated at the structural and conceptual levels. Consequently, there are two different semantics for or-objects. One of them treats or-sets as collections, while the other takes into account that an or-set denotes one of its elements.

To state this precisely, we first define *types* of objects. There are two type systems of interest: one dealing with sets and the other with multisets (bags):

(ST) $\quad t := b \mid t \times t \mid \{t\} \mid \langle t \rangle$

(BT) $\quad s := b \mid s \times s \mid \{\!|s|\!\} \mid \langle s \rangle$

Here b ranges over a collection of base types such as integers, booleans etc. $t \times t'$ is the product type; its elements are pairs (x, y) where x has type t and y has type t'. Values of the set type $\{t\}$ are finite sets of elements of type t. Values of $\{\!|t|\!\}$ and $\langle t \rangle$ are finite bags and or-sets of values of type t respectively. If $\mathbb{P}_{\text{fin}}(X)$ stands for the finite powerset of X and $\mathbb{P}_b(X)$ for the family of finite bags over X, then, assuming that a domain D_b of each base type is given, we define the *structural semantics* of types as follows:

- $[\![b]\!]_s = D_b$
- $[\![t \times t']\!]_s = [\![t]\!]_s \times [\![t']\!]_s$
- $[\![\{t\}]\!]_s = [\![\langle t \rangle]\!]_s = \mathbb{P}_{\text{fin}}([\![t]\!]_s)$
- $[\![\{\!|t|\!\}]\!]_s = \mathbb{P}_b([\![t]\!]_s)$

An object whose type is in the type system (ST) is called a *set-based complex object*. An object whose type is in (BT) is called a *bag-based complex object.* Any object containing or-sets is also called an *or-object.*

We need two translations between (ST) and (BT) and between set-based and bag-based objects. First, for any type t in (ST), we define t^{Bag} in (BT) by replacing all set brackets by bag brackets. Type s^{Set} is defined as s in which all bag brackets are replaced by set brackets. For any object X of an (ST) type t, define X^{Bag} of type t^{Bag} by replacing each set in X by a bag with the same elements and all multiplicities equal 1. For example, $(\{1,2\},\{3,4\})^{\text{Bag}} = (\{\!|1,2|\!\},\{\!|3,4|\!\})$. Conversely, for Y of a (BT) type s, Y^{Set} of type s^{Set} is defined by replacing each bag in Y with the set containing all elements of that bag (i.e. duplicates are eliminated). For example, $\{\!|\{\!|1,1,2|\!\},\{\!|1,2,2|\!\}|\!\}^{\text{Set}} = \{\{1,2\}\}$.

It should be noted that $(t^{\text{Bag}})^{\text{Set}} = t$ for any (ST) type t, and $(t^{\text{Set}})^{\text{Bag}} = t$ for any (BT) type t. However, while $(X^{\text{Bag}})^{\text{Set}} = X$ for any set-based object X, it is not necessarily the case that $(Y^{\text{Set}})^{\text{Bag}} = Y$ for a bag-based object Y.

Before we define the conceptual semantics, which will be called *normal form*, we need the notion of the *skeleton* of a type. The skeleton $sk(t)$ of a type t is defined to be the type formed by removing all or-set brackets from t. That is, $sk(b) = b$, $sk(t \times t') = sk(t) \times sk(t')$, $sk(\{t\}) = \{sk(t)\}$, $sk(\{\!|t|\!\}) = \{\!|sk(t)|\!\}$ and $sk(\langle t \rangle) = sk(t)$.

Next, we define a binary relation $x \lessdot y$ among objects whose meaning intuitively is "x is in the conceptual representation of y". (For example, $d \lessdot$ DESIGN iff d is a completed design.)

- For any x, x' of a base type, $x' \lessdot x$ iff $x = x'$.
- $(x', y') \lessdot (x, y)$ iff $x' \lessdot x$ and $y' \lessdot y$.
- $\{\!|x'_1, \ldots, x'_n|\!\} \lessdot \{\!|x_1, \ldots, x_n|\!\}$ iff there exists a permutation σ on $\{1, \ldots, n\}$ such that $x'_i \lessdot x_{\sigma(i)}$ for all $i = 1, \ldots, n$.
- $\{x'_1, \ldots, x'_n\} \lessdot \{x_1, \ldots, x_k\}$ iff there exists a partition $X_1, \ldots, X_n$ of $\{x_1, \ldots, x_k\}$ such that for any $i = 1, \ldots, n$ and for any $x \in X_i$: $x'_i \lessdot x$.
- $x \lessdot \langle x_1, \ldots, x_k \rangle$ iff $x \lessdot x_i$ for some x_i. (Recall that an or-set denotes one of its elements.)

Note that in the set clause it is not enough to ask for a permutation of elements $\{x_1, \ldots, x_n\}$ that would satisfy $x'_i \lessdot x_{\sigma(i)}$ because some of those x'_i may then be the same and $\{x'_1, \ldots, x'_n\}$ would not be a set. Hence, we need partitions.

Definition. *For any object X, its* normal form *$nf(X)$ is defined as the or-set $\langle x_1, \ldots, x_n \rangle$ of all objects x_i such that $x_i \lessdot X$. Note that the normal form is always finite.*

Lemma 1 *If X is of type t, then any $x \lessdot X$ is of type $sk(t)$. In particular, for any or-object X of type t, its normal form $nf(X)$ is of type $\langle sk(t)\rangle$.* □

In other words, the normal form of an object lists all possibilities that are encoded by the disjunctions present in that object. Each normal form entry is a regular complex object, i.e. does not have any or-sets.

3 Normalization theorems

The general idea of the normalization theorems is to give a list of operations that can be repeatedly applied to an object until the normal form is produced. Such a list was first presented in [LW93a]; here we go further in several aspects. First, we clearly distinguish between set and bag semantics. Second, we prove a *partial normalization* result that can be viewed as normalization at intermediate types. That is, while the standard normalization theorems find a unique representation of an object of type t at type $\langle sk(t)\rangle$, the partial normalization result finds such a representation at type s where s is "between" t and $\langle sk(t)\rangle$. To guarantee uniqueness, some restrictions on types must be imposed.

We need a language to express the operations used for normalizing objects. We adopt the framework of [LW93a] which in turn is based on [BBW92] and finds its origins in [AB88, BBN91]. The operators together with their most general types are given in figure 2.

General operators

$$\frac{g : u \to s \quad f : s \to t}{f \circ g : u \to t} \qquad \frac{f : u \to s \quad g : u \to t}{(f, g) : u \to s \times t}$$

$$\frac{}{\pi_1 : s \times t \to s} \qquad \frac{}{\pi_2 : s \times t \to t}$$

$$\frac{}{! : t \to unit} \qquad \frac{}{eq : t \times t \to bool} \qquad \frac{}{id : t \to t}$$

$$\frac{c : bool \quad f : s \to t \quad g : s \to t}{cond(c, f, g) : s \to t}$$

Operators on sets

$$\frac{}{K\{\} : unit \to \{t\}} \qquad \frac{}{\rho_2 : s \times \{t\} \to \{s \times t\}}$$

$$\frac{}{\cup : \{t\} \times \{t\} \to \{t\}} \qquad \frac{}{\eta : t \to \{t\}}$$

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

Operators on bags

$$\frac{}{K\{\!|\,|\!\} : unit \to \{\!|t|\!\}} \qquad \frac{}{b\_\rho_2 : s \times \{\!|t|\!\} \to \{\!|s \times t|\!\}}$$

$$\frac{}{\uplus : \{\!|t|\!\} \times \{\!|t|\!\} \to \{\!|t|\!\}} \qquad \frac{}{b\_\eta : t \to \{\!|t|\!\}}$$

$$\frac{f : s \to t}{b\_map\ f : \{\!|s|\!\} \to \{\!|t|\!\}} \qquad \frac{}{b\_\mu : \{\!|\{\!|t|\!\}|\!\} \to \{\!|t|\!\}}$$

Operators on or-sets

$$\frac{}{K\langle\rangle : unit \to \langle t\rangle} \qquad \frac{}{or\_\rho_2 : s \times \langle t\rangle \to \langle s \times t\rangle}$$

$$\frac{}{or\_\cup : \langle t\rangle \times \langle t\rangle \to \langle t\rangle} \qquad \frac{}{or\_\eta : t \to \langle t\rangle}$$

$$\frac{f : s \to t}{or\_map\ f : \langle s\rangle \to \langle t\rangle} \qquad \frac{}{or\_\mu : \langle\langle t\rangle\rangle \to \langle t\rangle}$$

Interaction

$$\frac{}{\alpha : \{\langle t\rangle\} \to \langle\{t\}\rangle} \qquad \frac{}{b\_\alpha : \{\!|\langle t\rangle|\!\} \to \langle\{\!|t|\!\}\rangle}$$

Figure 2: Operators of *or-$\mathcal{NRL}$* and *b_or-$\mathcal{NRL}$*

Recall briefly the semantics of the general and set operators. $f \circ g$ is composition of functions; (f, g) is pair formation. π_1 and π_2 are the first and the second projections. ! always returns the unique element of a special base type *unit*. *eq* is equality test; *id* is the identity and *cond* is conditional. For set operations: $K\{\}$ is the function that represents the constant $\{\}$; η forms singletons: $\eta(x) = \{x\}$; $\cup$ takes union of two sets; μ flattens sets of sets: $\mu(\{\{1, 2\}, \{2, 3\}\}) = \{1, 2, 3\}$; $map(f)$ applies f to all elements of a set; and ρ_2 is pair-with: $\rho_2(1, \{2, 3\}) = \{(1, 2), (1, 3)\}$.

Operators on or-sets are exactly the same as operators on sets except that the prefix *or* is added. Operators on bags are similar to those on sets, but additive union that adds up multiplicities is used. Also, flattening for bags is additive: $b\_\mu(\{\!|B_1, \ldots, B_n|\!\}) = B_1 \uplus \ldots \uplus B_n$.

Finally, α and b_α provide interaction between sets and or-sets and between bags and or-sets. Assume that $\mathcal{X} = \{X_1, \ldots, X_n\}$ and $\mathcal{Y} = \{\!|Y_1, \ldots, Y_n|\!\}$ where $X_i = \langle x_1^i, \ldots, x_{n_i}^i\rangle$ and $Y_i = \langle y_1^i, \ldots, y_{n_i}^i\rangle$. Let $\mathcal{F}$ be the family of "choice" functions from $\{1, \ldots, n\}$ to $\mathbb{N}$ such that $1 \le f(i) \le n_i$ for all i. Then

$$\alpha(\mathcal{X}) = \langle\{x_{f(i)}^i \mid i = 1, \ldots, n\} \mid f \in \mathcal{F}\rangle$$

$$b\_\alpha(\mathcal{Y}) = \langle\{\!|y_{f(i)}^i \mid i = 1, \ldots, n|\!\} \mid f \in \mathcal{F}\rangle$$

The main difference between these two definitions is that duplicates are removed from sets but not from bags. For example, $\alpha(\{\langle 1, 3\rangle, \langle 2, 3\rangle\})$ evaluates to $\langle\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3\}\rangle$, but $b\_\alpha(\{\!|\langle 1, 3\rangle, \langle 2, 3\rangle|\!\})$ is equal to $\langle\{\!|1, 2|\!\}, \{\!|1, 3|\!\}, \{\!|2, 3|\!\}, \{\!|3, 3|\!\}\rangle$.

Definition (see also [LW93a]). *The language or-$\mathcal{NRL}$ over type system (ST) includes all general operators, set operators, or-set operators and α. The language b_or-$\mathcal{NRL}$ over type system (BT) includes all general operators, bag operators, or-set operators and b_α.*

3.1 Normalizing types

Define the following rewrite rules on types:

$$s \times \langle t \rangle \to \langle s \times t \rangle \qquad \langle s \rangle \times t \to \langle s \times t \rangle \qquad \langle\langle t \rangle\rangle \to \langle t \rangle$$
$$\{\langle t \rangle\} \to \langle \{t\} \rangle \qquad \{\!|\langle s \rangle|\!\} \to \langle \{\!|s|\!\} \rangle$$

Define the rewrite system (STR) on (ST) types as the three rules in the first line and $\{\langle t \rangle\} \to \langle \{t\} \rangle$. The rewrite system (BTR) on (BT) types is defined as the top three rules and $\{\!|\langle s \rangle|\!\} \to \langle \{\!|s|\!\} \rangle$. We use the notation $s \twoheadrightarrow t$ if s rewrites to t in zero or more steps. Recall [DJ90] that a normal form of a rewrite system is a term that cannot be further rewritten.

Proposition 2 (see [LW93a]) *Both (STR) and (BTR) are terminating Church-Rosser rewrite systems. Consequently, each type has a unique normal form that can be calculated as $\langle sk(t) \rangle$ for any type t that involves or-sets.* □

3.2 Normalizing complex objects

It was suggested in [LW93a] to assign functions in the language to the rewrite rules so that for every rewriting from s to t there would be an associated definable function of type $s \to t$. The goal of this assignment is to obtain a function of type $s \to \langle sk(s) \rangle$ that produces the normal forms for objects of type s.

In subsection 3.3 we explain how to do this for bags. Subsection 3.4 deals with sets. We recall the result of [LW93a] and explain how normalization process for sets interacts with duplicate elimination. In subsection 3.5 we consider the case when the target type is not $sk(s)$ but an intermediate type t such that $s \twoheadrightarrow t \twoheadrightarrow \langle sk(t) \rangle$. We find types t for which any object of type s would have a unique representation at type t; the process of finding such a representation is called *partial normalization.*

3.3 Normalizing bag-based complex objects

We associate the following functions with the rewrite rules:

$$\begin{array}{rcl} or\_\rho_2 & : & s \times \langle t \rangle \to \langle s \times t \rangle \\ or\_\rho_1 & : & \langle s \rangle \times t \to \langle s \times t \rangle \\ or\_\mu & : & \langle\langle t \rangle\rangle \to \langle t \rangle \\ b\_\alpha & : & \{\!|\langle s \rangle|\!\} \to \langle \{\!|s|\!\} \rangle. \end{array}$$

Here $or\_\rho_1 = or\_map((\pi_2, \pi_1)) \circ or\_\rho_2 \circ (\pi_2, \pi_1)$ is pair-with over the first argument.

Now, following [LW93a], we define the function $\mathsf{app}_b(r) : s \to t$ where r is a rewrite strategy that rewrites s to t. First assume that t is a type and p a position in the derivation tree for t such that applying a rewrite rule with associated function f to t at p yields type s. We define a function $\mathsf{app}_b(t, p, f) : t \to s$ showing the action of rewrite rules on objects by induction on the structure of t:

- if p is the root of the derivation of t, then $\mathsf{app}_b(t, p, f) = f$;
- if $t = t_1 \times t_2$ and p is in t_1 , then $\mathsf{app}_b(t, p, f) = (\mathsf{app}_b(t_1, p, f) \circ \pi_1, \pi_2)$;
- if $t = t_1 \times t_2$ and p is in t_2, then $\mathsf{app}_b(t, p, f) = (\pi_1, \mathsf{app}_b(t_2, p, f) \circ \pi_2)$;
- If p is in t', then $\mathsf{app}_b(\{\!|t'|\!\}, p, f) = b\_map(\mathsf{app}_b(t', p, f))$;
- If p is in t', then $\mathsf{app}_b(\langle t' \rangle, p, f) = or\_map(\mathsf{app}_b(t', p, f))$.

For a rewrite strategy $r := t \xrightarrow{f_1} t_1 \xrightarrow{f_2} \ldots \xrightarrow{f_n} t_n = t'$ such that the rewrite rule with associated function f_i is applied at position p_i, we extend app_b to $\mathsf{app}_b(t, t', r) : t \to t'$ by $\mathsf{app}_b(t, t', r) = \mathsf{app}_b(t_{n-1}, p_n, f_n) \circ \ldots \circ \mathsf{app}_b(t_1, p_2, f_2) \circ \mathsf{app}_b(t, p_1, f_1)$.

Theorem 3 (Normalization for bags) *For any bag-based or-object x of type t and any rewrite strategy $r : t \twoheadrightarrow \langle sk(t) \rangle$, the following holds:*

$$\mathsf{app}_b(t, \langle sk(t) \rangle, r)(x) \quad = \quad nf(x)$$

3.4 Normalizing set-based complex objects

The normalization theorem for set-based objects was proved in [LW93a], though details were not explained there. Here we give its statement that follows immediately from theorem 3.

Let r be a rewriting $t_1 \to \ldots \to t_n$ where all t_is are types from (ST). By r^{Bag} we mean the rewriting $t_1^{\mathrm{Bag}} \to \ldots \to t_n^{\mathrm{Bag}}$ of (BT) types. Note that if $t_1 \twoheadrightarrow t_n$ is in (STR), then $t_1^{\mathrm{Bag}} \twoheadrightarrow t_n^{\mathrm{Bag}}$ is in (BTR).

Theorem 4 (Normalization for sets) *For any set-based or-object x and any rewrite strategy $r : t \twoheadrightarrow \langle sk(t) \rangle$, the following holds:*

$$(\mathsf{app}_b(t^{\mathrm{Bag}}, \langle sk(t^{\mathrm{Bag}}) \rangle, r^{\mathrm{Bag}})(x^{\mathrm{Bag}}))^{\mathrm{Set}} \quad = \quad nf(x)$$

In other words, turn x into a bag-object, and apply r^{Bag} by using app_b to obtain some object y. Then $nf(x) = y^{\mathrm{Set}}$.

Note that the statement of theorem 4 is different from (and in fact stronger than) the normalization theorem in [LW93a], which stated that $(\mathsf{app}_b(t^{\mathrm{Bag}}, \langle sk(t^{\mathrm{Bag}})\rangle, r^{\mathrm{Bag}})(x^{\mathrm{Bag}}))^{\mathrm{Set}}$ does not depend on the choice of r, and defined normal forms as the result of application of any such rewriting r.

The question arises if it is possible to construct the normal form without using the bag semantics. The answer to this question is negative. To see this, define $\mathsf{app}(t, t', r)$ for set-based objects in the same way we defined app_b, but using *map* instead of *b_map* to map over sets, and using α instead of b_α.

Proposition 5 *There exist set-based objects x of type t such that for no rewriting $r : t \longrightarrow\!\!\!\!\!\rightarrow \langle sk(t)\rangle$ is $\mathsf{app}(t, \langle sk(t)\rangle, r)(x)$ the normal form of x.* $\square$

The main reason that it is impossible to express normalization by means of app in *or-$\mathcal{NRL}$* is that duplicate elimination does not commute with normalization. That is, $nf(x^{\mathrm{Set}})$ is generally different from $nf(x)^{\mathrm{Set}}$, while $nf(y^{\mathrm{Bag}})^{\mathrm{Set}} = nf(y)$. We must admit here that proposition 5 contradicts a claim made in [LW93a] that normalization does not add expressiveness to *or-$\mathcal{NRL}$*. It does not enhance *b_or-$\mathcal{NRL}$*, but does add expressive power to *or-$\mathcal{NRL}$*.

3.5 Partial normalization

Suppose that a conceptual query asks a question about possibilities that are encoded only by some of the disjunctions, and that it does not take into account other disjunctions present in a given object. Do we have to complete the normalization process to answer such a query? If a query q can be answered by having an object of type s, and we have an object x of type t such that $t \longrightarrow\!\!\!\!\!\rightarrow s$, can we find a representation of x at type s to answer q?

In this section we explain when such a partial normalization can be performed. First notice that it is not always possible. Take $x = \langle\langle\langle 1, 2\rangle, \langle 2, 3\rangle\rangle\rangle$ of type $\langle\langle\langle int\rangle\rangle\rangle$. Then $or\_\mu(x) = \langle\langle 1, 2\rangle, \langle 2, 3\rangle\rangle$ and $or\_map(or\_\mu)(x) = \langle\langle 1, 2, 3\rangle\rangle$ – these are two different objects of the same type $\langle\langle int\rangle\rangle$.

Theorem 9 below says that essentially we only have to exclude situations like this. We consider bags here; the result for sets can be readily obtained, just as theorem 4 was obtained from theorem 3.

First, we need a criterion that would check if a type s can be rewritten to t. (We did not have this problem before, as it was easy to check if $t = \langle sk(s)\rangle$.) Let $t \prec s$ mean that s is obtained from t by removing some of the or-set brackets, i.e. s has fewer disjunctions. Now we define a new relation $\lhd$ on types using the rules below.

$$\frac{}{t \lhd t} \qquad \frac{t \lhd t' \quad s \lhd s'}{t \times t' \lhd s \times s'}$$

$$\frac{t \lhd s}{\{\!|t|\!\} \lhd \{\!|s|\!\}} \qquad \frac{t \prec t' \quad t' \lhd s}{t \lhd \langle s\rangle}$$

Proposition 6 *The above rules are sound and complete for $\longrightarrow\!\!\!\!\!\rightarrow$. That is, $s \longrightarrow\!\!\!\!\!\rightarrow t$ iff $s \lhd t$.* $\square$

The last rule for $\lhd$ introduces a new variable t' instead of suggesting a proof search strategy. One might think that this leads to (at least) exponential time algorithms for verifying $s \lhd t$. (This somewhat resembles the situation with the cut rule in sequent calculus. Although it can be eliminated, the cost is a hyperexponential blow-up in the proof length, cf. [Gir87].) Fortunately, this phenomenon is not observed for our rewrite system.

Proposition 7 *There exists a linear time complexity algorithm that, given two types s and t, returns* true *if $s \longrightarrow\!\!\!\!\!\rightarrow t$ and* false *otherwise.* $\square$

Now we say that a type t is a *μ-type* if it does not have a subtype of the form $\langle\langle v\rangle\rangle$. We next define the concept of a *μ-rewriting* between μ-types. Intuitively, μ-rewritings resolve all ambiguities arising from subtypes of form $\langle\langle v\rangle\rangle$. Formally, let s and t be two distinct μ-types such that $s \longrightarrow\!\!\!\!\!\rightarrow t$. Let r be a rewriting between s and t: $s = s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_n = t$. For each $i = 0, \ldots, n-1$, let $s_i^1, \ldots, s_i^{m_i}$ be all the types such that $s_i \longrightarrow s_i^j$ (in one step) and $s_i^j \longrightarrow\!\!\!\!\!\rightarrow t$. Let p_i^j be the position in s_i at which rewrite rule is applied to obtain s_i^j from s_i, $j = 1, \ldots, m_i$.

Then the rewriting $r : s \longrightarrow\!\!\!\!\!\rightarrow t$ is a *μ-rewriting* (written as $r : s \longrightarrow\!\!\!\!\!\rightarrow_\mu t$) if either $n = 1$ (one step rewriting) or $n > 1$ and it satisfies the following two properties for every $i = 0, \ldots, n-2$:

1. If one of s_i^js is a μ-type, then s_{i+1} is a μ-type.
2. If all s_i^j have subtypes of form $\langle\langle v\rangle\rangle$, then (a) $s_{i+1} = s_i^j$ such that there is no p_i^l closer to the root than p_i^j, and (b) s_{i+2} is obtained from s_{i+1} by applying the rule $\langle\langle v\rangle\rangle \longrightarrow \langle v\rangle$ on the newly created subtype $\langle\langle v\rangle\rangle$.

This definition resolves ambiguities arising from subtypes of form $\langle\langle v\rangle\rangle$. The first property says that they need not be introduced unless absolutely necessary, and the second property dictates that once we cannot avoid introducing a subtype $\langle\langle v\rangle\rangle$, it must be done as close to the root as possible, and then gotten rid of at the next step of the rewriting. To give an example, $\langle\{\langle t\rangle\}\rangle \times s \to \langle\{\langle t\rangle\} \times s\rangle \to \langle\langle\{t\}\rangle \times s\rangle \to \langle\langle\{t\} \times s\rangle\rangle \to \langle\{t\} \times s\rangle$ is a μ-rewriting, but the one that achieves the same result by doing $\langle\{\langle t\rangle\}\rangle \times s \to \langle\langle\{t\}\rangle\rangle \times s$ first is not because introduction of the double or-set subtype can be avoided.

Proposition 8 *Let s and t be μ-types and $s \twoheadrightarrow t$. Then there exists a μ-rewriting $r : s \twoheadrightarrow_\mu t$.* □

Using this proposition, we can formulate the partial normalization theorem.

Theorem 9 (Partial Normalization) *Let s and t be μ-types such that $s \twoheadrightarrow t$. Then for any two μ-rewritings $r_1, r_2 : s \twoheadrightarrow_\mu t$ and for any object x of type s, the following holds:*

$$\mathsf{app}_b(s,t,r_1)(x) \quad = \quad \mathsf{app}_b(s,t,r_2)(x)$$

This theorem tells us that any object of a μ-type s has an unambiguous representation of a μ-type t if $s \lhd t$. This representation is obtained by applying any μ-rewrite strategy that rewrites s to t.

One may wonder if restricting rewritings to μ-rewritings only is really necessary, and if so, are both the conditions on μ-rewritings necessary. The following proposition shows that it is.

Proposition 10 *It is possible to find μ-types s and t, an object x of type s and two rewritings r_1 and r_2 from s to t which violate either the first or the second property of μ-rewritings such that $\mathsf{app}_b(s,t,r_1)(x) \neq \mathsf{app}_b(s,t,r_2)(x)$.* □

4 Normalization algorithms and primitives

There is, of course, a trivial normalization algorithm based on the general normalization theorems. We present it below for bag-based complex objects.

- If X is not an or-object, then $nf(X) = \langle X\rangle$.
- If X is (x,y) of type $s \times t$, then $nf(X) = or\_cartprod(nf(x), nf(y))$ if both s and t involve or-sets, $nf(X) = or\_\rho_1(nf(x), y)$ if only s involves or-sets and $nf(X) = or\_\rho_2(x, nf(y))$ if only t involves or-sets.
- If $X = \{\!|x_1, \ldots, x_n|\!\}$, then $nf(X) = b\_\alpha(\{\!|nf(x_1), \ldots, nf(x_n)|\!\})$.

This algorithm does calculate the normal form, as follows from theorem 3. It can be readily adapted to the set-based complex objects.

The problem with this algorithm is its exponential space complexity, as shown in [LW93a]. It creates the whole normal form before any conceptual queries can be asked. We believe it would be more reasonable to design a new evaluation strategy, that produces the elements in the normal form one-by-one. Then the space usage would be linear and, in addition, some conceptual queries can be evaluated much faster.

For example, for an existential query over a normal form, satisfiability can now be verified for each newly produced entry. If the condition is satisfied, the evaluation stops without producing all elements in the normal form. That is, if x is of type t and p is of type $sk(t) \to bool$, and we want to find out if there is an element of $nf(x)$ that satisfies p (e.g. is there a cheap reliable design?), then we should be able to stop when such an element is found. The query $\exists p$ which will be shown later in this section does precisely that. Note that using the straightforward normalization algorithm, even evaluation of $\exists(\lambda x.true)$ requires exponential space as the normal form must be produced first!

The evaluation strategy that we are going to present is essentially the depth first search on the and-or tree underlying a complex object. This strategy will work for both set- and bag- based complex objects, as sets and bags will be translated into lists to give an order of evaluation. Using this evaluation strategy, we shall also suggest new, more flexible, normalization primitives.

We create a special data structure, called *annotated complex objects*, to represent and-or trees. Basically, an annotation gives a choice of an element for each or-set and also contains local conditions telling whether all possibilities encoded by an object are exhausted. For each object type t, we have a new annotated type $A(t)$ and the initial translation $t \to A(t)$. From each annotated object, we can get an entry in the normal form. At the heart of the algorithm lies a procedure that takes an annotated object and produces the "next" one. This enables us to list all normal form entries sequentially.

We translate sets and bags into lists, assuming some ordering. No matter which ordering is chosen, the algorithm will produce all normal form entries. However, the *order* in which they are produced does depend on the translation, and can be used for additional optimizations.

In what follows, we present the algorithm for set-based complex objects. The algorithm for bag-based complex objects can be obtained by repeating it verbatim and replacing "set" by "bag". We denote the type of lists of type t by $[t]$.

Definition (Annotated complex objects). *Type K (kind) has four possible values: B (base), P (product), S (set), and O (or-set). For each type t, we produce an* annotated *type $A(t)$ as follows:*

- $A(b) = K \times b$ *if b is a base type.*
- $A(s \times t) = K \times bool \times (A(s) \times A(t))$.
- $A(\{t\}) = K \times bool \times [A(t)]$.
- $A(\langle t \rangle) = K \times bool \times [(A(t) \times bool)]$.

The boolean value in these translation is set to *true* if there are still entries encoded by the object that have not been looked at. For or-sets, the boolean component inside lists is used for indicating the element that is currently used as the choice given by that or-set. In all algorithms only one entry in such a list will have the *true* boolean component.

Now we define three functions: $initial : t \to A(t)$ produces the initial annotation of an object; $pick : A(t) \to sk(t)$ produces an element of the normal form given by an annotation; $end : A(t) \to bool$ returns *true* iff all possibilities encoded by its argument have been exhausted.

The definitions of *initial* and *pick* are given in figure 3. By *void* we mean a special object used to indicate the end of the process of going over the normal form. P1–P5 give a simplified version of *pick* in which *void* is not propagated to the top level. Such propagation is done to detect inconsistencies encoded by empty or-sets.

The function *end* always returns *true* on (B, x). On any other annotated object $x = (k, c, v)$, $end\ x = \neg c$. We also define a function $reset : A(t) \to A(t)$ that disregards the annotation of an object and restores the initial one. The definition almost verbatim repeats *initial* and is omitted here.

A recursive algorithm for *next* is given in figure 4. We use the [] brackets for lists. For any list $X = [x_1, \ldots, x_n]$, X_{oi} stands for $[x_1, \ldots, x_{i-1}]$ and X_{1i} denotes $[x_{i+1}, \ldots, x_n]$ (they may be empty). We use the notation :: and @ for consing and appending. That is, $a::x$ puts a as the new head before the list x, and $x@y$ appends y to the end of x.

Now we can produce the following algorithm that lists elements of the normal form of an or-object o.

```
Calculating norm(cond,init,update,out)(o)

   acc := init;
   ao  := initial o;
   last := end ao;
   while ¬(cond(pick ao) ∨ last)
         do
               acc := update(pick ao,acc);
               ao := next ao;
               last := end ao
         end;
   return out((pick ao,last),acc)
```

Figure 5: Algorithm for *norm*

```
ao := initial o;
repeat
      print(pick ao);
      ao := next ao
until end(ao)
```

Theorem 11 *For any or-object o, the algorithm above prints all elements of $nf(o)$ and nothing else. Moreover, it has linear space complexity.* □

Although no duplicate elimination is done in this algorithm, it does not produce unnecessary copies.

Corollary 12 *Let o be an or-object such that all or-sets in it are pairwise disjoint. Then the above algorithm prints each entry in $nf(o)$ exactly once.* □

The correctness result suggests adding new, more flexible normalization primitives to *or-$\mathcal{NRL}$*. We propose the following one called *norm*.

$$\frac{cond : sk(t) \to bool \qquad update : sk(t) \times u \to u \qquad out : (sk(t) \times bool) \times u \to s \qquad init : u}{norm(cond, init, update, out) : t \to s}$$

Its "semantics" is given by the algorithm in figure 5. Intuitively, the output value is accumulated in *acc*, *cond* is used to break the loop if the condition is satisfied, *last* indicates if all possibilities have been looked at, and *out* forms the output.

Now, a number of functions can be defined using *norm*. Here we consider just two. In the first definition, p is of type $sk(t) \to bool$.

$$\exists p \equiv norm(p, false, \lambda x.\lambda y.false, \pi_1)$$
$$normalize \equiv norm(\lambda x.false, \langle\rangle, \lambda x.\lambda y.or\_\eta(x) or\_\cup y, \pi_2)$$

I1	$\mathit{initial}\ x = (B, x)$ if x is of base type.
I2	$\mathit{initial}\ (x, y) = (P, \mathit{true}, (\mathit{initial}\ x, \mathit{initial}\ y))$.
I3	$\mathit{initial}\ \{x_1, \ldots, x_n\} = (S, \mathit{true}, [\mathit{initial}\ x_1, \ldots, \mathit{initial}\ x_n])$.
I4	$\mathit{initial}\ \langle x_1, \ldots, x_n\rangle = (O, \mathit{true}, [(\mathit{initial}\ x_1, \mathit{true}), (\mathit{initial}\ x_2, \mathit{false}), \ldots, (\mathit{initial}\ x_n, \mathit{false})])$.
I5	$\mathit{initial}\ \langle\rangle = (O, \mathit{false}, [\,])$.
P1	$\mathit{pick}\ (B, x) = x$.
P2	$\mathit{pick}\ (P, c, (x, y)) = \mathit{if}\ c\ \mathit{then}\ (\mathit{pick}\ x, \mathit{pick}\ y)\ \mathit{else}\ \mathit{void}$.
P3	$\mathit{pick}\ (S, c, [x_1, \ldots, x_n]) = \mathit{if}\ c\ \mathit{then}\ \{\mathit{pick}\ x_1, \ldots, \mathit{pick}\ x_n\}\ \mathit{else}\ \mathit{void}$.
P4	$\mathit{pick}\ (O, c, [x_1, \ldots, x_n]) = \mathit{if}\ c\ \mathit{then}\ \mathit{pick}\ \pi_1(x_i)\ \mathit{else}\ \mathit{void}$ where $\pi_2(x_i) = \mathit{true}$.
P5	$\mathit{pick}\ (O, c, [\,]) = \mathit{void}$.

Figure 3: Definitions of *initial* (I1–I5) and *pick* (P1–P5)

Base

$$\frac{}{\mathit{next}\ (B, x) = (B, x)}$$

Pair

$$\frac{\neg \mathit{end}(\mathit{next}\ y)}{\mathit{next}\ (P, c, (x, y)) = (P, \mathit{true}, (x, \mathit{next}\ y))} \qquad \frac{\mathit{end}(\mathit{next}\ y) \qquad \mathit{end}(\mathit{next}\ x)}{\mathit{next}\ (P, c, (x, y)) = (P, \mathit{false}, (x, y))}$$

$$\frac{\mathit{end}(\mathit{next}\ y) \qquad \neg \mathit{end}(\mathit{next}\ x)}{\mathit{next}\ (P, c, (x, y)) = (P, \mathit{true}, (\mathit{next}\ x, \mathit{reset}\ y))}$$

Set

$$\frac{}{\mathit{next}\ (S, c, [\,]) = (S, \mathit{false}, [\,])} \qquad \frac{\neg \mathit{end}(\mathit{next}\ x_1)}{\mathit{next}\ (S, c, X) = (S, \mathit{true}, \mathit{next}\ x_1 :: [x_2, \ldots, x_n])}$$

$$\frac{\mathit{end}(\mathit{next}\ x_1) \qquad \mathit{next}\ (S, \mathit{true}, [x_2, \ldots, x_n]) = (S, c', X')}{\mathit{next}\ (S, c, X) = (S, c', \mathit{reset}\ x_1 :: X')}$$

Or-set

$$\frac{}{\mathit{next}\ (O, c, [\,]) = (O, \mathit{false}, [\,])} \qquad \frac{\pi_2(x_i) \qquad X_{1i} = [\,] \qquad \mathit{end}(\mathit{next}\ \pi_1(x_i))}{\mathit{next}\ (O, c, X) = (O, \mathit{false}, X)}$$

$$\frac{\pi_2(x_i) \qquad X_{1i} \neq [\,] \qquad \mathit{end}(\mathit{next}\ \pi_1(x_i))}{\mathit{next}\ (O, c, X) = (O, \mathit{true}, X_{0i}\ @\ [(\pi_1(x_i), \mathit{false}), (\pi_1(x_{i+1}), \mathit{true})]\ @\ [x_{i+2}, \ldots, x_n])}$$

$$\frac{\pi_2(x_i) \qquad \neg \mathit{end}(\mathit{next}\ \pi_1(x_i))}{\mathit{next}\ (O, c, X) = (O, \mathit{true}, X_{0i}\ @\ [(\mathit{next}\ \pi_1(x_i), \mathit{true})]\ @\ X_{1i})}$$

Figure 4: Algorithm for *next*

Corollary 13 *For any or-object o, 1)* $\exists p(o) = (x, c)$ *where* x *is a normal form entry satisfying* p *if* $c =$ *false and there are no normal form entries satisfying* p *if* $c =$ *true, and 2) normalize*(o) *is its normal form.* □

Note that $\exists p$ is very useful in evaluation of existential queries. If an entry that satisfies p is found, $\exists p$ stops and returns that entry without producing all other normal form entries. In contrast to the standard algorithm that requires exponential space to evaluate such queries even if p is $\lambda x.true$, $\exists(\lambda x.true)$ needs linear time and space to be evaluated.

As another application of the new evaluation strategy, it is possible to run normalization for a given time, and get the best entry in the normal form obtained in that time. This is often helpful if an approximate solution is satisfactory.

Space-efficient evaluation of recursive queries using normalization. Now we show a somewhat surprising application of our normalization algorithm – it deals with *algorithmic* expressive power of query languages. Recall that the *Abiteboul-Beeri algebra* $\mathcal{A}\&\mathcal{B}$ [AB88] is the nested relational algebra (general and set operators in figure 2) plus the *powerset* operator. While the nested relational algebra cannot express recursive queries such as transitive closure (tc) [LW94], $\mathcal{A}\&\mathcal{B}$ can express tc by first producing all possible relations on a given set of nodes and then selecting those that contain a given one and are transitive. Of course this way of computing tc uses exponential space. A remarkable result of [SP94] says that no matter how we write an $\mathcal{A}\&\mathcal{B}$-expression to compute tc, it will use exponential space. However, it is based on a contrived restriction that a "natural" evaluation strategy is used. If this restriction is dropped, then it is possible to devise an evaluation strategy that computes tc in polynomial space, as shown in [AH95].

It was proved in [LW93a] that α has essentially the expressive power of the *powerset* operator. Hence, we can view *or-*$\mathcal{NRL}$ as an extension of $\mathcal{A}\&\mathcal{B}$ with or-sets. Now we explain how to use *norm* to compute tc space-efficiently in this language. We use some meta-notation, but everything can be expressed in *or-*$\mathcal{NRL}$.

Let $R : \{b \times b\}$ be a nonempty binary relation. Define $\mathsf{N}_R = map(\pi_1)\ R\ \cup map(\pi_2)\ R$ (the set of nodes of R) and N_R^2 as $cartprod(\mathsf{N}_R, \mathsf{N}_R)$. Now let

$$\mathsf{P}_R = map(\lambda z.or\_\cup(or\_\eta(\{\}), or\_\eta(\eta\ z)))\ (\mathsf{N}_R^2)$$

That is, for each pair of nodes (x, y), the set P_R contains an element $\langle\{\}, \{(x, y)\}\rangle$. Let $rc : \{t \times t\} \times \{t \times t\} \to \{t \times t\}$ compute the relational composition (it can be done in any language that contains relational algebra as a sublanguage). Let e be of type $b \times b$ (i.e. an edge). Define

$$c_e = \lambda S.(rc(\mu\ S, \mu\ S) = \mu\ S)\&(R \subseteq \mu\ S)\&(e \notin \mu\ S)$$

Finally, let $tc_e = norm(c_e, (), \lambda x.(), \pi_2 \circ \pi_1)(\mathsf{P}_R)$.

Proposition 14 tc_e *evaluates to* true *if* e *is in* $tc(R)$ *and it evaluates to* false *otherwise. Consequently,* $tc(R)$ *can be computed in polynomial space using norm.* □

This proposition can be regarded as a counterpart of the result of [AH95] saying that tc can be evaluated in $\mathcal{A}\&\mathcal{B}$ using polynomial space under a special evaluation strategy. Here we used our space-efficient strategy for normalization to achieve the same result.

5 Objects with partial information and antichain semantics

The antichain semantics, defined in [Lib95, LW93a] and based on the ideas from [BJO91, Lib91], is used for objects with partial information. The key idea is that the notion of partiality can be conveyed by orderings, with $x \leq y$ meaning that y is more informative than x.

This ordering is usually given for base types. For example, a null value **ni** (no information) is less informative than any integer or boolean. For pairs, $(x, y) \leq (x', y')$ iff $x \leq x'$ and $y \leq y'$. It was explained in [LW93a] that the following two orderings, well-known in semantics of concurrency [Gun92], must be used for sets and or-sets respectively:

$$X \sqsubseteq^\flat Y \quad \Leftrightarrow \quad \forall x \in X\ \exists y \in Y:\ x \leq y$$
$$X \sqsubseteq^\sharp Y \quad \Leftrightarrow \quad \forall y \in Y\ \exists x \in X:\ x \leq y$$

Using these orderings suggests a new semantics in which an object can denote any other object that is more informative. This allows elimination of redundancies given by comparable elements, because $X \sqsubseteq^\flat Y$ iff $\max X \sqsubseteq^\flat Y$ and $X \sqsubseteq^\sharp Y$ iff $\min X \sqsubseteq^\sharp Y$, where $\max X$ and $\min X$ are sets of maximal and minimal elements of X.

In $\max X$ and $\min X$ elements are pairwise incomparable. Such sets are called *antichains.* Using $\mathbb{A}_{\text{fin}}(A)$ for the family of antichains over a poset A, we define the following (structural) antichain-based semantics. Here we consider only set-based objects.

- $[\![b]\!]_a = (D_b, \leq_b)$
- $[\![t \times s]\!]_a = [\![t]\!]_a \times [\![s]\!]_a$
- $[\![\{t\}]\!]_a = (\mathbb{A}_{\text{fin}}([\![t]\!]_a), \sqsubseteq^\flat)$
- $[\![\langle t\rangle]\!]_a = (\mathbb{A}_{\text{fin}}([\![t]\!]_a), \sqsubseteq^\sharp)$

As follows from the claims above, for each object x of type t there exists a semantically equivalent object x° in $[\![t]\!]_a$ defined by the following rules:

- $x^\circ = x$ for x of a base type.
- $(x,y)^\circ = (x^\circ, y^\circ)$.
- $\{x_1,\ldots,x_n\}^\circ = \max\{x_1^\circ,\ldots,x_n^\circ\}$.
- $\langle x_1,\ldots,x_n\rangle^\circ = \min\langle x_1^\circ,\ldots,x_n^\circ\rangle$.

Consequently, for each operation $f : s \to t$ in $or\text{-}\mathcal{NRL}$, we define a new operation f_a that takes $x \in [\![s]\!]_a$ and returns $f(x)^\circ \in [\![t]\!]_a$. It is known (see [Lib92, LW93a]) that α_a is an isomorphism between $[\![\{\langle t\rangle\}]\!]_a$ and $[\![\langle\{t\}\rangle]\!]_a$. Using these operations f_a, it is possible to define $\mathsf{app}_a(t,t',r) : t \to t'$ that applies a rewrite strategy $r : t \twoheadrightarrow t'$, exactly in the same way as we defined app, but using the index a everywhere.

The following two results state the normalization theorem for the antichain semantics, and the partial normalization theorem.

Theorem 15 *Let $x \in [\![t]\!]_a$ be an object of type t such that t involves or-sets. Then, for any rewriting $r : t \twoheadrightarrow \langle sk(t)\rangle$, the following holds:*

$$\mathsf{app}_a(t, \langle sk(t)\rangle, r)(x) \quad = \quad nf(x)^\circ$$

Theorem 16 *Let s and t be two μ-types such that $s \twoheadrightarrow t$. Then for any two μ-rewritings $r_1, r_2 : s \twoheadrightarrow_\mu t$ and any $x \in [\![s]\!]_a$,*

$$\mathsf{app}_a(s,t,r_1)(x) \quad = \quad \mathsf{app}_a(s,t,r_2)(x)$$

6 Experimental results

The basic normalization algorithm and the new space efficient normalization algorithm have been implemented in the system OR-SML[1] [GL94], which is a database programming language built on top of Standard ML of New Jersey [HMT90].

We ran a number of experments to compare the speed of the basic algorithm with the new algorithm described in this paper. As our test objects, we chose objects that are known to cause exponential blow-up in the size of the normal form [LW93a]. In addition, these objects are not well suited for the OR-SML duplicate elimination algorithm [GL94], so we could compare the speed of the standard algorithms for sets and bags.

In the table below, the first column shows (approximately) the number of entires in the normal form. Entries themselves are relatively small. The second column shows running time[2] for the standard algorithm for sets; that is, at the end duplicates are eliminated. The third column is running time for the standard algorithm for bags. The last column is running time for the new algorithm. Note that we compare time rather than space. Despite its space efficiency, then new algorithm still has to compute exponentially many entries. There are several reasons why figures in the last column are better; among them is winning in time due to not running garbage collections.

# entries	time (1)	time (2)	time (3)
$> 19{,}000$	> 11min	0.9sec	1.8sec
$> 59{,}000$	> 90min	8.9sec	5.8sec
$> 175{,}000$	> 16hr	31.1sec	19.1sec
$> 525{,}000$	> 2 days	1min35sec	59sec
$> 1.5 \cdot 10^6$	not done	out of memory	3min9sec
$> 4.5 \cdot 10^6$	not done	same	9min56sec
$> 14 \cdot 10^6$	not done	same	31min51sec

We have also considered an application of the normalization algorithm where one has to select a normal form entry which is best according to some criterion F. If the normal form is large, it is possible to run the algorithm for a given time, returning the best entry that was found so far. In one of our examples, with almost 3.5 billion entries in the normal form (going over them takes about 5 days), we obatined the value of F within 7% of the optimal by running the algorithm for only 15 seconds, and the value within 4% of the optimal in 30 minutes.

7 Conclusion

In this paper we have studied various techniques for normalizing databases with disjunctive information represented by or-sets. This problem is particularly important in the areas of application such as design and planning, as well as merging databases. Queries against such databases often ask questions about possibilities encoded by the database, rather than the information that is stored there. We rigorously defined the concept of normalization for both set and bag semantics. We explained how normal forms that list all possibilities encoded by an incomplete object can be calculated. Only a limited number of operations are needed for calculation of normal forms, and the sequence in which they are applied is irrelevant for both set and bag semantics.

Since normal forms can be of size exponential in the size of the objects, we need better tools for answering conceptual queries. We demonstrated two. Partial

[1] [GL94] describes the version of OR-SML in which the primitive *norm* is not available.

[2] On SGI Challenge XL – 8 R4400 150MHz processors with 1 Gigabyte RAM.

normalization allows us to answer queries without normalizing completely. We have also designed a new space-efficient normalization algorithm.

There are immediate practical benefits of the results presented in this paper. The new space efficient algorithm has been implemented in OR-SML – a system for querying databases with disjunctions. In addition to being space efficient and faster than the standard algorithm, it allows more control over the process of normalization. This makes the normalization techniques applicable in practical problems, such as computer automated design.

Acknowledgements: Thanks to Rick Hull, Tomasz Imielinski and Kumar Vadaparty for rightfully disputing the claim in [LW93a] that the produce-all normalization is *the* way to answer conceptual queries. I am very grateful to Peter Buneman, Elsa Gunter, Jon Riecke, Val Tannen and Limsoon Wong for their comments, help and criticism, and to Anthony Kosky for a careful reading of the manuscript.

References

[AB88] S. Abiteboul, C. Beeri, On the power of languages for the manipulation of complex objects, In *Proc. of Int. Workshop on Theory and Applications of Nested Relations and Complex Objects*, Darmstadt, 1988.

[AH95] S. Abiteboul and G. Hillebrand. Space usage in functional query languages. In *LNCS 893: Proc. ICDT-95*, pages 437–454.

[AKG91] S. Abiteboul, P. Kanellakis and G. Grahne. On the representation and querying of sets of possible worlds. *TCS* 78 (1991), 159–187.

[Alb91] J. Albert. Algebraic properties of bag data types. In *VLDB-91*, pages 211–219.

[BBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. of DBPL-91*, pages 9–19.

[BBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *LNCS 646: Proc. ICDT-92*, pages 140–154.

[BDW91] P. Buneman, S. Davidson, A. Watters, A semantics for complex objects and approximate answers, *JCSS* 43(1991), 170–218.

[BJO91] P. Buneman, A. Jung, A. Ohori, Using powerdomains to generalize relational databases, *TCS* 91(1991), 23–55.

[DJ90] N. Dershowitz and J.-P. Jouannand. Rewrite systems. In: *Handbook of Theoretical Computer Science*, North Holland, 1990, pages 243–320.

[Gir87] J.-Y. Girard. *"Proofs and Types"*, Cambridge, 1987.

[GM93] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *PODS-93*, pages 49–58.

[Gun92] C. Gunter. *"Semantics of Programming Languages"*. The MIT Press, 1992.

[GL94] E. Gunter and L. Libkin. OR-SML: A functional database programming language for disjunctive information and its applications. *LNCS 856: Proc. DEXA-94*, pages 641-650.

[HMT90] R. Harper, R. Milner, and M. Tofte. *"The Definition of Standard ML"*, The MIT Press, 1990.

[IL84] T. Imielinski, W. Lipski. Incomplete information in relational databases. *J. of ACM* 31(1984), 761–791.

[INV91a] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *SIGMOD-91*, pages 288–297.

[INV91b] T. Imielinski, S. Naqvi, and K. Vadaparty. Querying design and planning databases. In *LNCS 566: DOOD-91*, pages 524–545. Springer-Verlag.

[IMV89] T. Imielinski, R. van der Meyden and K. Vadaparty. Complexity tailored design: A new methodology for database design. To appear in *JCSS*. Extended abstract in *PODS-89*.

[Lib91] L. Libkin, A relational algebra for complex objects based on partial information, In *LNCS 495: MFDBS-91*, pages 36–41.

[Lib92] L. Libkin, An elementary proof that upper and lower powerdomain constructions commute, *Bulletin of the EATCS*, 48 (1992), 175–177.

[Lib95] L. Libkin. Approximation in databases. In *LNCS 893: Proc. ICDT-95*, pages 411–424.

[LW93a] L. Libkin and L. Wong. Semantic representations and query languages for or-sets. In *PODS-93*, pages 37–48.

[LW93b] L. Libkin and L. Wong. Some properties of query languages for bags. In *DBPL-93*, Springer Verlag, 1994, pages 97–114.

[LW94] L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In *PODS-94*, pages 155–166.

[LMR92] L. Lobo, J. Minker and A. Rajasekar. *"Foundations of Disjunctive Logic Programming"*. The MIT Press, 1992.

[Rou91] B. Rounds, Situation-theoretic aspects of databases, In *Proc. Conf. on Situation Theory and Applications*, CSLI vol. 26, 1991, pages 229-256.

[SP94] D. Suciu and J. Paredaens. Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure. In *PODS-94*, pages 201–109.