Query Language Primitives for Programming with Incomplete Databases

Leonid Libkin
Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974 USA
E-mail: libkin@bell-labs.com

Abstract

We study the problem of choosing a suitable collection of primitives for querying databases with disjunctive information. Theoretical foundations for querying such databases have been developed in [11, 12]. The main tool for querying disjunctive information has come to be known under the name of normalization.

In this paper we show how these theoretical results can lead to practical languages for querying databases with disjunctive information. We discuss a collection of primitives that one may want to add to a language in order to be able to ask a variety of queries over incomplete databases (including existential and optimization queries). We describe a new practical and easily implementable technique for partial normalization, and show how to combine it with the known technique for space-efficient normalization. As the result, we demonstrate that with very little extra added to the language, one can express a variety of primitives using just one general polynomial-space iterator. We discuss some practical implications, including nondeterminism of the resulting language, and the implementation project.

1 Introduction

We study querying databases in which incomplete information is represented via disjunctions. Such databases often arise in the design and planning areas, as was first noticed in [9]. For certain objects whose values are not known at present, a database may contain a number of possible values. Choosing one possibility for each instance of disjunctive information gives us a possible world described by an incomplete database. In practical applications, most queries the user would like to ask are queries against collections of *possible worlds*, rather than the representation of those possible worlds by means of disjunctive information. That is, additional transformation of the data stored in a database is needed in order to answer such queries. The need for distinguishing two classes of queries against databases with disjunctive information is known in the literature, cf. [9, 10, 12, 16]. Queries that ask questions about the representation of possible worlds are called *structural*, whereas *conceptual* queries ask questions about the data encoded by the information in a database.

For example, consider a template used by a designer (shown in figure 1). It may indicate that part D consists of two subparts, $A1$ *and* $A2$, and $A1$ is built from $B1$ *and* $B2$ *and* $B3$, while $B1$ is a *or* b *or* c, $B2$ is d *or* e, and $B3$ is f *or* g. The subpart $A2$ has a similar structure. In figure 1, vertical and horizontal lines represent parts that must be included, while the sloping lines represent possible choices. It must be stressed that the smallest subparts shown in figure 1 may in turn have very complex structure and involve incomplete information.

With the example in figure 1 we can illustrate the difference between structural and conceptual queries. A structural query may ask about the number of possible choices for $B1$ – this information can be directly obtained from the database. Conceptual queries ask questions about possible *completed* designs. Most typically, these are *existential* queries (is there a completed design that costs less than $\$m$?) or *optimization* queries (find the most reliable design).

Complexity of conceptual queries was studied in [9, 10] and a coNP-completeness result was proved. Then tight upper bounds on the number of possible worlds encoded by databases with disjunctions were obtained in [12]. Roughly, if a database has size n, the size of the collection of possible worlds encoded by it is bounded above by $n \cdot 1.45^n$. Thus, answering conceptual queries is generally very expensive; nevertheless, they do arise in practice and one needs mechanisms for answering them.

A collection of tools for answering conceptual queries was developed in [12] and further investigated in [11]. These tools have come to be known under the name of *normalization*, and the collection of all possible worlds as *normal forms*. A normalized database is a collection of all possible worlds encoded by a database; a conceptual query is simply a structural query on a normalized database. In [12], a simple algorithm to compute normalized databases was given. However, it required exponential *space*.

That solution was refined in [11], where a polynomial-space normalization mechanism was presented. It was achieved by reusing space for possible worlds, and processing them one at a time. This requires keeping a special structure, called an *annotated object*, to indicate choices for all instances of disjunctive information in a database. A new primitive called *norm*, based on this idea, was suggested in [11]. It allows more control over the process of normalization. For example, it can stop iterating if a condition is satisfied. This has a potential of speeding up existential queries. However, the solution of [11] is still far from what we need in practical problems. There are at least two reasons for this.

- Most importantly, a programmer may want a larger collection of primitives suitable for various kinds of queries. For example, if a normal form is so large that producing all its elements in infeasible, one may want to set a time limit and attempt to find an entry either satisfying a given condition, or optimizing a criterion for a given time. Moreover, one may want a mechanism for resuming this process from the point where it was stopped. In the case of optimizing criteria over extremely large normal form, one may want to randomize this process, trying possible worlds from different "areas".

- Some of the disjunctions may not be involved in conceptual queries. For instance, in the design example above, the designer may decide that the reliability of part $A2$ is irrelevant, and try to optimize the reliability of part $A1$. In current query evaluation methods, this would involve normalizing the whole object. So if part $A2$ has a complex structure, a lot of redundant computation will be done. Thus, we need tools for *partial normalization* that avoid such unnecessary computations. The solution of [11] was based on the concept of *μ-rewriting*, which is rather hard to grasp, and therefore very hard to incorporate into a query language.

The main goal of this paper is to use the theoretical results of [11, 12] to come up with a collection of query language primitives suitable for a variety of conceptual queries against databases with disjunctive information; in particular, we want to address the shortcoming mentioned above. The main contributions are summarized below.

1. We define the concept of a *partial normal form* which represents *incomplete* possible worlds. That is, some of the disjunctions are still allowed in possible worlds. Our concept of partial normal form is less general but much more intuitive than that of [11] and can be easily incorporated into a query language.

2. We generalize the normalization mechanism in two aspects. First, we make it work with both normal forms and partial normal forms. Second, its output includes a special data structure, called an annotated object, that allows us to resume the normalization process from the point where it was stopped.

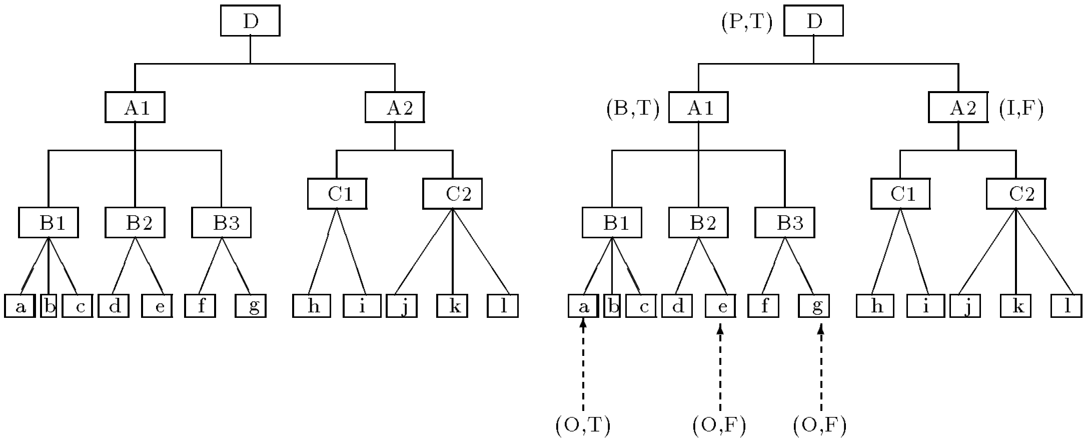

Figure 1: An incomplete database and its annotation

3. The normalization mechanism we present in this paper is suitable for extending the language with a number of primitives that are useful in various kinds of conceptual queries; moreover, as we shall show, it is easy to construct new primitives for new applications in a uniform way. For some applications, such as optimizing criteria over very large sets of possible worlds, we have to settle for operations with nondeterministic semantics. This is the price to pay for making the language more practical.

4. We briefly discuss the implementation of the operations presented in this paper. It is done as a library in OR-SML [7], the system for querying databases with disjunctive information.

Let us give a simple example to explain the gist of our approach. With each object, we associate an annotated object that indicates the choices made for each instance of disjunctive information that is relevant to the query. The idea of annotation is illustrated by the second picture in figure 1, where an arrow indicates the choice that was made. In this example we assume that a query only concerns $A1$ (for instance, what is the most reliable configuration of $A1$?). Hence, the subobject corresponding to $A2$ is not annotated.

Note that simply picking an element from each disjunctive collection is not enough to list all possible worlds, as we must also know which ones have been looked at. For this, we translate collections (bags, or multisets in this paper) into lists, and mark each subobject with a label, indicating its type and whether all possible subworlds it encodes have been looked at.

In the example in figure 1, we assume the order of elements in collections to be "from left to right". The D node receives the (P, T) label. Here P stands for "pair", and T is *true* – there are still possible worlds to look at. The label of the $A2$ node is (I, F). Here I is "initial" – we do not consider possible worlds encoded by this subobject. Hence, F (*false*) means that there are no additional objects that $A2$ may encode. The arrows point at the elements of disjunctive collections that are to be chosen. Since two arrows point to the last elements (in the lists), they are labeled by F. The key to the polynomial-space normalization is the algorithm that takes an annotation and produces the "next" one. In our example, the next annotation is produced by shifting the first arrow one position right (to point at b), and resetting two other arrows by making them point at d and f. Also, they will be labeled by T because they will no longer be pointing at the last element.

To formalize this intuitive notion of annotation, we need a formal way of distinguishing instances of disjunctive information. Our approach to representation of disjunctive information is based on [9, 12, 15]: to distinguish ordinary sets from collections of disjunctive possibilities, we call the latter *or-sets* and use $\langle\rangle$ to denote them. In the design example, $A1$ can be represented as a set or multiset $\{B1, B2, B3\}$, while $B1$ is an or-set $\langle a, b, c\rangle$. 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 language for sets and or-sets was designed in [12] and refined in [11]. We use it here as an ambient language. Note that we use the version based on *bags* (multisets) rather than sets. This is necessary because keeping duplicates is very important for the normalization process [11]. Our ambient language contains standard languages for nested bags, such as BALG [5, 6] and BQL [13, 14], as its sublanguages. To obtain the corresponding results for sets, one can use the techniques of [11] in a straightforward way, so here we only present results for bags.

Organization. We define normal forms, partial normal forms, the ambient language, and prove the generalized normalization theorem for partial normal forms in section 2. Annotated objects, space-efficient normalization algorithm and a general programming primitive for iterating over partial normal forms are presented in section 3. Extending the language with a variety of normalization primitives based on the general iterating schema is described in section 4. A brief description of the implementation project is given in section 5. Concluding remarks are given in section 6.

2 Normalization revisited

In this section we define our ambient language, the *Nested Bag–OrSet Algebra* $\mathcal{NBOA}$, and explain the concept of normalization. We also give a new definition of partial normalization that is suitable for being used in a query language, and is more intuitive than the one given in [11].

Types and Objects. Types of objects are given by the following grammar:

$$t := b \;\mid\; unit \;\mid\; t \times t \;\mid\; \{\!|t|\!\} \;\mid\; \langle t\rangle$$

Here b ranges over a collection of base types such as integers (type *int*), booleans (type *bool*) and reals (type *real*). Type *unit* has one value denoted by $()$. Values of the product type $t \times t'$ are pairs (x, y) where x has type t and y has type t'. Values of the bag type $\{\!|t|\!\}$ (or-set type $\langle t\rangle$) are finite bags (or-sets) of values of type t.

Any object containing or-sets is also called an *or-object*. Any type that uses the $\langle\rangle$ constructor is called an *or-type*. Empty or-sets $\langle\rangle$ mean inconsistency. Handling empty or-sets was discussed in [12], and we do not touch it here, assuming throughout the paper that no object contains an empty or-set subobject $\langle\rangle$.

Normal forms and partial normal forms. First, following [12], we define the rewrite system (TRS) on types:

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

We use the notation $s \xrightarrow{*} t$ if s rewrites to t in zero or more steps. A *normal form (type)* is a type that can not be rewritten any further. The *skeleton* $sk(t)$ is defined as t from which all or-set brackets have been removed. That is, $sk(b) = b$, $sk(t \times t') = sk(t) \times sk(t')$, $sk(\{\!|t|\!\}) = \{\!|sk(t)|\!\}$ and $sk(\langle t\rangle) = sk(t)$.

Lemma 1 ([12]) *The rewrite system (TRS) is Church-Rosser and terminating; hence, every type has a unique normal form. For every or-type t, $\langle sk(t)\rangle$ is its normal form.* □

Intuitively, objects of type $sk(t)$ are those encoded by objects of type t. For example, if an incomplete design is stored as an object of type t, then the completed designs represented by it have type $sk(t)$. One can also assume that certain disjunctions may still be allowed in the conceptual representation for the following reason. If a conceptual query asks only for possibilities encoded by certain disjunctions, others should not

$$\frac{}{t \lesssim t} \qquad \frac{t \lesssim s}{\{\!|t|\!\} \lesssim \{\!|s|\!\}} \qquad \frac{t_1 \lesssim s_1 \quad t_2 \lesssim s_2}{t_1 \times t_2 \lesssim s_1 \times s_2} \qquad \frac{t \lesssim s}{\langle t \rangle \lesssim s}$$

$$\frac{}{\mathsf{pnf}(t,t)} \qquad \frac{s = \langle u \rangle \quad u \neq t \quad t \lesssim u}{\mathsf{pnf}(t,s)}$$

Figure 2: Rules for $\lesssim$ and pnf

be unfolded in order to answer the query – that would be a redundant computation. Since the normalization process is very expensive, redundant computations may be too costly and may disallow some queries that are in fact answerable. To provide a mechanism for partial unfolding, we define the concept of partial normalization.

The intuition behind partial normalization is the following. We treat certain subtypes (perhaps involving or-sets) as base types and perform the usual normalization. This way those subtypes are not affected and consequently some of the disjunctions are not unfolded. To state this precisely, let $s[t/p]$ be s in which the subtype at position p is replaced by t, and let $s[t/t']$ be s in which every occurrence of the subtype t' is replaced by t. Let s_p denote the subtype of s at position p and let $b_1, b_2 \ldots$ be uninterpreted base types.

Definition. *Let s and t be two types, not involving $b_1, b_2, \ldots$. Then s is called a* partial normal form *of t if there exist $n \geq 0$ positions $p_1, \ldots, p_n$ in type t, no p_i dominating p_j, $i \neq j$, and two types s' and t' such that*

1) $t' = t[b_1/p_1, \ldots, b_n/p_n]$;
2) s' is the normal form of t';
3) $s = s'[t_{p_1}/b_1, \ldots, t_{p_n}/b_n]$.

The following diagram provides an illustration for this definition. We first replace subtypes at p_i's with b_i's, then normalize the type and then restore the subtypes at p_i's in place of b_i's. Note that a type may have more than one partial normal form, but only one normal form.

$$\begin{tikzcd}
t \arrow[r, dashed, "partial\ normal\ form"'] \arrow[d] & s = s'[t_{p_1}/b_1, \ldots, t_{p_n}/b_n] \\
t' = t[b_1/p_1, \ldots, b_n/p_n] \arrow[r, "\text{normalize}"] & s' \arrow[u]
\end{tikzcd}$$

Proposition 2 *Let a binary relation $\mathsf{pnf}(\cdot,\cdot)$ on types be defined by means of the rules in figure 2. Then $\mathsf{pnf}(t,s)$ holds iff s is a partial normal form of t. Consequently, every normal form is a partial normal form; and partial normal forms are recognizable in linear time.* □

Our next goal is to define the concepts of normal form and partial normal form on objects. Intuitively, an object x, not involving disjunctions, is in the normal form of an or-object y, written as $x \lessdot y$, iff it is in the conceptual representation of y. For partial normal forms we define the relation $x \lessdot [\![y : t, s]\!]$ meaning that x is in the conceptual representation of y of type t at type $\langle s \rangle$. That is, x of type s can be viewed as a representation of y under unfolding of those disjunctions that are to be unfolded in order to transform y

$$\textsc{Base}\quad \frac{}{x \lessdot x}\quad x \text{ of base type}$$

$$\textsc{Pair}\quad \frac{x_1 \lessdot y_1 \qquad x_2 \lessdot y_2}{(x_1, x_2) \lessdot (y_1, y_2)}$$

$$\textsc{Bag}\quad \frac{x_i \lessdot y_{\sigma(i)},\, i = 1, \ldots, n}{\{\!|x_1, \ldots, x_n|\!\} \lessdot \{\!|y_1, \ldots, y_n|\!\}}\quad \sigma \in \Sigma_n$$

$$\textsc{Or-Set}\quad \frac{x \lessdot y}{x \lessdot \langle \ldots, y, \ldots \rangle}$$

$$\textsc{Base}\quad \frac{}{x \lessdot [\![x : t, t]\!]}\quad \text{for } x : t$$

$$\textsc{Pair}\quad \frac{x_1 \lessdot [\![y_1 : t_1, s_1]\!] \qquad x_2 \lessdot [\![y_2 : t_2, s_2]\!]}{(x_1, x_2) \lessdot [\![(y_1, y_2) : t_1 \times t_2, s_1 \times s_2]\!]}$$

$$\textsc{Bag}\quad \frac{x_i \lessdot [\![y_{\sigma(i)} : t, s]\!],\, i = 1, \ldots, n}{\{\!|x_1, \ldots, x_n|\!\} \lessdot [\![\{\!|y_1, \ldots, y_n|\!\} : \{\!|t|\!\}, \{\!|s|\!\}]\!]}\quad \sigma \in \Sigma_n$$

$$\textsc{Or-Set}\quad \frac{x \lessdot [\![y : t, s]\!]}{x \lessdot [\![\langle \ldots, y, \ldots \rangle : \langle t \rangle, s]\!]}$$

Figure 3: Rules for $\lessdot$

into an object of type $\langle s \rangle$. It can also be viewed as an incomplete possible world for y. The formal definition of both versions of $\lessdot$ is given in figure 3. Σ_n denotes the group of permutations on $\{1, \ldots, n\}$.

Proposition 3 *1) Suppose that for an object y of type t and an object x there is a derivation, according to the rules of figure 3, for $x \lessdot [\![y : t, s]\!]$. Then x is of type s. Moreover, either $s = t$, or $\langle s \rangle$ is a partial normal form of t.*

2) Suppose that for some object y of type t there is a derivation for $x \lessdot y$. Then x is of type $sk(t)$. □

Definition. 1) *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 $x_i \lessdot X$.*

2) For any object X of type t, its partial normal form *over type $\langle s \rangle$, $pnf(X, s)$ is defined as the or-set of all x of type s such that $x \lessdot [\![X : t, s]\!]$.*

Note that $nf(X)$ and $pnf(X, s)$ are always finite. Furthermore, $nf(X)$ can be alternatively defined as $pnf(X, sk(t))$ if the or-object X is of type t.

Ambient language and normalization theorems. Normalization theorems provide us with a list of operations that can be applied to an object until the normal form is produced. We need a language that contains these operations. We adopt the framework of [12] based on [2, 3]. The operators and their most general types are given in figure 4.

Semantics. For general operations: $f \circ g$ is function composition; (f, g) is pair formation; π_1 and π_2 are the first and the second projections; ! always returns the unique element of type *unit*; *eq* is equality test; *id* is the identity and *cond* is conditional. For bag operations: *b_empty* is the function that represents the constant $\{\!||\!\}$; *b_sng* forms singletons: $b\_sng(x) = \{\!|x|\!\}$; $\uplus$ takes additive union of two bags; *b_flat* flattens bags of bags, adding up multiplicities: $b\_flat(\{\!|\{\!|1, 2|\!\}, \{\!|2, 3|\!\}|\!\}) = \{\!|1, 2, 2, 3|\!\}$; $b\_map(f)$ applies f to all elements of a bag; and b_pair_2 is pair-with: $b\_pair_2(1, \{\!|2, 3|\!\}) = \{\!|(1, 2), (1, 3)|\!\}$. Operators on or-sets are exactly the same as

General operators

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

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

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

Operators on bags

$$\frac{}{b\_empty : unit \to \{\!|t|\!\}} \qquad \frac{}{b\_pair_2 : s \times \{\!|t|\!\} \to \{\!|s \times t|\!\}}$$

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

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

Operators on or-sets

$$\frac{}{or\_empty : unit \to \langle t \rangle} \qquad \frac{}{or\_pair_2 : s \times \langle t \rangle \to \langle s \times t \rangle}$$

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

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

Interaction

$$\frac{}{combin : \{\!|\langle t \rangle|\!\} \to \langle \{\!|t|\!\} \rangle}$$

Figure 4: Operators of $\mathcal{NBOA}$

operators on bags except that the prefix *or* is used, and duplicates are eliminated.

Finally, *combin* provides interaction between bags and or-sets. Let $\mathcal{X} = \{\!|X_1, \ldots, X_n|\!\}$ where $X_i = \langle x^i_1, \ldots, x^i_{n_i} \rangle$. Let $\mathcal{F}$ be the family of "choice" functions from $\{1, \ldots, n\}$ to $\mathbb{N}$ such that $1 \leq f(i) \leq n_i$ for all i. Then

$$combin(\mathcal{X}) = \langle \{\!|x^i_{f(i)} \mid i = 1, \ldots, n|\!\} \ \mid\ f \in \mathcal{F} \rangle$$

For example, $combin(\{\!|\langle 1, 2\rangle, \langle 2, 3\rangle|\!\}) = \langle \{\!|1, 2|\!\}, \{\!|1, 3|\!\}, \{\!|2, 2|\!\}, \{\!|2, 3|\!\} \rangle$.

It was suggested in [12] to assign functions in the language to the rewrite rules so that for every rewriting from t to s there would be an associated definable function of type $t \to s$. The goal of this assignment is to obtain a function of type $t \to \langle sk(t) \rangle$ that produces the normal forms for or-objects of type t.

We associate the following functions with the rewrite rules:

$$\begin{array}{rcl} or\_pair_2 & : & s \times \langle t \rangle \to \langle s \times t \rangle \\ or\_pair_1 & : & \langle s \rangle \times t \to \langle s \times t \rangle \\ or\_flat & : & \langle\langle t \rangle\rangle \to \langle t \rangle \\ combin & : & \{\!|\langle s \rangle|\!\} \to \langle \{\!|s|\!\} \rangle \end{array}$$

Here $or\_pair_1 = or\_map((\pi_2, \pi_1)) \circ or\_pair_2 \circ (\pi_2, \pi_1)$ is pair-with over the first argument. It is possible to define the function $\mathsf{app}(r)$ that applies rewrite rules to objects using the above functions. For example, applying the rewriting $r = \{\!|\{\!|\langle s \rangle|\!\}|\!\} \to \{\!|\langle \{\!|s|\!\} \rangle|\!\}$ yields the function $b\_map(combin)$. This function can be extended to rewrite strategies by composition. (Technical details of the definitions can be found in [11, 12].)

The following result is new. The normalization theorems of [11, 12] can be seen as its corollaries.

Theorem 4 (Partial Normalization) *For any or-object x of type t, any type $\langle s\rangle$ which is a partial normal form of t and any rewrite strategy $r : t \xrightarrow{*} \langle s\rangle$, the following holds:* $\mathsf{app}(r)(x) \;=\; pnf(x, s)$.

Corollary 5 (Normalization [11]) *For any or-object x of type t and any rewrite strategy $r : t \xrightarrow{*} \langle sk(t)\rangle$, the following holds:* $\mathsf{app}(r)(x) \;=\; nf(x)$.

3 Annotations and polynomial-space normalization

In this section we extend the polynomial-space normalization of [11] to partial normal forms. The idea of the polynomial-space normalization is similar in the spirit to that of the "pipeline" evaluation of queries in the powerset algebra of Abiteboul and Beeri, see [1]. Note that combining polynomial-space normalization primitives and partial normalization was an open problem mentioned in [11].

As the first step, we introduce *annotated types* τ. An annotated type denotes an *and-or tree* underlying an or-object, and it indicates a choice of element for certain or-sets. Using these choices in places of or-sets, we obtain elements of partial normal forms, or, if the choice is specified for all or-sets, elements of normal forms.

Annotated types are given by the grammar

$$\tau := K \times t \;\mid\; K \times bool \times \tau \times \tau \;\mid\; K \times bool \times [\tau] \;\mid\; K \times bool \times [\tau \times bool]$$

Here K is a type that has four possible values: I (Initial case), P (Pair), B (Bag) and O (Or-set); t is an object type, and $[\tau]$ is the type of lists of type τ.

For each pair of types t and s, for which $\mathsf{pnf}(t, \langle s\rangle)$ holds, we produce an annotated type $A(t, s)$ as explained below. First though, we treat the simplified case in which s is the skeleton of t (i.e. $\langle s\rangle$ is the normal form of t). Then we use the notation $A(t)$. The translation is given by the following inductive rules:

- $A(b) = K \times b$
- $A(t_1 \times t_2) = K \times bool \times A(t_1) \times A(t_2)$
- $A(\{\!|t|\!\}) = K \times bool \times [A(t)]$
- $A(\langle t\rangle) = K \times bool \times [A(t) \times bool]$

The boolean value is *true* if not all entries encoded by the object have 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.

For any or-type t, $A(t, s)$ is defined if and only if $\langle s\rangle$ is a partial normal form of t. The idea of annotation is the same as above, except that some subtypes (maybe involving or-sets) are treated as base types and are not annotated. The positions of those subtypes in t are determined by s; they are precisely the subtypes whose disjunctions are not to be unfolded in the process of normalization. The annotated types $A(t, s)$ are defined by the following rules, which are applied in the order in which they are given below.

- $A(t, t) = K \times t$
- $A(t_1 \times t_2, s_1 \times s_2) = K \times bool \times A(t_1, s_1) \times A(t_2, s_2)$
- $A(\{\!|t|\!\}, \{\!|s|\!\}) = K \times bool \times [A(t, s)]$
- $A(\langle t\rangle, s) = K \times bool \times [A(t, s) \times bool]$

Proposition 6 *If t is an or-type, $t \neq s$, and $A(t, s)$ is defined, then $\langle s\rangle$ is a partial normal form of t.* □

Objects of type $A(t, s)$ can be seen as *and-or* trees underlying or-objects, such that selection of possibilities for all *or*-nodes gives us a complex object in the partial normal form. Hence, for evaluation of conceptual queries, we need mechanisms for a) translating or-objects into annotated objects, b) obtaining (partial) normal form entries encoded by an annotation, and, most importantly, c) iterating over all possible annotations.

The solution proposed in [11] can be readily adapted here. Moreover, the iteration mechanism remains unchanged for partial normalization. We need three functions. The first, $init_s : t \rightarrow A(t,s)$, produces the initial annotation of an object, provided $A(t,s)$ is defined. It is given by the following rules:

- $init_s\ x = (I, x)$ if x is of type s.
- $init_{s_1 \times s_2}\ (x, y) = (P, true, (init_{s_1}\ x, init_{s_2}\ y))$.
- $init_{\{\!|s|\!\}}\ \{\!|x_1, \ldots, x_n|\!\} = (B, true, [init_s\ x_1, \ldots, init_s\ x_n])$.
- $init_s\ \langle x_1, \ldots, x_n \rangle = (O, true, [(init_s\ x_1, v_1), \ldots, (init_s\ x_n, v_n)])$,

where $v_1 = false$ and $v_2 = \ldots = v_n = true$.

The function $pick : A(t,s) \rightarrow s$ produces an element of the partial normal form given by an annotation. In the definition below, $void$ indicates the end of traversing an annotated object, i.e., all possibilities have been looked at.

- $pick\ (I, x) = x$.
- $pick\ (P, c, (x, y)) = if\ c\ then\ (pick\ x, pick\ y)\ else\ void$.
- $pick\ (B, c, [x_1, \ldots, x_n]) = if\ c\ then\ \{\!|pick\ x_1, \ldots, pick\ x_n|\!\}\ else\ void$.
- $pick\ (O, c, [x_1, \ldots, x_n]) = if\ c\ then\ pick\ \pi_1(x_i)\ else\ void$ if $\pi_2(x_i) = true$.

Finally, $end : A(t,s) \rightarrow bool$ returns $true$ iff all possibilities encoded by its argument have been exhausted: $end\ (I, x) = true$, and on any annotated object $x = (k, c, v)$, $end\ x = \neg c$.

The key part of the normalization algorithm is the iterator $next : A(t,s) \rightarrow A(t,s)$ which provides the depth first search on the *and-or* trees, obtaining all possible annotations (given by the positions of the boolean components in lists encoding or-sets). The version of [11] has type $A(t, \langle sk(t) \rangle) \rightarrow A(t, \langle sk(t) \rangle)$ but it can be easily modified to produce the one of type $A(t,s) \rightarrow A(t,s)$. Also, $next$ can be implemented in a purely functional language.

Now we can show that starting with $init_s\ (o : t)$ and repeatedly applying $next$ to it, we obtain annotations for all elements in $pnf(o, s)$.

Theorem 7 *Let o be an or-object of type t and $\langle s \rangle$ be a partial normal form of t. Let $ao = init_s(o)$. Consider the following algorithm:*

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

Then this algorithm prints precisely all elements of $pnf(o, s)$. Moreover, the algorithm has linear space complexity. □

To produce annotated objects, bags and or-sets are translated into lists assuming some order on their elements. Theorem 7 says that this order is irrelevant: all entries in partial normal forms are produced. However, the order in which they are produced depends on the way bags and or-sets are translated into lists.

Based on theorem 7, we propose the following general iterator, called $apnorm^{cont}$. First, we need some abbreviations. By $\Phi(s, u, v)$ we mean the following collection of assertions:

$$\begin{array}{rcl} initial\_acc & : & u \\ condition & : & s \rightarrow bool \\ update & : & s \times u \rightarrow u \\ out & : & (s \times bool) \times u \rightarrow v \end{array}$$

By $\mathcal{P}$ we shall mean the quadruple $(initial\_acc, condition, update, out)$. Then we introduce a new primitive, which is our basic normalization iterator.

$$\frac{\Phi(s,u,v) \qquad \mathsf{pnf}(t,\langle s\rangle)}{apnorm^{cont}(\mathcal{P}) : A(t,s) \to v \times A(t,s)}$$

The algorithm for calculating $apnorm^{cont}(\mathcal{P})(ao)$ is shown below.

```
acc := initial_acc;    last := end ao;
while ¬(condition(pick ao) ∨ last)
  do
       acc := update(pick ao,acc);
       ao := next ao;
       last := end ao
  end
return (out((pick ao,last),acc), ao)
```

Intuitively, during the course of iteration over the partial normal form, the output value is accumulated in *acc*, *condition* breaks the loop when it becomes true, *last* indicates if all possibilities have been looked at, and *out* forms the output. The output also includes the last annotation that was processed (either the one on which *condition* was satisfied, or the last one produced by *next*); hence *"cont"* (continuation) in the superscript.

Before we show how a number of desirable primitives can be added to the language if $apnorm^{cont}$ is present, let us explain how we view the problem of adding this primitive to the language. The annotated types do not belong to the type system of $\mathcal{NBOA}$. We propose to extend the language with annotated types. However, the programmer is only allowed to use a very limited number of operations that deal with annotated objects, nand (s)he is not allowed to "see" them. Annotated objects can be produced from the usual or-objects using *init*, *pick* can be applied to them to obtain usual complex objects, and they can be used as inputs and outputs of primitives such as $apnorm^{cont}$. Since types themselves can not be manipulated with in our language, the family of functions $init_s$ is implemented as one function of type $t \times s \to A(t,s)$, where an additional object is needed only to supply the correct type (the size of this object can be made linear in the size of the type). It is expected that in a more user-friendly implementation of the language than the current one, the user will be allowed to supply the type instead of an object of that type to specify partial normal forms.

4 Extending the language

In this section we show how a number of desirable normalization primitives mentioned in the Introduction can be obtained if $apnorm^{cont}$ is present in the language. We divide these primitives into four groups.

It is known that there exists a calculus version of $\mathcal{NBOA}$, see [3, 12], in which expressions denote objects and not functions. This equally expressive version of the language allows the standard *if-then-else* construct, as well as using λ-abstraction to specify the function argument of *b_map* and *or_map*. In this section, we shall use both *if-then-else* construct and λ-abstraction. However, this does not enrich expressiveness of the language.

General normalization primitives

PARTIAL NORMALIZATION, STARTING WITH AN ANNOTATION. For operations in this group, we require presence of *init*. For our first operation, the idea is the same as for the general partial normalization: we start with an annotation and iterate over all partial normal form entries, producing the result. It is typed and defined as follows:

$$\frac{\Phi(s,u,v) \qquad \mathsf{pnf}(t,\langle s\rangle)}{apnorm(\mathcal{P}) : A(t,s) \to v} \qquad apnorm = \pi_1 \circ apnorm^{cont}$$

Partial normalization, parameterized by types. The idea is the same as above, but no annotated objects are involved. Instead, these primitives are parameterized by types of partial normal forms.

$$\frac{\Phi(s,u,v) \qquad \mathsf{pnf}(t,\langle s\rangle)}{pnorm_s(\mathcal{P}) : t \to v} \qquad pnorm(\mathcal{P}) \ = apnorm(\mathcal{P}) \ \circ \ init_s$$

Standard normalization. Given an object, iterate over its normal form, checking for condition and accumulating the result. This is the *norm* primitive of [11]. It is simply $pnorm_{sk(t)}(\mathcal{P})$.

Normalization with time constraints

Large sizes of normal forms can make iterating over them impractical. Then it is reasonable to set up a time limit for the normalization process to run, and return the result obtained so far, and an annotated object, so that the process of normalization can be resumed. To allow this, we use primitives of Standard ML of New Jersey [8] and define a new type *timer* and two functions: $start\_timer : unit \to timer$ starts a new timer, and $get\_time : timer \to real$ gives the time that passed since the timer was initiated. We also use the $\underline{\mathsf{let}}\ldots\underline{\mathsf{in}}\ldots\underline{\mathsf{end}}$ construct for local declarations, see [8].

Partial normalization with time constraints. The normalization process starts from an annotated object and runs for a given time, returning the result formed by *out*, and the last annotation processed. The typing and a definition based on $apnorm^{cont}$ are as follows:

$$\frac{\Phi(s,u,v) \qquad \mathsf{pnf}(t,\langle s\rangle)}{apnorm^{time}(\mathcal{P}) : A(t,s) \times real \to v \times A(t,s)}$$

$$\begin{array}{l} apnorm^{time}(\mathcal{P})(ao,T) \quad = \quad \underline{\mathsf{let}}\ tm = start\_timer() \\ \quad \underline{\mathsf{in}} \quad apnorm^{cont}(\mathcal{P}[\lambda x.condition(x) \ \vee \ (get\_time(tm) > T)/condition])\ ao \\ \quad \underline{\mathsf{end}} \end{array}$$

Partial normalization with time constraints, parameterized by types. Instead of an arbitrary annotation, we start with the initial one. Such a family of functions $pnorm_s^{time}(\mathcal{P}) : t \times real \to v \times A(t,s)$ is defined by $pnorm_s^{time}(\mathcal{P})(o,T) \ = \ apnorm^{time}(\mathcal{P})(init_s(o),T)$. The full normalization with time limit $norm^{time}(\mathcal{P}) : t \times real \to v \times A(t)$ is then simply $pnorm_{sk(t)}^{time}(\mathcal{P})$.

Optimization primitives

Often one has to find a (partial) normal from entry which is best according to some criteria (e.g., the most reliable design). For this we need the optimizing version of normalization primitives. Now by P we denote the pair $(F, \leq_F)$, and $\Psi(s,v)$ is the abbreviation for

$$\Psi(s,v) \qquad\qquad F : s \to v \qquad \leq_F : v \times v \to bool$$

Here F is the criterion to be maximized with respect to the comparison function $\leq_F$. The main operator we use for this purpose is *opt_apnorm*:

$$\frac{\Psi(s,v) \qquad \mathsf{pnf}(t,\langle s\rangle)}{opt\_apnorm(\mathsf{P}) : A(t,s) \to s \times v}$$

The semantics is the following: starting with a given annotation, look at all annotations that can be obtained from it by applying *next*, and return the one with the maximal value of F, together with that F-value:

$$\begin{array}{l} \mathit{opt\_apnorm}(\mathsf{P})\ ao \\ = \mathit{apnorm}(\mathcal{P}[\ \lambda x.\mathit{false}/\mathit{condition}, \\ \qquad (\mathit{pick}\ ao, F(\mathit{pick}\ ao))/\mathit{initial\_acc}, \\ \qquad \lambda x.\lambda y.\mathit{if}\ \leq_F (F(y), \pi_2(x))\ \mathit{then}\ x\ \mathit{else}\ (y, F(y))/\mathit{update}, \\ \qquad \pi_2/\mathit{out} \qquad\qquad])\ ao \end{array}$$

We can also define $\mathit{opt\_pnorm}_s(\mathsf{P}) : t \to s \times v$ that optimizes F for all elements of the partial normal form of type $\langle s \rangle$ (as $\mathit{opt\_pnorm}_s(\mathsf{P}) = \mathit{opt\_apnorm} \circ \mathit{init}_s$), and $\mathit{opt\_norm}(\mathsf{P}) : t \to \mathit{sk}(t) \times v$ which optimizes F over the normal form (as $\mathit{opt\_pnorm}_{\mathit{sk}(t)}(\mathsf{P})$).

Optimization with time constraints

We present two functions that evaluate optimization queries under time constraints. The first one starts with an annotated object, and uses *next* to produce new annotations for the time specified by a time limit. When the time has run out, it returns the best partial normal form entry found so far and the last annotation.

$$\frac{\Psi(s, v) \qquad \mathsf{pnf}(t, \langle s \rangle)}{\mathit{opt\_apnorm}^{\mathit{time}}(\mathsf{P}) : A(t, s) \times \mathit{real} \to (s \times v) \times A(t, s)}$$

$$\begin{array}{l} \mathit{opt\_apnorm}(\mathsf{P})\ (ao, T)\ = \\ \underline{\mathsf{let}}\ \mathit{tm} = \mathit{start\_timer()} \\ \underline{\mathsf{in}}\ \mathit{apnorm}^{\mathit{cont}}(\mathcal{P}[\lambda x.\mathit{get\_timer}(\mathit{tm}) > T/\mathit{condition}, \\ \qquad (\mathit{pick}\ ao, F(\mathit{pick}\ ao))/\mathit{initial\_acc}, \\ \qquad \lambda x.\lambda y.\mathit{if}\ \leq_F (F(y), \pi_2(x))\ \mathit{then}\ x\ \mathit{else}\ (y, F(y))/\mathit{update}, \\ \qquad \pi_2/\mathit{out} \qquad\qquad])\ ao \\ \underline{\mathsf{end}} \end{array}$$

Optimizing criteria with time constraints and random annotations. There is a more intersting approach to optimizing criteria on very large normal forms, when it is not feasible to calculate the value of F for each normal form entry. Indeed, the simple time limit approach may not be sufficient, because optimal values may be "very far" from a given annotation in terms of the number of times *next* must be applied. Then, we believe (and experimental results confirm this), the right approach is to generate randomly a number of annotations and run the optimizing version of normalization from each of them for a given time. At the end, the best entry that was found is returned. This solution is given by the function

$$\frac{\Psi(s, v) \qquad \mathsf{pnf}(t, \langle s \rangle)}{\mathit{opt\_pnorm}_s^{\mathit{rand}}(\mathsf{P}) : t \times \mathit{real} \times \mathit{int} \to s \times v}$$

The semantics is as follows. On the argument $(o : t, T : \mathit{real}, n : \mathit{int})$, the following operation is performed n times: a random annotation for o is generated, such that applying *pick* to it produces an object of type s. Then, from this annotation, we generate new ones (repeatedly using *next*), until the last one is produced, or the time limit T is reached, returning the best one with respect to F. Having done this, we have n pairs of type $s \times v$ of objects of type s and values of F on them. The result of $\mathit{opt\_pnorm}^{\mathit{rand}}(\mathsf{P})$ is the best one with respect to F.

The function $\mathit{opt\_pnorm}_s^{\mathit{rand}}$ is implementable using the basic iteration mechanism and three auxiliary functions:

$$\frac{\mathsf{pnf}(t, \langle s \rangle)}{\mathit{random}_s : t \to A(t, s)} \qquad \frac{\Psi(s, v)}{\mathit{select\_best}(\mathsf{P}) : \{\!|s|\!\} \to s} \qquad \frac{}{\mathit{gen} : \mathit{int} \to \{\!|\mathit{int}|\!\}}$$

The first one produces a random annotation of type $A(t, s)$. The second, *select_best*, selects the best element from a bag of type $\{\!|s|\!\}$ with respect to the criterion F. It is undefined on empty bags, and, if more than one element of a bag have nondominated F-values, selects one nondeterministically. The semantics of *gen* is given by $gen(n) = \{\!|1, \ldots, n|\!\}$ (this function plays an important role in establishing equivalences between set and bag languages with structural recursion and power operators [13, 14]).

Now opt_pnorm^{rand} is defined in two steps. First, we define one iteration step

$$iter\_opt\_pnorm_s^{rand}(\mathsf{P})(o, T) = opt\_apnorm_s^{time}(\mathsf{P})(random_s(o), T)$$

and then

$$\begin{array}{l} opt\_pnorm_s^{rand}(\mathsf{P})(o, T, n) \ = \\ \quad if\ n \leq 1\ then\ iter\_opt\_pnorm_s^{rand}(\mathsf{P})(o, T) \\ \quad else\ select\_best(\pi_2, \leq_F)(b\_map(\lambda x.iter\_opt\_pnorm_s^{rand}(\mathsf{P})(o, T))(gen(n))) \end{array}$$

Summing up, to obtain the list of desirable normalization primitives, we do not have to add them all to the language. Instead, it is enough to have one general iteration scheme $apnorm^{cont}$ and a limited number of auxiliary functions. In this way it is easy to add new variations of normalization primitives.

5 Implementation project

The collection of normalization primitives discussed in this paper has been implemented as a library in the system OR-SML [7], which itself is a database programming language on top of Standard ML of New Jersey [8]. In OR-SML, complex objects are SML-values, and one can take advantage of combining the features of a query language with the features of a fully-fledged programming language. For example, we can use SML library that provides objects of type *timer* and functions on them to express time-constrained normalization primitives in the same way as it is done in section 4.

For the extended abstract, we mention just one experimental result. If a normal form is very big, optimizing a criterion over it may take weeks. Using a time limit, we may not reach a good result. In the example of [11], a criterion was optimized for 30 minutes, and the result within 4% of the optimal was produced. However, using the function opt_pnorm^{rand}, we can see entries in different "areas" of the normal form. In fact, in the example from [11], using 10 iterations, each running 30 seconds (for the total of only 5 minutes), we consistently obtained results within 0.5% of the optimal.

6 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. Most of previous work provided foundations for asking queries against such databases. However, proposed solutions were impractical, mostly because of their complexity.

In this paper we took advantage of the polynomial-space normalization iterator, proposed in [11], and extended the idea behind it. As the result, we came up with a number of query language primitives that can help answer a variety of conceptual queries. In fact, all that must be added to the language is one general iterator and a small number of auxiliary functions. The resulting variants of normalization are suitable for various kinds of conceptual queries. In addition, they provide mechanism for answering queries approximately, which is very helpful when one has to optimize some criteria over extremely large number of encoded objects. In order to obtain such approximate solution, we often have to settle for nondeterministic operations, which limits our ability to reason about the resulting language. This is the price to pay for making the language applicable in practice.

Summing up, we believe that using techniques of this paper can provide good practical algorithms for dealing with large applications involving databases with disjunctive information.

Acknowledgements: I thank Peter Buneman for his comments on [11] that influenced this paper, Jon Riecke for a careful reading of an earlier draft, Lal George for the ML code for the random number generator, and anonymous referee for many helpful suggestions.

References

[1] S. Abiteboul and G. Hillebrand. Space usage in functional query languages. In *Proceedings International Conference on Database Theory, 1995*, Springer LNCS 893, pages 437–454.

[2] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of Database Programming Languages 1991*, Morgan Kaufmann, 1991, pages 9–19.

[3] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Proceedings International Conference on Database Theory, 1992*, Springer LNCS 646, pages 140–154.

[4] P. Buneman, A. Jung, A. Ohori, Using powerdomains to generalize relational databases, *Theoretical Computer Science* 91(1991), 23–55.

[5] S. Grumbach and T. Milo. Towards tractable algebras for bags. In *Symp. Principles of Database Systems*, ACM Press, 1993, pages 49–58.

[6] S. Grumbach, T. Milo and Y.Kornatzky. Calculi for bags and their complexity. In *Proceedings of Database Programming Languages 1993*, Springer Verlag, 1994, pages 65–79.

[7] E. Gunter and L. Libkin. OR-SML: A functional database programming language for disjunctive information and its applications. *Proc. Database and Expert Systems Applications, 1994*, Springer LNCS 856, pages 641-650.

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

[9] T. Imielinski, S. Naqvi, and K. Vadaparty. Incomplete objects — a data model for design and planning applications. In *Proc. Int. Conf. on Management of Data (SIGMOD-91)*, pages 288–297.

[10] T. Imielinski, R. van der Meyden and K. Vadaparty. Complexity tailored design: A new design methodology for databases with incomplete information. *Journal of Computer and System Sciences*, 51(1995), 405–432.

[11] L. Libkin. Normalizing incomplete databases. In *Symp. Principles of Database Systems 1995*, ACM Press, pages 219–230.

[12] L. Libkin and L. Wong. Semantic representations and query languages for or-sets. *Journal of Computer and System Sciences*, 52(1996), 125–142.

[13] L. Libkin and L. Wong. Some properties of query languages for bags. In *Proceedings of Database Programming Languages 1993*, Springer Verlag, 1994, pages 97–114.

[14] L. Libkin and L. Wong. New techniques for studying set languages, bag languages and aggregate functions. In *Symp. Principles of Database Systems 1994*, ACM Press, pages 155–166.

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

[16] K. Vadaparty and S. Naqvi. Using constraints for efficient query processing in non-deterministic databases. *IEEE Trans. Knowledge and Data Eng.*, 7(1995), 850–864.