INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians
Universität
München

Indefinite Information with a Data Model Based on Algebraic Datatypes

Heribert Schütz

http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-19, September 1997

Indefinite Information with a Data Model Based on Algebraic Datatypes

Heribert Schütz

Universität München, Institut für Informatik, Oettingenstr. 67, D-80538 München
Heribert.Schuetz@informatik.uni-muenchen.de

Abstract. This paper investigates an approach to the management of indefinite information in databases with a data model based on algebraic datatypes, like the ones used in modern functional programming languages such as ML and Haskell. A (definite) value of an algebraic datatype is a tree and an indefinite value can be seen as a set of trees (a "forest"). The true value is known to be a member of the forest but it is not known which one it is.
The paper considers a class of possibly infinite forests that can be represented by finite tree automata. It investigates which operations for definite values can be extended to indefinite values in such a way that the result of the operation again belongs to the representable class. In order to provide some comparability to approaches for the representation of indefinite information in relational databases, the focus is on such an extension for emulated relational algebra operations.

1 Introduction

A database describes some part of the world. If that part is entirely known and described in some database state, then we speak of a *definite* database state. If it is not entirely known, it may still be possible to store the information that we have in a database. We speak of an *indefinite* database state in this case. The incomplete information in an indefinite database state corresponds to several possible states of the relevant part of the real world, each of which can be described by a definite database state. So we can view an indefinite database state as (a representation of) a set of definite database states.

An ad-hoc way to deal with indefinite information is to reify the meta-knowledge (i.e., the knowledge about what is known) and to model it in the data model provided by the database system. One disadvantage of this approach is that it is the user's responsibility to invent a reified representation that reflects the possible degrees of indefiniteness. Another disadvantage is that updates and queries also have to be adapted to the chosen reification, which typically makes them far more complex. Therefore it is desirable that the database management system supports the storage of indefinite information. Then it is, e.g., possible to formulate a query to an indefinite database in the same way as it would be formulated to a definite database. The computed answer will then span a range of possibilities.

Of course it is usually not desirable to simply enumerate all the possibilities allowed by an indefinite database, especially if there are infinitely many of them. One rather wants a more abstract and compact representation. Several approaches for representing incomplete information in databases have been investigated in the literature. The most well-known tool for this purpose are null-values (e.g., [1,4,6,3]).

In this paper a new technique for modelling indefinite information is investigated. It works with a data model based on algebraic datatypes, like the ones used in modern functional programming languages such as ML and Haskell. A (definite) value of an algebraic datatype is a tree and an indefinite value is a set of trees (a "forest").

It is not possible nor useful to represent arbitrary such forests. The paper therefore considers a restricted class of forests. These forests can be represented by finite tree automata. The forests themselves may nevertheless become infinite. It will also be investigated which operations for definite values can be extended to indefinite values in a sensible way. In order to provide some comparability to approaches for the representation of indefinite information in relational databases, we will see how the relational data model can be emulated with algebraic datatypes and how the indefiniteness carries over to the relational model.

The paper is organized as follows: In the next section the basic notions are introduced. Section 3 explains how the relational data model can be emulated by algebraic types. Section 4 investigates for various types of indefiniteness whether they can be represented with our model. Section 5 investigates which operations on database states are possible in our model. Section 6 concludes the paper.

2 Preliminaries

2.1 Algebraic Datatypes

A family of algebraic datatypes is given by a *signature* which consists of

- a finite set of *type names* and
- a finite set of *constructor symbols*, each associated with a type $\tau_1 \times \ldots \times \tau_n \to \tau$ $(n \geq 0)$, where $\tau_1, \ldots, \tau_n, \tau$ are type names.

We say that a constructor with type $\tau_1 \times \ldots \times \tau_n \to \tau$ is an n-ary constructor for the type τ. The type names $\tau_1, \ldots, \tau_n$ are called the argument types of the constructor. We denote the association of a constructor with a type by $c : \tau_1 \times \ldots \times \tau_n \to \tau$.

Given a set of variables where each variable is associated to a type name we can construct *terms* by a finite number of applications of these rules:

- A variable of type τ is a term of type τ.
- If c is a constructor symbol of type $\tau_1 \times \ldots \times \tau_n \to \tau$ and $t_1, \ldots, t_n$ are terms of types $\tau_1, \ldots, \tau_n$, respectively, then $c(t_1, \ldots, t_n)$ is a term of type τ.

Ground terms, i.e., terms without variables, will also be called *trees*. For a tree we will say that the nodes are labeled by the respective constructor symbols. A set of trees is called a *forest*.

Note that we do not consider infinite trees as in Haskell, but the finite trees can become arbitrarily deep. We also do not consider parameterized types.

2.2 Tree Automata

A finite tree automaton accepts or rejects a tree in a way similar to a finite automaton accepting or rejecting a string. We will consider a certain kind of finite tree automata, namely finite deterministic bottom-up tree automata. Since this is the only kind of automata that will be used in this paper, we will simply speak of automata. For a given signature Σ and a type name τ_0 from Σ such an automaton is defined as a tuple (S, δ, S_{acc}) where

- S is a family of finite sets S_τ of *states* for every type name τ from Σ.
- δ is a family of functions δ_c for every constructor symbol c from Σ. If the type of c is $\tau_1 \times \ldots \times \tau_n \to \tau$, then δ_c must be a function from $(S_{\tau_1} \times \ldots \times S_{\tau_n})$ to S_τ.
- S_{acc}, the set of *accepting states*, is a subset of S_{τ_0}.

An automaton traverses a tree from the leaves towards the root and assigns a state to every node v in the following way:

- Let $s_1, \ldots, s_n$ be the states assigned to the n children of v and let c be the label of v. Then v is assigned the state $\delta_c(s_1, \ldots, s_n)$.

The automaton *accepts* the tree if the state assigned to the root node is in S_{acc}. Otherwise it *rejects* the tree. We say that an automaton *defines* the forest of those trees that it accepts.

From an algebraic point of view, the pair (S, δ) is an algebra for the signature Σ with finite carrier sets S_τ, and the automaton evaluates a ground term in this algebra.

2.3 Tree Grammars

A *simple tree grammar* for a signature Σ and a type name τ_0 from Σ is a tuple (N, P, Ax) where

- N is a family of disjoint finite sets N_τ of *nonterminal symbols* for every type name τ from Σ,
- P is a finite set of *productions* of the form $a \to c(a_1, \ldots, a_n)$, where $a, a_1, \ldots, a_n$ are nonterminals and c is a constructor symbol. If the type of c is $\tau_1 \times \ldots \times \tau_n \to \tau$, then $a, a_1, \ldots, a_n$ must be members of $N_\tau, N_{\tau_1}, \ldots, N_{\tau_n}$ respectively.
- Ax, the set of *axioms*, is a subset of N_{τ_0}.

An *extended tree grammar* for Σ and τ_0 is defined like a simple tree grammar, except that the right-hand sides of productions can be chosen more flexibly:

– A non-terminal from N_τ is a right-hand side of type τ.
– If c is a constructor symbol of type $\tau_1 \times \ldots \times \tau_n \to \tau$ and $b_1, \ldots, b_n$ are right-hand sides of types $\tau_1, \ldots, \tau_n$, respectively, then $c(b_1, \ldots, b_n)$ is a right-hand side of type τ.

Now if the left-hand side of a production is a nonterminal in N_τ, then the right-hand side must be of type τ. Notice that right-hand sides are essentially terms where the non-terminals play the role of variables. Except for the minor difference that we allow for a set of axioms rather than a single one, tree grammars are special context-free grammars. The nonterminals are the constructor symbols, the comma, and the parentheses.

A simple or extended tree grammar defines a language as usual, except that we may start from an arbitrary axiom to generate a member of the language. Obviously all the members of the language defined by a tree grammar for a type τ_0 are trees of this type τ_0.

Since (simple and extended) tree grammars are the only grammars used in this paper, we will simply speak of (simple and extended) grammars.

In the rest of the paper we will rely on the following results:

Proposition 1. *Let L be a forest. Then the following are equivalent:*

– *There is an automaton defining L.*
– *There is a simple grammar defining L.*
– *There is an extended grammar defining L.*

Furthermore there are algorithms that convert automata into equivalent simple and extended grammars and vice versa.

It is decidable whether the forest defined by a given automaton or grammar

– *contains a given tree,*
– *is empty,*
– *is finite,*
– *is a subset of the forest defined by another given automaton or grammar.*

Given two automata or grammars defining the forests L and L', resp., it is possible to construct an automaton and a grammar defining the forests $L \cup L'$, $L \cap L'$, and $L \setminus L'$.

Proofs of these results for the untyped case are similar to the proofs for the corresponding results for finite automata operating on strings and left-linear or right-linear grammars. They can, e.g., be found in [2].[1] The extension to the typed case is straight-forward.

A forest is *regular* if it is defined by some automaton, or, equivalently, by some simple/extended grammar.

We will need a grammar that generates every tree of some type τ_0. Such a grammar can obviously be constructed as follows:

[1] In fact, the results from [2] even hold for terms with variables. For our purposes it suffices to deal with ground terms.

- For every type name τ the set N_τ contains exactly one nonterminal, say a_τ.
- For every constructor $c : \tau_1 \times \ldots \times \tau_n \to \tau$ there is one production $a_\tau \to c(a_{\tau_1}, \ldots, a_{\tau_n})$
- The only axiom is a_{τ_0}.

We call this grammar the *complete grammar* for τ_0.

3 Algebraic Types and Relational Databases

For the rest of the paper we will assume that the schema of a database consists of a signature Σ and a type τ_0 and that the trees of type τ_0 are the legal definite database states.

This data model suffices to emulate relational databases. Relations are sets of tuples. The tuples in a relation have the same number of components and corresponding components in different tuples of a relation are of the same type.

Sets or multisets can be represented by list types if we ignore the order (and in the case of sets, also the multiplicity) of members of a list. Lists can be defined as algebraic datatypes, and the same holds for tuples.

Since we did not introduce parameterized types, we need different list and tuple types for different relation schemas:

- A type name τ is a *list type* if there are exactly two constructors for τ, a nullary one and a binary one, and the second argument type of the binary constructor is again τ. The first argument type of the binary constructor is called the *member type* of τ.
- A type name τ is a *tuple type* if there is exactly one constructor for τ. The argument types of this constructor are called the *component types* of τ.

Notice that the nullary and the binary constructors for list types play the role of `nil` and `cons` in Lisp. For the sake of readability we will use similar constructor names for list types in examples.

Now a relation schema corresponds to a list type whose member type is a tuple type with component types corresponding to the respective attribute domains. A schema for a relational database corresponds to a tuple type with component types corresponding to the relation schemas.

Relational algebra operations like projection, selection, and join are typically parameterized operations. For example a generic projection operation is parameterized by the list of attributes that should appear in the output relation and a selection is parameterized by the selection condition. In a typed environment, operations are also (implicitly) parameterized by the types of their input relations. Instead of such generic operations we will use a specific operation for every parameterization.

Example 2 (projection). Let the typenames `A`, `B`, `C`, `R1`, `T1`, `R2`, and `T2` be given together with the constructors `Cons1` : `T1` $\times$ `R1` $\to$ `R1`, `Nil1` : `R1`, `MakeT1` : `A` $\times$ `B` $\times$ `C` $\to$ `T1`, `Cons2` : `T2` $\times$ `R2` $\to$ `R2`, `Nil2` : `R2`, `MakeT2` : `A` $\times$ `C` $\to$ `T2`, and some constructors for `A`, `B`, and `C`.

Then we can project a relation of type R1 to a relation of type R2 by omitting the second attribute. This specific projection function can be defined as

$$\begin{array}{lll} f : \mathit{Ext}_{\mathtt{R1}} & \to & \mathit{Ext}_{\mathtt{R2}} \\ \quad \mathtt{Nil1} & \mapsto & \mathtt{Nil2} \\ \quad \mathtt{Cons1(MakeT1(}a\mathtt{,}b\mathtt{,}c\mathtt{),}r\mathtt{))} & \mapsto & \mathtt{Cons2(MakeT2(}a\mathtt{,}c\mathtt{),}f\mathtt{(}r\mathtt{))} \end{array}$$

Notice that such a function can be implemented in a functional programming language without any use of higher-order features. Also notice that it does not matter whether this language is strict or not.

Similar considerations hold for the other operations of relational algebra.

4 Indefinite Database States

According to the considerations in the introduction, an indefinite database state is (a representation of) a set of definite database states. In general there can be an infinite number of definite database states. Therefore there are more than countably many indefinite database states and we cannot represent all of them. In this paper we consider the class of those indefinite database states that are regular sets of definite database states, i.e., that can be described by finite automata or grammars.

In order to get an impression of the expressive power of regular forests, we will now look at some examples of indefinite database states that do or do not belong to this class.

4.1 Finite Number of Alternatives

Any indefinite database state representing a finite set of definite database states is regular. It can be described by an extended grammar with a single axiom a_0 and productions $a_0 \to t$ for every possible definite state t.

4.2 Wildcards

Let a term t be given in which every variable occurs only once. (Variables occurring only once will also be called *wildcards*.) Then we can define an indefinite database state representing all the instances of t.

Such an indefinite database state belongs to our class since it is defined by an extended grammar that can be constructed as follows: We start with the complete grammar for an arbitrary type name and add

- a new nonterminal a_0 to N_{τ_0}, which is also made the only axiom, and
- one production $a_0 \to t'$ where t' is t with the variables replaced by nonterminals a_τ with appropriate types τ.

When the given signature emulates a relational database schema, then there are two important special cases of this kind of indefiniteness:

– Wildcards appearing as components of tuples in relations emulate the usual "no-information" null values.
– Let t be a list representing some relation R and let t' be the same tree as t except that the nullary constructor that terminates the list t is replaced by a wildcard. Then the indefinite database state corresponding to t' represents all appropriately typed supersets of R.
This covers the frequent kind of incomplete knowledge in relational databases when just a subset of the real relation is known and it is not known how many additional tuples there should be in the relation, nor is anything known about the contents of these tuples.

4.3 Simple Constraints

A refinement of the approach described in Section 4.2 is the following: We may attach certain constraints to the wildcards; for every wildcard we may require that it may be instantiated only with a tree from some given regular forest.

Suppose that for every wildcard v we have an extended grammar Γ_v defining the trees that may be substituted for the wildcard. Then we can construct an extended grammar for the indefinite database state as follows:

– First, we rename the nonterminals in the grammars Γ_v in such a way that no two of these grammars have nonterminals in common.
– Then we ensure that each of these grammars has exactly one axiom by introducing a new nonterminal a_v as the only axiom and adding productions $a_v \rightarrow a$ for all old axioms a of the respective grammar.
– Finally we collect all the nonterminals and productions of these grammars in a single grammar and add a new nonterminal a_0 as the only axiom of this grammar as well as a production $a_0 \rightarrow t'$ where t' is the term t in which every wildcard v is replaced by the nonterminal a_v.

4.4 Relations with Lower and Upper Bounds

Assume that for some database relation R we know some finite subset R^- and some finite superset R^+. Since there is only a finite number of relations R for which $R^- \subseteq R \subseteq R^+$ holds, the indefinite database state represents only a finite set of definite database states, and from Section 4.1 we know that such finite sets are always regular.

However, it is possible to construct a grammar for the indefinite database state in a more elegant and less "brute-force" manner:

– Let τ be the type of the lists used to represent R and let `Cons` and `Nil` be the binary and the nullary constructors for τ.
– Let $t_1, \ldots, t_n$ be tuples corresponding to the members of $R^+ \setminus R^-$. Then we introduce nonterminals $a_1, \ldots, a_{n+1} \in N_\tau$ and productions $a_i \rightarrow a_{i+1}$ and $a_i \rightarrow \texttt{Cons}(t_i, a_{i+1})$ for $i = 1, \ldots, n$, and a production $a_{n+1} \rightarrow \texttt{Nil}$.

– Let t be a list corresponding to R^- and let t' be the same tree with the `Nil` terminating the list replaced by a_1. We also introduce a nonterminal $a_0 \in N_\tau$, which also becomes the only axiom, and a production $a_0 \rightarrow t'$.

When we generate a tree using this grammar, this tree will via t' include all the tuples from R^-, and for any tuple $t_i \in R^+ \setminus R^-$ we have the choice to include it or not by selecting the appropriate production for a_i.

4.5 Equality Constraints

In Section 4.2 we have seen that an indefinite database state that can be described by a term with wildcards is regular, and in Section 4.3 we have seen that it is possible to attach certain constraints to the wildcards without sacrificing regularity. Now we want to attach constraints that involve more than one wildcard.

A simple case is the following: All we know is that the database state is of the form `C(_,_)` and the same tree must occur at the two wildcard positions marked by "`_`". That is, we have attached an equality constraint to the two wildcard positions. The corresponding forest $\{\texttt{C(}t\texttt{,}t\texttt{)} \mid t \text{ is a tree of appropriate type}\}$ is finite and therefore regular according to Section 4.1 if the extent of the type τ for the two wildcard positions is finite.

If, however, the extent of τ is infinite, then the forest is not regular, as we can see with the following consideration: Assume there were a finite automaton defining the forest. This automaton assigns some state to any tree of type τ. Since the extent of τ is infinite and there are only finitely many states, there must be two different trees t and t' which are mapped to the same state. So the automaton also assigns the same state to the trees $\texttt{C(}t\texttt{,}t\texttt{)}$ and $\texttt{C(}t\texttt{,}t'\texttt{)}$. Therefore it cannot accept the former and reject the latter.

It is now obvious that even in cases where the extent of τ is finite, an automaton would need one state per tree of type τ in order to implement such equality constraints. This might not be feasible in many practical cases.

5 Operations on Indefinite Databases

A query to a definite database is a function that maps a definite database state to some answer. Let us assume that answers are trees again. This assumption is useful because it allows us to compose queries. An update operation for a definite database can also be seen as a function that maps an old database state to a new one. So queries and updates for definite databases are functions mapping trees to trees. Similarly queries and updates for indefinite databases can be seen as functions mapping forests to forests.

We distinguish two kinds of functions for indefinite databases:[2]

[2] Such a distinction is, e.g., also made in [5].

– We can *lift* a function f from definite database states to indefinite database states. The result of the lifted function $f\uparrow$ applied to an indefinite database state s is the set of results of f applied to the definite database states represented by s, i.e., $f\uparrow(s) := \{f(t) \mid t \in s\}$.
– There are also functions for indefinite databases that cannot be described as lifted versions of functions for definite databases. These functions typically involve some information about the indefiniteness itself. For example, such a function might return whether an indefinite database state represents a finite or an infinite number of definite states.

We say that a function mapping forests to forests is *regular* if it maps regular forests to regular forests. Regular functions are interesting because they can be implemented by functions mapping automata or grammars to automata or grammars. We also say that a function mapping trees to trees is *regular* if its lifted version is regular. In the rest of this section we will consider various kinds of functions and see whether they are regular.

We start with the simple function

$$\begin{aligned} f : \mathit{Ext}_\tau &\to \mathit{Ext}_{\tau'} \\ x &\mapsto \mathtt{C(}x\mathtt{,}x\mathtt{)} \end{aligned}$$

where $\mathtt{C}$ is a constructor with type $\tau \times \tau \to \tau'$. This function is not regular because $f\uparrow$ maps the regular forest $\{t \mid t \text{ is a tree of type } \tau\}$ to the non-regular forest $\{\mathtt{C(}t\mathtt{,}t\mathtt{)} \mid t \text{ is a tree of type } \tau\}$ (see Section 4.5).

5.1 Nonrecursive Linear Functions

Notice that the problem with the example above is that the argument x is used twice in the right-hand side of the definition of f. We will avoid the problem by using "linear" function definitions, i.e., function definitions that use every parameter at most once on the right hand side. For the sake of simplicity we will also not use recursion in this section.

A *nonrecursive linear function* is a function defined as

$$\begin{aligned} f : \mathit{Ext}_\tau &\to \mathit{Ext}_{\tau'} \\ p_1 &\mapsto t_1 \\ &\vdots \\ p_m &\mapsto t_m \end{aligned}$$

where the p_i and the t_i are terms and the following conditions hold:

– Every tree of type τ is matched by exactly one of the p_i.
– Every variable in a t_i occurs in the corresponding p_i.
– No variable occurs twice in a p_i or in a t_i.
– All the p_i are of type τ and all the t_i are of type τ'.

We will call the terms p_i the *patterns* in the function definition.

Every function defined in this way is regular. To see this, we show how a simple grammar Γ defining a set s of trees of type τ can be transformed into an extended grammar Γ' defining the forest $f \uparrow (s)$:

- We construct sets $Q, Q_1, \ldots, Q_m$ of terms containing nonterminals:
  1. Let Q be the set of axioms of Γ and let the Q_i be empty initially.
  2. We will now try to assign every member q of Q to some case of the function definition. If necessary we will "split" q:
     - If all the trees matching q match the same pattern p_i, then we remove q from Q and add it to Q_i.
     - If trees matching q match more than one pattern p_i, then we replace q in Q by several more specific terms which together match the same trees as q:
       Let p_i be one of the relevant patterns, i.e., one of the patterns that matches some trees that are also matched by q. There must be a nonterminal a in q at some position such that at the same position in p_i there is not a variable. (Otherwise p_i would match all the trees matched by q, and, according to the conditions above, no other $p_{i'}$ would match any of the trees matched by q.) We expand q at this position by a single derivation step with all the productions for a in Γ and replace q in Q by all the generated terms.
       We continue with step 2 for the new members of Q.

  Since the patterns p_i are of finite height, this process terminates. Then the set Q is empty. From the terms in Q_i one can derive exactly those trees that are generated by Γ and match the pattern p_i. Furthermore every term in Q_i has exactly the shape of p_i, except that the variables are replaced by nonterminals.
- The grammar Γ' now consists of the following:
  - All the nonterminals and productions of Γ.
  - A new nonterminal a' which is the only axiom of Γ'.
  - For every member q of some Q_i a production $a' \to u$ where u is generated from t_i by replacing every variable x with the nonterminal that appears in q at the position where x appears in p_i.

So the idea of this approach was to expand the axioms of Γ until for every expansion exactly one case of the function definition is relevant. Then we can apply f to these expansions as if the nonterminals were trees. This is essentially what enables us to apply the function to a possibly infinite number of trees in finite time.

5.2 Relational Algebra Operations

We will now see whether and under which conditions the relational algebra operations are regular. The following definitions will be used:

Let an indefinite relation be represented by a simple grammar for lists of type τ with the binary constructor `Cons` and the nullary constructor `Nil`. Then certain nonterminals and productions are used to generate the "backbone" of the list. We call them the *backbone nonterminals* and *backbone productions*, respectively, and extract them from the grammar as follows:

- All the axioms are backbone nonterminals.
- If the left-hand side of a production is a backbone nonterminal, then the production is a backbone production. A backbone production is called *terminating* if its right-hand side is `Nil`. Otherwise, i.e., if its right-hand side is of the form `Cons(`a`,`a'`)`, it is called *non-terminating*.
  In this latter case a' is also a backbone nonterminal and we call a a *member nonterminal*.

Union The union of two relations becomes the list concatenation in our representation. The concatenation function for lists of type τ with constructors `Cons` and `Nil` can be defined as follows:

$$\begin{aligned} conc : & \; Ext_\tau \times Ext_\tau && \to Ext_\tau \\ & (\mathtt{Cons}(x,y), z) && \mapsto \mathtt{Cons}(x, conc(y,z)) \\ & (\mathtt{Nil}, z) && \mapsto z \end{aligned}$$

This function terminates since the length of the first argument is finite.

Now let two simple grammars Γ_1 and Γ_2 be given. We construct an extended grammar Γ that generates exactly the concatenations of two lists generated by Γ_1 and Γ_2.

A straight-forward approach would be to follow the ideas of Section 5.1, i.e., to expand the axioms of Γ_1 as long as needed and to apply the cases of the function definition to the expansions. However, the recursion in such a procedure would not terminate in general since even though all the lists generated by Γ_1 are of finite length, there is not necessarily a finite upper bound for this length.

It is nevertheless possible to construct Γ as follows:

- If necessary, rename nonterminals in Γ_2 so that Γ_1 and Γ_2 have no nonterminals in common.
- Replace every terminating backbone production $a \to \mathtt{Nil}$ in Γ_1 by productions $a \to a'$ where a' ranges over all the axioms of Γ_2.
- Now Γ consists of all the nonterminals and productions of Γ_1 and Γ_2. Its axioms are the axioms of Γ_1.

Notice that this approach works only if the two indefinite lists are independent from each other, i.e., if the definite list for the second argument can be chosen independently from the choice of the first argument. A simple example where this independence is not given is the following:

Let f be the function that appends a list to itself. If we apply $f \uparrow$ to the forest

$$s := \{\mathtt{Cons}(t,\mathtt{Nil}) \mid t \text{ is an arbitrary tree of appropriate type}\},$$

where the type of t has an infinite extent, we get

$$\{\texttt{Cons(}t\texttt{,Cons(}t\texttt{,Nil))} \mid t \text{ is an arbitrary tree of appropriate type}\}.$$

This forest is not regular, even though s is. So f is not regular. The problem is again that the definition of f is not linear since it uses its argument twice.

From the relational algebra point of view this example is not too convincing: The operation constructing the union of a set with itself is the identity. It can be emulated by the identity operation on lists, which is obviously regular. But it is easy to construct slightly more complex examples where such a solution does not work.

Projection With a very restrictive definition of the word, a projection just removes some attributes from a relation. With a more general definition, a projection applies some function f to every tuple of a relation and collects the resulting tuples in an output relation. We will see that a projection in this more general sense is regular if the function f is.

Let Γ be a simple grammar describing the indefinite input relation. Then a grammar Γ' for the output relation is obtained from Γ as follows:

- For every member nonterminal b of Γ let Γ_b be the same grammar as Γ except that b is the only axiom.
  We apply$f \uparrow$ to the forest defined by Γ_b and, since f is regular, we get a regular forest again. Let $f(\Gamma_b)$ denote a grammar for this forest. We will assume that the nonterminals are renamed in such a way that any two grammars $f(\Gamma_b)$ and $f(\Gamma_{b'})$ with $b \neq b'$ do not have any nonterminal in common and that no grammar $f(\Gamma_b)$ has any nonterminal in common with Γ.
- Now Γ' consists of the following:
  - All the nonterminals and productions from all the grammars $f(\Gamma_b)$.
  - All the backbone and member nonterminals and all the backbone productions of Γ.
  - For every member nonterminal b of Γ and every axiom a of $f(\Gamma_b)$ a production $b \to a$.
  - The axioms of Γ' are the axioms of Γ.

The underlying idea for this construction is similar to the one used for the concatenation: Do not use the recursive definition of a projection function (as, e.g., given in example 2), but handle the relevant parts of the grammar (the member nonterminals in this case, the terminating productions of Γ_1 in the case of the concatenation) directly.

Notice that a projection that duplicates an attribute is not regular unless the type of this attribute has a finite extent. This is again due to the lack of linearity.

Selection A selection applies some predicate, the *selection condition*, to all the tuples of a relation and collects those tuples for which the selection condition holds. Typical simple selection conditions are that an attribute must be equal to some given value or that two attributes must be equal. But already the latter kind of condition may lead to a non-regular selection condition: Consider the regular forest

$$s := \{\texttt{Cons(Tup(}t\texttt{,}t'\texttt{),Nil)} \mid t \text{ and } t' \text{ are trees of type } \tau\}$$

for some appropriate type τ with infinite extent and the selection function f with the condition that the two components of a tuple must be equal. Then

$$f \uparrow (s) = \{\texttt{Nil}\} \cup \{\texttt{Cons(Tup(}t\texttt{,}t\texttt{),Nil)} \mid t \text{ is a tree of type } \tau\}$$

(The first and the second argument of the union stand for the cases $t \neq t'$ and $t = t'$, respecively.) is not regular and therefore f is not regular.

Nevertheless an equality test between two attributes is possible in a selection condition if the selection function also immediately removes one of the two involved attributes. In the above example we get the regular forest

$$\{\texttt{Nil}\} \cup \{\texttt{Cons'(Tup'(}t\texttt{),Nil')} \mid t \text{ is a tree of type } \tau\}$$

with appropriate constructors `Cons'`, `Nil'` and `Tup'`.

We will now consider the following class of selection functions, which covers the aforementioned conditions: Let some term p containing variables $v_1, \ldots, v_n$ be given. (Every variable may have several occurrences.) Then a list member is selected if it is an instance of p. Let another term t be given which contains each of the variables v_i at most once and no other variables. Then the element inserted into the result of the selection is the instance of t where the variables v_i are instantiated in the same way as in p.

Any such selection function is regular. Given a simple grammar Γ for an indefinite input relation and the terms p and t with variables $v_1, \ldots, v_n$, we construct an extended grammar Γ' for the indefinite output relation as follows:

- For every member nonterminal b of Γ let Γ_b be the same grammar as Γ except that b is the only axiom.
- For every member nonterminal b of Γ we construct sets Q_b and S_b and a boolean value F_b as follows: (Q_b contains terms derivable from b that still need to be handled; S_b contains grammars needed for the case that the selection condition succeeds, and F_b will finally say whether the selection condition might fail for some tree derivable from b.)
  1. Initially Q_b is $\{b\}$, S_b is empty and F_b is false.
  2. We will now try to decide for every member q of Q_b whether it satisfies the selection condition or not. If necessary we will "split" q:
     - If there is a position in p with a constructor symbol, the same position exists in q and there is a different constructor symbol, then we remove q from Q_b. If in addition there is at least one tree that can be derived from q with productions from Γ, then we set F_b to true.

- If for every position in p with a constructor symbol the same position exists in q and there is the same constructor symbol, then we remove q from Q_b. Furthermore we add to S_b a grammar Γ_q which is constructed as follows:
  - For every variable v_i $(i = 1, \dots, n)$ we find its occurrences in p. Let A_{v_i} be the set of nonterminals at the corresponding positions in q. (All the subterms at these position are in fact nonterminals!) For every $a \in A_{v_i}$ let Γ_a be the same grammar as Γ except that a is the only axiom.

    If there are different nonterminals a_1 and a_2 in A and different trees $t_1 \in \Gamma_{a_1}$ and $t_1 \in \Gamma_{a_2}$ and if none of the Grammars Γ_a $(a \in A)$ defines the empty forest, then we also set F_b to true as a side effect.

    Let Γ_{v_i} be a grammar that defines the intersection of the forests defined by the grammars Γ_a.
  - Rename the nonterminals in these grammars Γ_{v_i} in such a way that all these grammars have disjoint sets of nonterminals.
  - Now Γ_q contains
    - all the nonterminals and productions from the Γ_{v_i},
    - a new nonterminal a_t, which becomes the only axiom of Γ_q,
    - a new nonterminal a_{v_i} for $i = 1, \dots, n$, and
    - a production $a_t \to t'$ where t' is the term t with all variables v_i replaced by the corresponding nonterminal a_{v_i}.
- If there is a nonterminal a at some position in q such that the same position exists in p and there is a constructor symbol at this position, then we expand q at this position by a single derivation step with all the productions for a in Γ and replace q in Q by all the generated terms.

  We continue with step 2 for the new members of Q.

This procedure terminates since p is finite. In the end Q_b is empty.
Let Γ'_b be a grammar defining the union of the forests defined by the grammars in S_b. (This grammar generates all the instances of t that correspond to instances of p that are generated by Γ_b.) Rename the nonterminals in the grammars Γ'_b in such a way that all these grammars and also Γ have disjoint sets of nonterminals.

– Now Γ' contains
  - all the nonterminals and productions from the grammars Γ'_b where b is a member nonterminal of Γ,
  - all the backbone nonterminals of Γ,
  - all the terminating backbone productions of Γ,
  - all the non-terminating backbone productions of Γ with the member nonterminal b replaced by any axiom of Γ'_b.
  - for every non-terminating production $a \to \texttt{Cons}(b, a')$ of Γ (where `Cons` is the binary list constructor for the backbone type): if F_b is true (i.e., the selection might fail on a tree generated from b) then the production $a \to a'$.

Cartesian Product Consider the list

$$l := \texttt{Cons1(Tup1(A),Cons1(Tup1(B),Nil1))}$$

and the regular forest

$$s := \{\texttt{Cons2(Tup2(}t\texttt{),Nil2)} \mid t \text{ is a tree of appropriate type}\}.$$

The forest of cartesian products of l and a member of s is

$$\{\texttt{Cons3(Tup3(A,}t\texttt{),Cons3(Tup3(B,}t\texttt{),Nil3))} \mid t \text{ is a tree of appropriate type}\}$$

with appropriate constructors `Tup3`, `Cons3`, and `Nil3`. This forest is not regular. So we see that the cartesian product is not necessarily regular in one argument even if the other argument is definite.

5.3 Non-Lifted Operations

An important non-lifted operation is the merging of two sources of information. Let s and s' be the forests represented by two regular indefinite database states. So we know that the one definite database state that represents the real world is a member of both s and s'. So a merged indefinite database state must represent $s \cap s'$, which is also regular according to Proposition 1.

Other regular non-lifted operations are

- the set difference between forests, which can be used to exclude possibilities in an indefinite database state when they are no more possible.
- the union of forests, which can be used if it is known that one of several indefinite databases reflects the real world.

6 Conclusion

This paper has introduced a way to specify indefinite information in databases with a data model based on algebraic datatypes. We have seen how this also allows to emulate indefinite information in relational databases.

The introduced class of indefinite values, the regular forests, has been investigated for whether it is closed with respect to certain operations, especially the operations of relational algebra. We have seen that it is closed for unions, projections, and selections under certain circumstances. These are also roughly the operations that are possible with simple "no-information" null values (also called "Codd null values"; [4]). The reason for the restriction to these operations is in both cases that it cannot be expressed that two values are unknown but equal to each other.

The given algorithms for the operations are certainly not optimal and are not intended for direct implementation. They just served as evidence that the respective operations are computable in principle.

The following two approaches could be followed in order to allow more (relational algebra) operations on indefinite databases:

– We might extend the class of regular forests to a larger class of forests which is closed w.r.t. all the interesting operations. It is, however, not clear whether all the nice decidability results that exist for regular forests also hold for the extended class.
– Another approach would be to give up the exact computation of forests for the sake of efficiency. We could rather approximate forests by regular upper bounds. So for example, the non-regular forest

$$\{\texttt{C(}t\texttt{,}t\texttt{)} \mid t \text{ is a tree of appropriate type}\}$$

could be approximated by the regular superset

$$\{\texttt{C(}t\texttt{,}t'\texttt{)} \mid t \text{ and } t' \text{ are trees of appropriate type}\}.$$

A similar idea has been followed by Reiter [6], which makes his algorithm only "sometimes complete".

Acknowledgement

The support for the author by Bayerischer Habilitations-Förderpreis is appreciated.

References

1. E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.
2. F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.
3. G. Grahne. *The Problem of Incomplete Information in Relational Databases*. Number 554 in LNCS. Springer Verlag, 1991.
4. T. Imieliński and W. Lipski. Incomplete information in relational databases. *Journal of the Association for Computing Machinery*, 31(4):761–791, October 1984.
5. L. Libkin. *Aspects of Partial Information in Databases*. PhD thesis, University of Pennsylvania, 1994.
6. R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the Association for Computing Machinery*, 33(2):349–370, April 1986.