Available online at www.sciencedirect.com

ScienceDirect

Information Sciences 178 (2008) 2507–2533

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

# Safety, domain independence and translation of complex value database queries

Hong-Cheu Liu [a,*,1], Jeffrey Xu Yu [b], Weifa Liang [c]

[a] *Department of Computer Science and Information Engineering, Diwan University, Tainan County 72153, Taiwan*
[b] *Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong, China*
[c] *Department of Computer Science, The Australian National University, Canberra ACT 0200, Australia*

Received 1 February 2007; received in revised form 17 December 2007; accepted 12 February 2008

**Abstract**

This paper considers the theory of database queries on the complex value data model with external functions. Motivated by concerns regarding query evaluation, we first identify recursive sets of formulas, called *embedded allowed*, which is a class with desirable properties of "reasonable" queries.

We then show that all embedded allowed calculus (or fix-point) queries are domain independent and continuous. An algorithm for translating embedded allowed queries into equivalent algebraic expressions as a basis for evaluating safe queries in all calculus-based query classes has been developed.

Finally we discuss the topic of "domain independent query programs", compare the expressive power of the various complex value query languages and their embedded allowed versions, and discuss the relationship between safety, embedded allowed, and domain independence in the various calculus-based queries.
© 2008 Elsevier Inc. All rights reserved.

*Keywords:* Complex value; Safe query; Domain independence; Query translation; Expressive power

## 1. Introduction

One major issue in the study of database query languages is their expressiveness and complexity. A precise language-independent characterization of "reasonable" queries will help us to understand what a query language may express. A database is just a finite set of finite relations which are defined on some domain. A query language for such a database is a first-order logic formula over the signature. The signature includes predicate symbols for these relations. In practice, it may also include other predicate symbols and function symbols (like "$\leqslant$", "$+$", in case the intended domain is a subset of real numbers). The signature might be in the form: $\Omega = \{R_1, \ldots, R_n, f_1, \ldots, f_k, p_1, \ldots, p_l\}$, where $R_i$, $1 \leqslant i \leqslant n$, are database relations, $f_i$, $1 \leqslant i \leqslant k$, are scalar

[*] Corresponding author.
*E-mail addresses:* hcliu@dwu.edu.tw (H.-C. Liu), yu@se.cuhk.edu.hk (J.X. Yu), wliang@cs.anu.edu.au (W. Liang).
[1] Most of this research was carried out while the author was at The Australian National University and also visiting The Chinese University of Hong Kong.

0020-0255/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2008.02.005

external functions and $p_i$, $1 \leqslant i \leqslant l$, are predicates. The answer to the query $\phi(x_1, \ldots, x_k)$ with $k$ free variables is a relation with arity $k$ containing a finite set of $k$-tuples which satisfy the formula for the underlying structure $\mathscr{M} = (D, \Omega)$ with domain $D$ and a fixed signature $\Omega$.

Ideally, every formula of a query language can serve as a query. However, not every formula can be accepted as a query as far as the query output is concerned. Given a query we normally expect that values of any correct answer lie within the active domain of the query or the input [26]. There are only certain calculus queries (or formulae) which can be regarded as reasonable in this sense. Such queries are called *domain independent* queries as they yield the same answer no matter what the underlying domain of interpretation. The following are examples of unreasonable query phenomenon adapted from [3,29].

(1) Given a relation schema $Movies(Title, Director, Actor)$,$q_1 \equiv \{x \mid \neg Movies(CriesandWhispers, Bergman, x)\}$
(2) Given two relations $R(w, y)$ ($w$ requires $y$) and $S(x, y)$ ($x$ supplies $y$). The question, "*Which* suppliers supply all parts required by project ANU2000?" is expressed by $q_2 \equiv \{x \mid \forall y[\neg R(ANU2000, y) \vee S(x, y)]\}$.
(3) $q_3 \equiv \{x, y \mid S(x, f(y))\}$.

For query $q_1$, different answers are obtained if the type *Actor* includes all and only the current actors or includes all current and potential actors. If the usual semantics of predicate calculus are adapted directly to this context, i.e., the underlying domain is an infinite set **dom**, then the query $q_1$ produces all tuples $\langle a \rangle$ where $a \in$ **dom** and $< CriesandWhispers, Bergman, a >$ is not in the input. This answer would be an infinite set. The problem of query $q_2$ is that if $R(ANU2000, y)$ is empty, then $q_2$ is true for all values of $x$, which may not be the intended answer(s). For query $q_3$, infinitely many values for $y$ might map to a member of the second coordinate of $S$, in which case the output of $q_3$ would be infinite. The set of correct answers for each of the above queries depends on the domains of the variables.

Complex object models are being widely used to support new applications such as engineering design, multimedia data management, spatial information systems, Web databases and data mining applications. Complex values (nested relations, complex objects) are increasingly parts of these advanced database systems. Intuitively, complex values are relations in which the entries may be themselves tuples or relations.

In the complex value databases (e.g., object-relational databases), a construct is provided whereby the user can create a named user-defined type (UDT) with its own behavioral specification by specifying methods in addition to the attributes. In general, a UDT type can have a number of user-defined functions associated with it. Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended persistent stored modules language of SQL. External functions are written in a host language, with only their signature (interface) appearing in the UDT definition.

Support for both complex values and user-defined functions is important in a database management system. In response to these requirements, SQL3 generalizes the relational model into an object model offering abstract data types and therefore allows users to define data types which suit their applications.

The desirable properties of *reasonable* queries are that they should be domain independent, computable and representable in the same format as the input (i.e., the answer should be a finite relation). The query processor should be able to distinguish reasonable queries which satisfy the above criteria from unreasonable ones. However, it is undecidable whether a given calculus query is domain independent. Much research on database query languages has been devoted to the determination of *safe* formulas which gives guarantees of finite output. Therefore all commercial database systems provide users to express queries only formulas from some decidable, syntactically defined subclass of the finite queries.

Previous work has been concerned with identifying safe class of first-order formulas which are domain independent queries. Some attempts to extend the definition of domain independence for the complex value data model incorporated with external functions have already been made [1,24]. Abiteboul and Beeri [1] and Escobar-Molano et al. [13] proposed two related definitions for reasonable queries with functions. They tried to identify queries expressed in certain logic-based query languages with external functions as reasonable. However, their definitions fail for languages with some forms of iterations, like *fix-points* or *while*, etc. Suciu [24] proposed a notion of domain independence (called *external-function domain independence*) for queries with external functions and considered only algebraic query languages. None of these attempts, however, address the implementation issues of calculus queries with complex values within fix-points and deductive paradigms.

The goals of this paper is to identify recursive sets of complex value formulas which define domain independent queries and investigate the implementation issue of query translation of complex value calculus formulas.

The main contributions of the paper are the following.

- We explore the issue of the semantics of complex value calculus queries in the presence of functions. We first investigate the notion of domain independence of complex value calculus queries and focus on safe queries. We adopt finiteness dependency approach which is different from [1] to define the notion of syntactic criteria for queries which guarantee domain independence, namely, *embedded allowed*.
- A non-trivial procedure for translating embedded allowed formulas into equivalent algebra queries is presented in Section 5. The work presented in this section can be viewed primarily as an extension and synthesis of work in [13,29].
- Based on this translation, we show that all embedded allowed calculus queries are domain independent. We also consider fix-point queries and show that all embedded allowed fix-point queries in complex value calculus are domain independent and continuous.
- We investigate the issue of domain independent query programs and show that every query expressed in embedded allowed stratified Datalog satisfying some constraints is embedded domain independence.
- Finally we compare the expressive power of the various complex value query languages and their embedded allowed versions and discuss the relationship between embedded allowed formulas, domain independence and finiteness in the various calculus-based query languages.

The benefit of these results is that a query processor can determine whether input queries are reasonable by using these notions and then can transform embedded allowed formulas into equivalent complex value algebra expressions. In modern intelligent database systems, we expect that query systems are able to handle a wider range of calculus formulas correctly and efficiently. Accordingly, they will require more general query translators and optimizers. This paper undertakes a more comprehensive study of query translation in the complex value model.

The paper is organized as follows. After briefly discussing previous related work in the next section, we briefly review some basic concepts and definitions in Section 3. In Section 4, we identify recursive sets of complex value formulas which define domain independent queries. In Section 5, we describe a procedure to translate any embedded allowed formula into an equivalent algebraic expression. The relationship between embedded allowed formulas and domain independence is presented in Section 6. In Section 7, we investigate the important problem of finding syntactic restrictions on the database query programs (Datalog$^{cv}$) that ensure domain independence. In Section 8, we present a brief discussion concerning the relationship between safe-range and domain independence. Finally, we provide some conclusions and open problems in Section 9.

An extended abstract of this paper was presented at 1999 International Database Engineering and Applications Symposium [19].

## 2. Related work

There have been several attempts to define a decidable syntactically restricted subclass of the domain independent formulas in the context of relational databases. The most important proposals are: *range separable formulas* [11], *range restricted formulas* [21], *evaluable formulas* [12], *allowed formulas* [26], and *safe range queries* [3]. A rather simple and syntactic definition of *safe* appears in Ullman's revised edition [28] which has actually been implemented. Demolombe [12] proved that evaluable formulas are domain independent and showed that evaluable formulas have the same expressive power as domain independent formulas. Van Gelder and Topor [29] investigated the properties of evaluable and allowed classes and developed algorithms to transform an evaluable formula into an equivalent allowed formula and from there into relational algebra. The basic idea of their proposal is to define *generated* and *constrained* relations between variables and formulas in order to ensure that the scope of every quantified variable in a formula is sufficiently restricted for the entire formula to be domain independent.

The notion of safe range queries developed in [1] is based on "range-restriction" of variables occurring in calculus formulas. Each variable is attached to a range formula. The positive literal $R(x)$, $x \in t$, $x = t$, $x \subseteq t$ and

$\forall y(y \in x \rightarrow \varphi(y))$, and formulas obtained from them by using $\wedge$, $\vee$ are called range formulas. For example, $R(z) \wedge y \in z$ restricts $z$ and $y$; if $\varphi_1(x_1), \ldots, \varphi_n(x_n)$ restrict $x_1, \ldots, x_n$ respectively, then $\varphi_1(x_1) \wedge \cdots \wedge \varphi_n(x_n) \wedge f(x_1, \ldots, x_n) = y$ restricts $y$.

Escobar-Molano et al. [13] introduced the notion of *embedded allowed*, which generalized the "allowed" criteria to incorporate scalar functions and developed an algorithm for translating these embedded allowed queries into the relational algebra. The notion of range-restriction is weaker than the notion of embedded allowed. The notion of embedded domain independence developed in [13] is restriction to the flat relational case of the notion of bounded depth domain independence in [1]. Both notions are used only in conjunction with query languages without recursive queries (or any other kind of iterations). This paper generalizes the notion of embedded allowed for (recursive) queries for the complex value model.

The notion of external-function domain independence, proposed by [23], generalizes those of generic and domain independent queries on databases without external functions and can also be applied to query languages with fix-points or other kinds of iterations. Suciu [24] showed that all queries in a nested relational algebra language over a set of external functions, $\mathcal{NRA}(\Omega) + fix$, are external-function domain independent and continuous, while this paper shows that all "embedded allowed formulas" are external-function domain independent.

Beeri and Milo [8] studied the issue of safe calculus and their translation to an algebra based on the perspective of algebraic specifications. The notion of "strict DB-domain independent" defined in [8] is similar to the notion of embedded domain independent in [13]. Their paper also provided a definition of "safe" calculus queries, showed that they are strict DB-domain independent and indicated how to translate these into algebra queries. However, the notion of safe used in [8] is strictly weaker than "embedded allowed" developed in [13].

Badia [7] have defined a family of query languages with generalized quantifiers, *QLGQ*, and argued that the languages in this family allow users to express complex queries in an easy and intuitive manner. A safe and domain independent version was constructed. Almendros-Jimenez and Becerra-Teron [4] have presented an extended relational calculus for expressing queries in function-logic deductive databases. They have also studied syntactic conditions for the calculus formulas in order to ensure the domain independence property.

Avron [6,5] has developed a unified framework for dealing with constructibility and absoluteness in set theory, decidability of relations in effective structures and domain independence of queries in database theory.

## 3. Basic concepts

This section presents terminology for those previously studied concepts including database queries, complex value databases and query languages, and briefly reviews the notion of domain independence.

### 3.1. Complex value databases

The complex value data model is formed using the tuple and set constructors recursively. A fundamental characteristic of such complex values is that, in them, sets may contain members with arbitrarily deep nesting of tuple and/or set constructors.

We fix a countably infinite set **dom**, called the underlying domain. The elements of this domain are called atomic values. We associate a *sort* (i.e., type) with each relation name and each (complex) value. The following two definitions are adopted from [3].

**Definition 1.** The abstract syntax of sorts is given by

$$\tau = \mathbf{dom} | < B_1 : \tau, \ldots, B_k : \tau > | \{\tau\},$$

where $k \geqslant 0$ and $B_1, \ldots, B_k$ are distinct attributes.

The *sort* of a relation $R$ is simply denoted by $sort(R)$.

**Definition 2.** The interpretation of a sort $\tau$ (i.e., the set of values of $\tau$), denoted as $[\![\tau]\!]$, is defined recursively as follows:

(1) $[\![\mathbf{dom}]\!]=\mathbf{dom}$,
(2) $[\![\{\tau\}]\!] = \mathscr{P}^{fin}([\![\tau]\!]) = \{X | X \subseteq [\![\tau]\!]$ and $X$ finite$\}$, and
(3) $[\![\langle B_1 : \tau_1, \ldots, B_k : \tau_k\rangle]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_k]\!] = \{\langle B_1 : v_1, \ldots, B_k : v_k\rangle | v_j \in [\![\tau_j]\!], j \in [1, \ldots, k]\}$.

**Example 1.** Consider the sort of a complex value relation $R$: $\{\langle A : \mathbf{dom}, B : \mathbf{dom}, C : \{\langle A : \mathbf{dom}, E : \{\mathbf{dom}\}\rangle\}\rangle\}$. A value of this sort is $\{\langle A : a, B : b, C : \{\langle A : c, E : \{e\}\rangle, \langle A : d, E : \{\}\rangle\}\rangle, \langle A : e, B : f, C : \{\}\rangle\}$.

A (complex value) relation of sort $\tau$ is a finite set of values of sort $\tau$ – that is, a finite subset of $[\![\tau]\!]$.

**Definition 3.** A (complex value) *relation schema* is a relation name with an associated sort. A (complex value) *database schema* is a finite set of relation schemes, $\mathbf{R} = \langle R_1 : \tau_1, \ldots, R_n : \tau_n\rangle$, where $\tau_i$ is a sort for $i \in [1 \ldots n]$; and $R_i \neq R_j$ if $i \neq j$.

An *instance* $r$ of a relation schema $R$ is a finite subset of $[\![sort(R)]\!]$.

## 3.2. Query languages

Theoretical research on data models and languages for manipulating complex value data has grown out of three different but equivalent perspectives, namely, algebraic, calculus-based and logic programming oriented paradigms. This subsection briefly reviews notation that encompasses different formulations reflecting each of them.

### 3.2.1. A complex value algebra

We now introduce a many-sorted algebra for complex value databases [1]. Let $\Omega = \{R_1, \ldots, R_n, f_1, \ldots, f_l\}$ be a signature, where $R_i$, $1 \leqslant i \leqslant n$, are database relations and $f_i$, $1 \leqslant i \leqslant k$, are external functions. The complex value algebra over $\Omega$ is denoted as $\mathrm{ALG}^{cv}(\Omega)$. A family of core operators of the algebra is presented as follows. The detailed description of this algebra can be found in [3].

*The core of ALG$^{cv}$*

Let $r$, $r_1$, $r_2, \ldots$ be relations of sort $\tau$, $\tau_1$, $\tau_2, \ldots$, respectively.

*Set operations*: Union ($\cup$), intersection ($\cap$), and difference ($-$) are binary set operations.
*Tuple operations*: Selection ($\sigma$) and projection ($\pi$) are defined in the natural manner.
*Powerset*: $powerset(r)$ is a relation of sort $\{\tau\}$ where $powerset(r) = \{v \mid v \subseteq r\}$.
*Tuple creation*: If $A_1, \ldots, A_n$ are distinct attributes, $tup\_create_{A_1,\ldots,A_n}(r_1, \ldots, r_n)$ is of sort $\langle A_1 : \tau_1, \ldots, A_n : \tau_n\rangle$, and $tup\_create_{A_1,\ldots,A_n}(r_1, \ldots, r_n) = \{\langle A_1 : v_1, \ldots, A_n : v_n\rangle \mid \forall i(v_i \in r_i)\}$.
*Set creation*: $set\_create(r)$ is of sort $\{\tau\}$, and $set\_create(r) = \{r\}$.
*Tuple destroy*: If $r$ is of sort $\langle A : \tau'\rangle$, $tup\_destroy(r)$ is a relation of sort $\tau'$ and $tup\_destroy(r) = \{v \mid \langle A : v\rangle \in r\}$.
*Set destroy*: If $\tau = \{\tau'\}$, then $set\_destroy(r)$ is a relation of sort $\tau'$ and $set\_destroy(r) = \cup r = \{w \mid \exists v \in r, w \in v\}$.

An important subset of ALG$^{cv}$, called *nested relation algebra*, is formed from the core operations of ALG$^{cv}$ by removing the *powerset* operator and adding the *nest* operator. The nested relational algebra $\mathscr{NRA}$ [17] and monad algebra [10] is expressively equivalent to ALG$^{cv}$ without powerset.

**Example 2.** Flatting a complex value relation $S$ with $sort(S) = \langle B : \mathbf{dom}, B' : \{\mathbf{dom}\}\rangle$ means producing a set of flat tuples, each of which contains the first component of a tuple of $S$ and one of the elements of the second component. This is the unnest operation in the extended algebra. In the core algebra, we first obtain the set of values occurring in the $B'$ sets using

$$E = tuple\_create_C(set\_destroy(tup\_destroy(\pi_{B'}(S)))).$$

We can next compute $(E \times S)$. Then the desired query is given by

$$\pi_{BC}(\sigma_{C \in B'}(E \times S)).$$

### 3.2.2. A complex value calculus

In the complex value data model, the calculus is a many-sorted calculus. Calculus variables may denote sets so the calculus will permit quantification over sets. The calculus, denoted CALC$^{cv}$, is a strongly sorted extension of first order logic [1]. The vocabulary of the calculus language is defined as follows.

(1) parentheses (,);
(2) logical connectors $\wedge$, $\vee$, $\neg$, $\rightarrow$;
(3) quantifiers $\exists$, $\forall$;
(4) equality=, membership $\in$, and containment $\subseteq$ symbols;
(5) sorted predicate symbols;
(6) sorted tuple functions $\langle\rangle_{\tau_1,\ldots,\tau_n}$, and sorted set functions $\{\}_{\tau_1,\ldots,\tau_n}$.

**Definition 4.** *Terms* of the complex value calculus language are defined as follows:

(1) complex value constants of some sort $\tau$,
(2) variables whose sorts can be inferred from the context, and
(3) if $x$ is a tuple variable and $C$ is an attribute of $x$, then $x.C$ is a term.

**Definition 5.** *Atomic formulas* (positive literals) are sorted expressions of the form

$$R(t_1,\ldots,t_n),\quad t=t',\quad t\in t'\quad \text{or}\quad t\subseteq t',$$

where $R\in\mathbf{R}$, $\mathbf{R}$ is a database schema; and $t_i$, $t$, and $t'$ are terms or function symbols with the obvious sort compatibility restrictions.

*Formulas* are defined from atomic formulas using standard connectives and quantifiers.

**Example 3.** The collection of subsets of the second component of tuples of relation $R:\{\langle A,B\rangle\}$, which do not contain the values 5 or 10, is represented by the formula: $\{y\mid \exists u(R(u)\wedge y\subseteq u.B\wedge 5\notin y\wedge 10\notin y)\}$

**Example 4.** The replacement of the second component of tuples of relation $R:\{\langle A,B:\{\}\rangle\}$ by its cardinality is represented by the formula: $\{x\mid \exists y(R(y)\wedge x\cdot A=y\cdot A\wedge x\cdot B=count(y\cdot B))\}$ where count is a function which computes cardinality of a set.

### 3.2.3. Rule-based languages for complex values

Query languages based on the deduction paradigm are extensions of Datalog to incorporate complex values. These languages are based on the calculus and do not increase the expressive power of ALG$^{cv}$ or CALC$^{cv}$. However, certain queries can be expressed in this deduction paradigm more efficiently and with lower complexity than they can be by using the powerset operator in the ALG$^{cv}$ [14,15]. A major difference between the various proposals of logic programming with a set construct lies in their approach to nesting: grouping in LDL [9], data functions in COL [2], and a form of universal quantification in [18]. The key features of languages LDL and COL are reviewed in Appendix A.

The Datalog language is typed. A literal is an atomic formula or a negated atomic formula.

**Definition 6.** A *database clause* (rule) is an expression of the form

$$p(x_1,\ldots,x_i)\leftarrow L_1,\ldots,L_n,$$

where the head $p$ is a derived predicate, and each $L_i$ of the body is a literal. A program $\mathscr{P}$ is a finite set of rules.

We distinguish between the *intentional* predicates and functions, which appear in heads of rules, and the *extensional* ones, which appear only in bodies.

**Definition 7.** A *database* is a finite set of database clauses. A *query* is a formula of the form $\leftarrow W$, where $W$ is a calculus formula (i.e., $W\in$ CALC$^{cv}$) and any free variables in $W$ are assumed to be universally quantified at the front of the query.

**Definition 8.** Let $\mathscr{P}$ be a database program, $Q$ a query $\leftarrow W$. An *answer* to $\mathscr{P} \cup \{\leftarrow W\}$ is a ground substitution $\theta$ such that $\forall(W\theta)$ is a logical consequence of $\mathscr{P}$.

**Definition 9.** Let $\mathscr{P}$ be a database program, $W$ a formula, and $S$ an interpretation. Then ans($\mathscr{P}$, $W, S$) is the set of all answers to $\mathscr{P} \cup \{\leftarrow W\}$ that are ground substitutions for all free variables in $W$.

**Example 5.** The following is a Datalog$^{cv}$ program:

$p(\{a\}) \leftarrow$

$r(x) \leftarrow q(x) \wedge p(z) \wedge x \notin z.$

## 3.3. Database queries

To define the semantics of complex value calculus queries, it is convenient to introduce some notation. Given a database instance **I**, its active domain *adom*(**I**) is the set of all constants occurring in **I**. The set of constants occurring in a query $q$ is denoted as *adom*($q$). We use *adom*($q$, **I**) as an abbreviation for *adom*($q$) $\cup$ *adom*(**I**). We define $dom(\tau, \mathbf{d})$, for some sort $\tau$ and set **d** to be

(1) $dom(\mathbf{dom}, \mathbf{d}) \overset{\text{def}}{=} \mathbf{d}$,
(2) $dom(\{\tau'\}, \mathbf{d}) \overset{\text{def}}{=} \mathscr{P}^{fin}(dom(\tau', \mathbf{d}))$, and
(3) $dom(< B_1 : \tau_1, \ldots, B_k : \tau_k >, \mathbf{d}) \overset{\text{def}}{=} dom(\tau_1, \mathbf{d}) \times \cdots \times dom(\tau_k, \mathbf{d})$.

We write $\varphi(x_1, \ldots, x_n)$ to indicate that $x_1, \ldots, x_n$ is a listing of the variables occurring free in $\varphi$. Let $\mathbf{R} = (R_1, \ldots, R_k)$ be a database schema and $\mathbf{I} = (r_1, \ldots, r_k)$ be a database instance of **R** over domain **d**. We focus on a fixed finite set $\mathscr{F}$ of external functions which are associated with signatures. An interpretation is $(\mathbf{d}, \mathbf{F}, \mathbf{I})$, where $\mathbf{d} \subseteq \mathbf{dom}$, $\mathbf{F} \subset \mathscr{F}$, $\mathbf{F} = (f_1, \ldots, f_l)$, $f_i : dom(\tau_i, \mathbf{d}) \rightarrow dom(\tau_i', \mathbf{d})$ are external functions. A database *DB* is given by interpretation of each relation $r_i$ as a finite relation over **d** and augmented with a number of external functions $f_1, \ldots, f_l$.

Unless otherwise specified, we shall assume throughout the paper that the database signatures include external functions which are computable and partial. We allow external functions with arbitrary types of inputs and outputs, i.e., scalars or complex values.

Let $\varphi$ be a formula with free variables $x_1, \ldots, x_n$. If $\sigma$ is a valuation over free($\varphi$), then the notion of the interpretation $(\mathbf{d}, \mathbf{F}, \mathbf{I})$ satisfying $\varphi$ under $\sigma$, denoted as $\mathbf{I} \models_{(\mathbf{d},\mathbf{F})} \varphi[\sigma]$, is defined in the usual manner.

The notion of the answer $q(DB)$ to the query $q$ on the database $DB = (\mathbf{d}, \mathbf{F}, \mathbf{I})$ is defined by

$$q(DB) = \{\sigma(< x_1, \ldots, x_n >) | \mathbf{I} \models_{(\mathbf{d},\mathbf{F})} \varphi[\sigma]\}.$$

A database query can be viewed as a partial function $q$ mapping any database *DB* with interpretation $(\mathbf{d}, \mathbf{F}, \mathbf{I})$ to $q(DB)$.

Let us introduce a complex value database with external functions and illustrate a reasonable (i.e., data-independent) query, in the sense that the only observations it makes about the individual values in the database are their equality, inequality, and applications of the external functions.

**Example 6.** Consider the following database which has nested relations *Product* and *Part*, shown in Fig. 1.

*Product* = (*prodname*, *Warranty*(*premium*, *country*, *w-period*),
*Composition*(*c-name*, *Parts*(*p-name*, *quantity*)),
*Distributor*(*company*, *fee*))
*Part* = (*p-name*, *weight*, *Source*(*company*, *cost*))

A reasonable query in this context is “return the total fee for each product including its required parts’ source fees and distributor’s delivery fee.” This query may be implemented by an external function *f* which applies to product name, Parts, Distributor and Source attributes, and calculates total values for each product by using fee and cost information from both relations. The result of the query is shown in Fig. 1c.

(a) *Product*

| prodname | Warranty | | | Composition | | | Distributor | |
|---|---|---|---|---|---|---|---|---|
| | premium | country | w-period | c-name | Parts | | company | fee |
| | | | | | p-name | quantity | | |
| prod-A | $120 | U.S.A. | 5 yrs. | c1 | p1 | 6 | comp-A | $500 |
| | | | | | p2 | 2 | | |
| prod-B | $200 | Aust. | 3 yrs. | c2 | p2 | 3 | comp-B | $600 |
| | | | | | p3 | 4 | | |
| | | | | | p4 | 1 | | |
| . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . |
| . | . | . | . | . | . | . | . | . |

(b) *Part*

| p-name | weight | Source | |
|---|---|---|---|
| | | company | cost |
| p1 | 200g | comp-A | $10 |
| p2 | 350g | comp-B | $20 |
| p3 | 300g | comp-C | $30 |
| p4 | 250g | comp-A | $40 |
| . | . | . | . |
| . | . | . | . |
| . | . | . | . |

(c) $q(DB)$

| prodname | total fee |
|---|---|
| prod-A | $600 |
| prod-B | $820 |
| . | . . |
| . | . |
| . | . |

Fig. 1. Product database.

## 3.4. The calculus ($CALC^{cv}$) + fix-point

We use an equivalent but logic-oriented construct to augment the calculus. The construct, called a *fixpoint operator*, allows the iteration of calculus formulas up to a fixpoint. Fix-point operators are redundant in the context of complex value query languages. However, a fix-point construct provides a tractable form of recursion, e.g., it can express transitive closure in polynomial space (time) and yield languages which are well-behaved with respect to expressive power [15]. We review *inflationary* and *non-inflationary* extensions of the calculus with recursion.

*Partial fix-point operator and logic*

Let $\mathbf{R}$ be a database schema, and let $T : \{\langle\tau_1, \ldots, \tau_m\rangle\}$ be a sorted relation which is not in $\mathbf{R}$. Let $\mathbf{S}$ denote the schema $\mathbf{R} \cup \{T\}$. Let $\varphi(T)$ be a formula using $T$ and relations in $\mathbf{R}$, with $m$ free variables $x_1 : \tau_1, \ldots, x_m : \tau_m$. Given an instance $\mathbf{I}$ over $\mathbf{R}$, $\mu_T(\varphi(T))$ denotes the relation with sort $\{\langle\tau_1, \ldots, \tau_m\rangle\}$ which is the limit, if it exists, of the sequence $\{J_n\}_{n \geqslant 0}$ defined by

$$J_0 = \emptyset;$$
$$J_n = \varphi(J_{n-1}), \quad n > 0,$$

where $\varphi(J_{n-1})$ denotes the result of evaluating $\varphi$ on the instance $\mathbf{J}_{n-1}$ over $\mathbf{S}$ whose restriction to $\mathbf{R}$ is $\mathbf{I}$ and $\mathbf{J}_{n-1}(T) = J_{n-1}$. The expression $\mu_T(\varphi(T))$ can be used as a term or as a relation in more complex formulas like any other relation. For example, if $x$ is a variable of sort $\{\langle\tau_1, \ldots, \tau_m\rangle\}$ then $x = \mu_T(\varphi(T))$ is a fix-point formula.

The extension of the calculus with $\mu$ is called *partial fix-point logic*, denoted as $\text{CALC}^{cv} + \mu$. $\text{CALC}^{cv} + \mu$ *formulas* are built by repeated applications of $\text{CALC}^{cv}$ operators and the partial fix-point operator, starting from atoms. Partial fix-point operators can be nested. $\text{CALC}^{cv} + \mu$ *queries* over a database schema $\mathbf{R}$ are expressions of the form

$$\{(x_1, \ldots, x_n) | \xi\},$$

where $\xi$ is a $\text{CALC}^{cv} + \mu$ formula.

*Inflationary fix-point operators and logic*

Adding $\mu^+$ instead of $\mu$ to CALC$^{cv}$ yields the *inflationary fix-point logic*, denoted as by CALC$^{cv}$+ $\mu^+$. The definition of $\mu^+_T(\varphi(T))$ is identical to that of the partial fix-point operator except that the sequence $\{J_n\}_{n\geqslant 0}$ is defined as follows:

$$J_0 = \emptyset;$$
$$J_n = J_{n-1} \cup \varphi(J_{n-1}), \quad n > 0.$$

Note that the sequence $\{J_n\}_{n\geqslant 0}$ is increasing: $J_{i-1} \subseteq J_i$ for each $i > 0$, and the sequence converges in all cases.

**Example 7.** The transitive closure of a graph $G$ is defined by the following CALC$^{cv}$ + $\mu^+$ query:

$$\{\langle x,y\rangle | \mu^+_T(G(x,y)) \vee \exists z(T(x,z) \wedge G(z,y))(x,y)\}.$$

The relation $J_n$ holding pairs of nodes at distance at most $n$ can be defined inductively using the above formula as follows:

$$J_0 = \emptyset;$$
$$J_n = J_{n-1} \cup \varphi(J_{n-1}), \quad n > 0.$$

Here $\varphi(J_{n-1})$ denotes the result of evaluating $G(x,y) \vee \exists z(T(x,z) \wedge G(z,y))$ when the value of $T$ is $J_{n-1}$. As the sequence $\{J_n\}_{n\geqslant o}$ converges, there exists some $k$ for which $J_k = J_j$ for every $j > k$. Clearly, $J_k$ holds the transitive closure of the graph and is also a fixpoint of $\varphi(T)$.

## 3.5. Domain-independent computable queries

There are two important interpretations of calculus queries. Under the active-domain semantics, all quantified variables range over the active domain of a database. Under the natural semantics, all quantified variables range over infinite set $\mathbf{d} \subseteq \mathbf{dom}$. It is easy to write queries with undefined output under natural interpretation. Although the active domain semantic has the advantage that the output is always defined, the active domain information is not readily available to users. One approach to resolving this problem is to consider the class of queries that yield the same output on all possible underlying domains [12,16,26,28,29]. This property usually imposed upon queries is called *domain independence*, meaning that the answer of the query depends only on the active domain of the input instance and not on the underlying universe. We give its definition as follows.

**Definition 10.** A calculus query $q$ is *domain independent* if for any databases $DB_1 = (\mathbf{d}_1, \mathbf{I})$ and $DB_2 = (\mathbf{d}_2, \mathbf{I})$ (i.e., same instances, but different domains), where $adom(q, \mathbf{I}) \subseteq \mathbf{d}_1, \mathbf{d}_2 \subseteq \mathbf{dom}$, then $q(DB_1) = q(DB_2)$.

A query is *computable* if there exists a Turing Machine which, when started with a natural encoding of a database instance **I** on its tape, halts with an encoding of $q(\mathbf{I})$ on the tape, or diverges, when $q(\mathbf{I})$ is undefined. That is, the query must be "implementable" by a Turing Machine.

## 4. Embedded allowed formulas

This section explores the issue of the semantics of complex value calculus queries in the presence of external functions. As mentioned in Section 1, some calculus queries can not be answered sensibly. For example, the formula $\exists t(p(x,y) \wedge \neg q(u) \wedge t \in u)$ is not domain independent. So it is highly desirable to check whether a formula satisfies the *safety* property when we need to support any complex values and any user-defined functions in a query language. However, domain-independence is undecidable even for the flat relational data model without functions.

Our research focuses on identifying a large decidable subclass of domain independent query formulas. We define the class of *embedded allowed* formulas for the complex value model. We first introduce finiteness dependency and bounding functions and then give the definition of embedded allowed.

## 4.1. Finiteness dependency and bounding function

This section proposes two important concepts, which are used in this paper, namely, finiteness dependency and bounding function.

A key element in our notion of embedded allowed formulas is the definition of the bounding function *bd* which associates *finiteness dependencies* (FinDs) to formulas. First we introduce FinD over basic type and complex value type variables. Note that the finiteness dependencies proposed in [13], [22] and [27] only consider basic types. We extend previous related work and adopt the notion of FinD to complex value calculus formulas.

The answer to $q$ on an input instance $\mathbf{I}$ depends on the *closure* of $adom(q, \mathbf{I})$. This notion for complex objects is reviewed as follows.

Given a database instance $\mathbf{I}$ and a query $q$, let $C_q$ be a set of constants that appear in $q$. Following [23], $term^n(DB)$ for some database $DB$ with interpretation $(\mathbf{d}, \mathbf{F}, \mathbf{I})$ is defined as follows:

$$term^0(DB) \overset{\text{def}}{=} adom(\mathbf{I}, C_q)$$

$$term^{n+1}(DB) \overset{\text{def}}{=} term^n(DB) \cup \{adom(f_i(x)) \mid f_i \in \mathbf{F}; x \in dom(\tau_i, term^n(DB)), i = 1, \ldots, l\},$$

where $adom(\mathbf{I}, C_q)$ are all values in domain $\mathbf{d}$ mentioned in the instance $\mathbf{I}$ and $C_q$.

Intuitively, the $term^i(DB)$ captures the accessible domain of some database $DB$ to level $i$, i.e., it extends the active domain to a bounded distance which includes new values invented by external functions.

To specify a class of dependencies, one must define the syntax and the semantics of the dependencies of concern. This is done next for finite dependency.

**Definition 11.** If $U$ is a set of variables occurred in a formula $\varphi$, then a finite dependency over $U$ is an expression of the form $X \to Y$, where $X$, $Y \subseteq U$.

Intuitively, given values for the variables of the left-hand side of a FinD, the sets of values for the variables of the right-hand side are finite. A formula $\varphi$ satisfies the FinD $\{x_1, x_2\} \to \{y_1, y_2\}$ if for each database $DB$, $x_i$ range over $dom(\tau_i, term^k(DB))$, $i = 1, 2$, then $\varphi$ will be true only on assignments which map $y_i$ into the finite set $dom(\tau'_i, term^{k+l}(DB))$, for some $l$.

**Definition 12.** Let $\varphi$ be a formula and $X \to Y$ a FinD over variable set $X$. A formula $\varphi$ satisfies the *finiteness dependency* $X \to Y$, denoted as $\varphi \models X \to Y$, if for each database $DB = (\mathbf{dom}, \mathbf{F}, \mathbf{I})$ and each $k \geqslant 0$ there is some $l \geqslant 0$ such that $\forall y_i \in Y$, $\sigma(y_i) \in dom(\tau_{y_i}, term^{k+l}(DB))$ whenever $\sigma$ is a variable assignment for $X$ satisfying $\sigma(x_i) \in dom(\tau_{x_i}, term^k(DB))$, $\forall x_i \in X$ and $\mathbf{I} \models \varphi[\sigma]$.

**Example 8.** Let the sort of relation $R$ be $\{\langle z : \mathbf{dom}, x : \{\mathbf{dom}\}\rangle\}$. Consider the formula

$$\varphi \equiv R(z, x) \land z \in x \land \neg Q(z) \land f(z) = y.$$

The values of $z$ and $x$ are restricted in $R$. For any variable assignment $\sigma$, $\sigma(z)$ must be in $dom(\mathbf{dom}, term^0(DB))$; $\sigma(x)$ must be in $dom(\{dom\}, term^0(DB))$. So $\varphi \models \emptyset \to zx$. Given any value for $x$, we can determine the value of $z$ as $z \in x$, i.e., for any variable assignment $\sigma$ for $x$ satisfying $\sigma(x) \in dom(\{\mathbf{dom}\}, term^0(DB))$, $\sigma(z) \in dom(\mathbf{dom}, term^0(DB))$. So $\varphi \models x \to z$. Similarly, the value of $y$ is determined by applying function $f$ to variable $z$, i.e., $\sigma$ is value assignment for $z$ satisfying $\sigma(z) \in dom(\mathbf{dom}, term^0(DB))$, then $\sigma(y) \in dom(\tau_y, term^1(DB))$. So $\varphi \models z \to y$.

FinDs satisfy the properties of functional dependencies [13,28]. We now present the following basic inference rules.

**FD1** (reflexivity) If $Y \subseteq X$, then $X \to Y$,
**FD2** (augmentation) If $X \to Y$, then $XZ \to YZ$, and
**FD3** (transitivity) If $X \to Y$ and $Y \to Z$, then $X \to Z$,

where $X$, $Y$, $Z$ range over sets of variables.

If $\Gamma$ is a set of FinDs and $X \rightarrow Y$ is a FinD then $\Gamma \vdash X \rightarrow Y$ if $X \rightarrow Y$ can be derived from $\Gamma$ using the inference rules given above. If $\Gamma$ is a set of FinDs over a variable set V then the *closure* of $\Gamma$ over V is $\Gamma^{*,V} = \{X \rightarrow Y \mid XY \subseteq V \text{ and } \Gamma \vdash X \rightarrow Y\}$. For a formula $\varphi$, $\Gamma^{*,\varphi}$ is a shorthand for $\Gamma^{*,free(\varphi)}$.

We now introduce the bounding function *bd* in the complex value data model, which associates FinDs with calculus formulas. As defined in [13], FinDs and *bd* are only defined in the relational model. In this paper, we extend it to the complex value data model. The bounding function will produce a set of FinDs for a formula. This information will be helpful for us to specify syntactic restriction for a reasonable query, e.g., embedded allowed query.

| | $\varphi$ | $bd(\varphi)$ |
|---|---|---|
| 1 | $R(t_1, \ldots, t_n)$ | $\{\emptyset \rightarrow X\}^{*,\varphi}$ where $X$ = set of variables that are members of $\{t_1, \ldots, t_n\}$. |
| 2 | $\neg R(\vec{t})$ | $\emptyset^{*,\varphi}$ |
| 3 | $\neg\xi$ | $bd(pushnot(\neg\xi))$ for $\xi$ not of the form $R(\vec{t})$. |
| 4 | $f(t_1, \ldots, t_n) = t$ | $\emptyset^{*,\varphi}$ if $t$ is not a variable, or $t$ is a variable occurring in one of $t_1, \ldots, t_n$. |
| 5 | $f(t_1, \ldots, t_n) = t$ | $\{X \rightarrow t\}^{*,\varphi}$ if $t$ is a variable not occurring in any of $t_1, \ldots, t_n$, where $X$ = set of variables occurring in $t_1, \ldots, t_n$. |
| 6 | $x = y$ | $\{x \rightarrow y, y \rightarrow x\}^{*,\varphi}$ |
| 7 | $t_1 \neq t_2$ | $\emptyset^{*,\varphi}$ |
| 8 | $\xi_1 \wedge \ldots \wedge \xi_n$ | $(bd(\xi_1) \cup \ldots \cup bd(\xi_n))^{*,\varphi}$ |
| 9 | $\xi_1 \vee \ldots \vee \xi_n$ | $(bd(\xi_1) \otimes \ldots \otimes bd(\xi_n))^{*,\varphi}$ |
| 10 | $\exists\vec{x}\xi$ | $(bd(\xi)$ minus all FinDs in which some variable in $\vec{x}$ occurs$)^{*,\varphi}$ |
| 11 | $\forall\vec{x}\xi$ | $(bd(\xi)$ minus all FinDs in which some variable in $\vec{x}$ occurs$)^{*,\varphi}$ |
| 12 | $x \subseteq \{y \mid \phi(y)\}$ | $\{X \rightarrow x\}$ if $x$ is a variable not occurring in $\phi$, $X$ = set of variables occurring in $\phi$. |
| 13 | $\forall y(y \in x \rightarrow \phi(y))$ | $\{X \rightarrow x\}$ if $x$ is a variable not occurring in $\phi$, $X$ = set of variables occurring in $\phi$. |
| 14 | $t \in t'$ | $\{X \rightarrow t\}^{*,\varphi}$ if $t$ is a variable not occurring in $t'$, where $X$ = set of variables occurring in $t'$. |
| 15 | $t \in t'$ | $\emptyset^{*,\varphi}$ if $t$ is not a variable or $t$ is a variable occurring in $t'$. |
| 16 | $t \subseteq t'$ | $\{X \rightarrow t\}^{*,\varphi}$ if $t$ is a variable not occurring in $t'$, where $X$ = set of variables occurring in $t'$. |
| 17 | $t \subseteq t'$ | $\emptyset^{*,\varphi}$ if $t$ is not a variable or $t$ is a variable occurring in $t'$. |
| 18 | $x.i$ | if $x$ is of type $< \tau_1, \ldots, \tau_n >$, then for each $i$, $1 \leq i \leq n$, $x \rightarrow x.i$. |
| 19 | $x$ | if $x$ is of type $< t_1, \ldots, t_n >$, $\emptyset \rightarrow x.i$ for each $i$ then $\emptyset \rightarrow x$. |

Fig. 2. The overall definition of the function *bd*.

The set of FinDs derived from a *bd* function provide us finite dependency information among sets of variables. In the complex value model, we need to consider constructed terms, such as $x{\cdot}A$, and complex terms, such as $x \subseteq \{y \mid \phi(y)\}$, in addition to positive literals of the complex value calculus. In Fig. 2, the formulas 1–11 and their associated functions *bd* were presented in [13]. We add formulas 12–19 for the complex value model. The finite dependency information will be useful to determine whether a given variable ranges over a bounded set of possible values. For example, consider $t \in t'$. If $\sigma$ is a variable assignment for the set $X$ of variables occurring in $t'$ such that $\sigma[X] \subseteq dom(\tau_{t'}, term^k(DB))$, then $\sigma[t]$ is bound to the finite domain $dom(\tau_t, term^k(DB))$. Intuitively, it means that variable $t$ will be bound to the finite domain once the sets of values of variables occurring in $t'$ are finite.

Given a formula $\varphi$, bounding function, $bd(\varphi)$, returns a set of FinDs, as shown in Fig. 2.

In Fig. 2, the function *pushnot*$(\neg\varphi)$ is defined as follows: $\neg(\varphi_1 \wedge \varphi_2) = \neg\varphi_1 \vee \neg\varphi_2$; $\neg(\varphi_1 \vee \varphi_2) = \neg\varphi_1 \wedge \neg\varphi_2$; $\neg\exists x\varphi = \forall x\neg\varphi$; $\neg\forall x\varphi = \exists x\neg\varphi$; $\neg\neg\varphi = \varphi$; $\neg(x = y) = (x \neq y)$; $\neg(x \neq y) = x = y$. The operator $\otimes$ is defined as follows: given sets $\Gamma_1, \ldots, \Gamma_n$ of FinDs, $\Gamma_1 \otimes \cdots \otimes \Gamma_n = \{X_1 \ldots X_n \to Y \mid X_i \to Y \in \Gamma_i \text{ for } i \in [1, \ldots, n]\}$ For example, $\varphi = f(x) = y \vee y \in z$, then $bd(\varphi) = (\{x \to y\}^* \otimes \{z \to y\}^*)^* = \{xz \to y\}^*$.

## 4.2. Embedded allowed formulas

This section presents a generalized notion of embedded allowed for the complex value model. The intuitive idea of our notion is to define a relation between variables and formulas in order to ensure that the scope of every quantified variable in a formula is sufficiently restricted for the entire formula to be domain independent.

**Definition 13.** A formula $\varphi$ is *embedded allowed* (*em-allowed*) if the following conditions hold:

(a) $bd(\varphi) \models \emptyset \to free(\varphi)$;
(b) for each sub-formula $\exists\vec{x}\psi$ of $\varphi$, $bd(\psi) \models free(\exists\vec{x}\psi) \to [\vec{x} \cap free(\psi)]$;
(c) for each sub-formula $\forall\vec{x}\psi$ of $\varphi$, $bd(\neg\psi) \models free(\forall\vec{x}\psi) \to [\vec{x} \cap free(\psi)]$;
(d) for each sub-formula $x \subseteq \{y \mid \phi(y)\}$, $bd(\phi) \models free(\phi) \to y$.

**Example 9.** Consider the formula

$$\varphi = p(x,y) \wedge \exists t(u = f(x,y) \wedge \neg q(u,t) \wedge t = g(u)).$$

We have $\emptyset \to xy$, $xy \to u$ and $u \to t$. We can get $\emptyset \to xyu$ which satisfies the condition (a). $u \to t$ satisfies the condition (b). So $\varphi$ is em-allowed.

**Example 10.** Consider the formula

$$\varphi(x,y) = \exists z\exists u\phi(x,y), \text{ where } \phi(x,y) = (R(z) \wedge S(u) \wedge x \subseteq \{y | (y + z.A) \in u.C\}),$$

$bd(\phi) = \{\emptyset \to zu, zu \to y, zuy \to x\}^{*,\phi}$. We have $bd(\varphi) \models \emptyset \to xy$ (i.e., $\emptyset \to free(\varphi)$) which satisfies the condition (a), $bd(\phi) \models \emptyset \to zu$, which satisfies the condition (b) (i.e., $free(\exists z\exists u\phi) \to \{z,u\}$) and $zu \to y$ which satisfies the condition (d). So $\varphi$ is em-allowed.

**Example 11.** The following formula is not em-allowed.

$$\forall y(y \in x \to R(y)).$$

Let $\phi = (y \in x \to R(y))$. Note that $bd(\neg\phi) = bd(\neg(\neg(y \in x) \vee R(y))) = bd(y \in x \wedge \neg R(y)) = \{x \to y\}$ which satisfies condition (c). However, $\emptyset \to x$ does not hold. It violates condition (a).

**Example 12.** Recall query 2 stated in Section 1 which contains relations $R(w,y)$ ($w$ requires $y$) and $S(x,y)$ ($x$ supplies $y$). A natural variant of query 2 is the following question: "Does some supplier supply all parts required by project ANU2000?" This query can be expressed by $\varphi = \exists x\forall y[R(ANU2000,y) \to S(x,y)]$. We see that $\varphi$ is not em-allowed because the criterion (b) dose not hold. The reason is $bd(R(ANU2000,y) \to S(x,y)) = bd(\neg R(ANU2000,y) \vee S(x,y)) = bd(\neg R(ANU2000,y)) \otimes bd(S(x,y)) = \emptyset \otimes \{\emptyset \to xy\} = \emptyset$.

The safety condition and the equivalence of the domain-independent CALC$^{cv}$, the safe CALC$^{cv}$ and the ALG$^{cv}$ have been studied in [1]. The syntactic condition known as *safety* ensures that each variable is range restricted, in the sense that relative to a given ordering, it is restricted by the formula to lie within the active domain of the query or the input. For example, in the case $x = f(x_1, \ldots, x_k)$ in a formula $F$, $x$ is restricted if all the $x_i$ precede $x$ in the ordering. A formula is safe relative to a given partial ordering if all the variables are restricted in it. It is easy to check that every safe formula defined in [1] is em-allowed.

## 5. Translation of embedded allowed calculus queries

This section describes a non-trivial procedure for translating embedded allowed (complex value) formulas (Definition 13) into equivalent algebra queries. Our algorithm consists of the following four steps.

Step 1. First, replace all sub-formulas of the form $\forall y(y \in x \rightarrow \varphi(y))$ with $x \subseteq \{y \mid \varphi(y)\}$. Next, replace any remaining sub-formula of the form $\forall\phi$ by $\neg\exists\neg\phi$. Finally, rename the quantified variables if necessary.
Step 2. Transform the em-allowed formula $F$ obtained in Step 1 into an equivalent formula $F'$ in Existential Normal Form (ENF).
Step 3. Transform the formula $F'$ into an equivalent Complex Value Algebra Normal Form $\psi$ (ALG$^{cv}$NF).
Step 4. Translate $\psi$ into an equivalent algebraic expression $E_\psi$.

Steps 1–3 are accomplished using families of transformations which map sub-formulas to equivalent sub-formulas, and Step 4 is accomplished using transformations which map sub-formulas to algebraic expressions. The major difference between our algorithm and those of [1,13,29] are as follows.

- The algorithms of [13,29] apply only to the relational model, while our algorithm extends their work to the complex value model.
- The algorithm of [29] does not consider external functions. The algorithm of [13] considers formulas incorporated with scalar functions. In our algorithm, we consider domain independent queries with external functions in a general setting, by allowing the inputs and outputs of the external functions to be scalar values, sets, nested sets, etc.
- The paper [1] provides a translation from range-restricted calculus formulas into the complex value algebra. This translation is motivated from a primarily theoretical perspective. Our algorithm adopts the finite dependency approach, not considered in [1], which influences the choice of some transformation steps.
- In Step 3, we introduce transformations T17 and T18 not included in the relational model [13,29].

A rather involved construction is now presented for translating em-allowed queries into the algebra. The translation involves four steps which are described in the following four subsections.

### 5.1. Removing universal quantifiers and renaming variables

This step replaces sub-formulas of the form $\forall y(y \in x \rightarrow \varphi(y))$ with $x \subseteq \{y \mid \varphi(y)\}$, replaces any remaining sub-formula of the form $\forall\varphi$ by $\neg\exists\neg\varphi$ and renames the quantified variables if necessary.

**Example 13.** Consider the following formula $\varphi = \exists z R(z) \wedge \forall y(y \in x \rightarrow y \in z)$. This formula can be translated as follows: $\varphi = \exists z R(z) \wedge x \subseteq \{y \mid y \in z\}$.

As shown in the following lemma, applying these transformations preserves the em-allowed property.

**Lemma 1.** *If $\varphi$ is em-allowed and $\varphi_t$ is the result of applying the following two transformations*: (1) *replacing sub-formulas of the form $\forall y(y \in x \rightarrow \varphi(y))$ with $x \subseteq \{y \mid \varphi(y)\}$*, *and* (2) *replacing any remaining sub-formula of the form $\forall\varphi$ by $\neg\exists\neg\varphi$, to $\varphi$, then $\varphi_t$ is em-allowed.*

**Proof.** As shown in Fig. 2, $bd(\forall y(y \in x \to \varphi(y))) = bd(x \subseteq \{y|\varphi(y)\})$.

So applying this transformation preserves the em-allowed property. □

We assume that all the formulas considered in the following three subsections have already been transformed in this manner.

### *5.2. Transforming a formula into existential normal form*

This subsection describes an algorithm that transforms a formula into Existential Normal Form (ENF). First we review the procedure, called "simplification of a formula" [3,13,29], which is an important component in the translation. Then we present additional transformations that are necessary to get the desired form and show that each transformation preserves the em-allowed property.

**Definition 14.** A formula is *simplified* if and only if the following properties hold.

- There is no occurrence of $\neg\neg\varphi$.
- There is no occurrence of $\neg(\tau_1 = \tau_2)$ or $\neg(\tau_1 \neq \tau_2)$, for terms $\tau_1$ and $\tau_2$.
- The operators 'and' ($\wedge$), 'or' ($\vee$) and existential quantifier ($\exists$) are flattened; that is,
  - in subformula $\varphi_1 \wedge \cdots \wedge \varphi_n$, no operand $\varphi_i$ is itself a conjunction,
  - in subformula $\varphi_1 \vee \cdots \vee \varphi_n$, no operand $\varphi_i$ is itself a disjunction, and
  - in subformula $\exists\vec{x}\varphi$, $\varphi$ does not begin with $\exists$.
- In every subformula $\exists\vec{x}\varphi$, each variable $x_i$ is actually free in $\varphi$.

The simplification of a formula is achieved by using the following equivalence-preserving rewriting rules:

(1) replace $\neg\neg\phi$ by $\phi$ ($T_1$); $\neg(t_1 = t_2)$ by $t_1 \neq t_2$ ($T_2$); $\neg(t_1 \neq t_2)$ by $t_1 = t_2$ ($T_3$);
(2) flatten 'and's, 'or's, and existential quantifiers ($T_4$, $T_5$ and $T_6$);
(3) replace $\exists\vec{x}\varphi$ by $\exists\vec{y}\varphi$ where each variable $y_i$ is actually free in $\varphi$. ($T_7$).

It is easy to define an algorithm, SIMPLIFY, which iterates over the above transformations to produce simplified formulas. During the simplification process, if there is sub-formula $x \subseteq \{y \mid \phi(y)\}$ in $\varphi$ then we need to simplify $\phi(y)$ as well.

As shown below, these transformations preserve em-allowedness.

**Lemma 2.** *Given an em-allowed formula $\varphi$, the transfromation using the above rewriting rules $T_1$–$T_7$ yields an equivalent em-allowed simplified formula.*

**Proof.** See Appendix B. □

It is convenient to think of a calculus formula in terms of its parse tree where the leaves of the tree are atomic formulas. For example, we will say that the formula $(R(x) \wedge \exists y(S(y,z)) \wedge \neg T(x,z))$ has 'and' ($\wedge$) as a root (which has an atom, an $\exists$, and a $\neg$ as children). There is a sub-formula which corresponds to each internal node labeled by $\vee, \wedge, \neg, \exists x$ or a sub-formula $x \subseteq \{y \mid \varphi(y)\}$.

We now introduce the concept of Existential Normal Form and give additional transformations to put a formula into this form. Our notion of ENF is slightly different from that of [13], in that we add item 4.

As noted in [13], the simplified formulas can be classified into two groups, positive and negative.

**Definition 15.** A simplified formula $\varphi$ is *negative* if and only if $\varphi \equiv \neg\phi$ for some formula $\phi$ or $\varphi \equiv t_1 \neq t_2$ for some terms $t_1$, $t_2$. A simplified formula is *positive* if it is not negative.

**Definition 16.** A formula $F$ is in *existential normal form* (ENF) if and only if the following conditions hold:

(1) It is simplified.
(2) Each disjunction $(t_1 \vee, \ldots, \vee t_n)$ in the formula satisfies:
  - the parent of the disjunction, if it has one, is $\wedge$, and
  - each operand of the disjunction is a positive formula.
(3) The parent, if any, of a conjunction $(t_1 \wedge, \ldots, \wedge t_n)$ of negative formulas is $\exists$.
(4) For each sub-formula $x \subseteq \{y \mid \varphi(y)\}$ in the formula $F$, $\varphi$ is in ENF.

We present the algorithm ENF, that transforms a formula into an equivalent ENF formula by simplifying the formula then alternating between applying transformations T8–T12 and simplifying the formula, until a fixed point is reached. The result of ENF on input $\varphi$ is denoted ENF($\varphi$).

```
Algorithm (ENF)
Input: Formula without universal quantifiers, $\varphi$
Output: An ENF formula equivalent to $\varphi$
begin
  $\varphi$:=SIMPLIFY($\varphi$)
  while some sub-formula $\chi$ of $\varphi$ matches any one of the following do
  caseT8: $\chi = \neg(\psi_1 \wedge \ldots \wedge \psi_n)$, where for each $i$, $\psi_i$ is negative and $\psi_i \equiv \neg\xi_i$;
          $\varphi$:=replace $\chi$ in $\varphi$ by $\xi_1 \vee \cdots \vee \xi_n$
      T9: $\chi = \psi \vee \xi_1 \vee \cdots \vee \xi_n$, where $\psi$ is negative and $\psi \equiv \neg\xi$;
          $\varphi$:=replace $\chi$ in $\varphi$ by $\neg(\xi \wedge \neg\xi_1 \wedge \cdots \wedge \neg\xi_n)$
      T10: $\chi = (\theta_1 \wedge \cdots \wedge \theta_k) \vee \xi_1 \vee \cdots \vee \xi_n$, where $\theta_i$ is negative and $\theta_i \equiv \neg\lambda_i$;
          $\varphi$:=replace $\chi$ in $\varphi$ by $\neg((\lambda_1 \wedge \cdots \vee \lambda_k) \wedge \neg\xi_1 \wedge \ldots \wedge \neg\xi_n)$
      T11: $\chi = \neg(\psi_1 \vee \cdots \vee \psi_n)$;
          $\varphi$:=replace $\chi$ in $\varphi$ by $(\neg\psi_1 \wedge \cdots \wedge \neg\psi_n)$
      T12: $\chi = \exists\vec{x}(\psi_1 \vee \cdots \vee \psi_n)$;
          $\varphi$:= replace $\chi$ in $\varphi$ by $(\exists\vec{x}_1\psi_1 \vee \cdots \vee \exists\vec{x}_h\psi'_h)$, where $\vec{x}_i$ are variables not occurring in the formula
and $\psi'_1$ is the result of renaming $\vec{x}$ by $\vec{x}_i$.
  end case
  if there is sub-formula $\chi = x \subseteq \{y \mid \phi(y)\}$ in $\varphi$ then
    $\varphi$:=replace $\chi$ in $\varphi$ by $x \subseteq \{y \mid \text{ENF}(\phi(y))\}$
  return $\varphi$
end
```

**Example 14.** Consider the formula $\varphi = \neg(\neg(f(x) = y \wedge x \in z) \wedge \neg T(x)) \wedge S(z)$ It can be translated into $\varphi' = ((f(x) = y \wedge x \in z) \vee T(x)) \wedge S(z)$ which is in ENF.

**Lemma 3.** *Given an em-allowed formula $\varphi$, the algorithm ENF terminates and yields an equivalent formula in ENF that is em-allowed.*

**Proof.** See Appendix B. □

## 5.3. Transforming an ENF formula into Complex Value Algebra Normal Form

This subsection presents the translation of an ENF formula into a formula in Complex Value Algebra Normal Form (ALG$^{cv}$NF). We start by defining the Complex Value Algebra Normal Form, and then give necessary transformation rules not present in the relational model.

**Definition 17.** A sub-formula of a simplified formula is *restrictive* if its parent is $\wedge$ and it is either negative or $t_1 = t_2$ or $x < y$. Any other formula is called *constructive*.

**Example 15.** Let $F$ be the formula $(p(x) \vee \neg q(x)) \wedge \neg r(x) \wedge x < y \wedge x \subseteq \{t|s(z) \wedge t \in z\}$.

Then the constructive sub-formulas of $F$ are $p(x)$, $q(x)$, $\neg q(x)$, $(p(x) \vee \neg q(x))$, $r(x)$, $x \subseteq \{t \mid s(z) \wedge t \in z\}$, $s(z)$, $t \in z$, $s(z) \wedge t \in z$ and $F$ itself. The sub-formula $\neg r(x)$ and $x < y$ are restrictive.

Our aim is to transform a given em-allowed query into an equivalent query, all of whose constructive sub-queries can be effectively translated. Intuitively, for each constructive sub-formula, we hope to build an equivalent complex value algebra expression. If a sub-formula is restrictive, it will be used to augment an algebra query already constructed from its siblings. During this step, the function *bd* is crucial to decide whether a sub-formula is em-allowed.

**Definition 18.** An em-allowed formula $F$ is in *complex value algebra normal form* (ALG$^{cv}$NF) if $F$ is in ENF and every constructive sub-formula of $F$ is em-allowed.

Now we present the algorithm ALG$^{cv}$NF, that transforms an em-allowed formula in ENF into an equivalent formula which is in ALG$^{cv}$NF. The output of ALG$^{cv}$NF on input $\varphi$ is denoted ALG$^{cv}$NF($\varphi$).

```
Algorithm (ALG$^{cv}$NF)
Input: Formula without universal quantifiers, $\varphi$
Output: An ALG$^{cv}$NF formula equivalent to $\varphi$
begin
  while some sub-formula $\chi_0$ of $\varphi$ matches any one of the following do
  case T13: $\chi_0 = \exists\vec{y}\phi \wedge \xi_1 \wedge \cdots \wedge \xi_n$, where $\phi$ is not em-allowed;
      $\alpha$:=ALG$^{cv}$NF(SIMPLIFY($\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$)), where $\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$ is em-allowed
      $\chi_t := \exists\vec{y}\alpha \wedge \xi_{i_{k+1}} \wedge \cdots \wedge \xi_{i_n}$
      $\varphi$:=replace $\chi_0$ by $\chi_t$ in $\varphi$
    T14:$\chi_0 = (\phi_1 \vee \cdots \vee \phi_m) \wedge \xi_1 \wedge \cdots \wedge \xi_n$, where $\phi_1 \vee \cdots \vee \phi_m$ is not em-allowed;
      for $i = 1, m$ do
        $\alpha_i$:=ALG$^{cv}$NF(SIMPLIFY($\phi_i \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$)), where $\phi_i \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$ is em-allowed
      $\chi_t := (\alpha_1 \vee \cdots \vee \alpha_m) \wedge \xi_{i_{k+1}} \wedge \cdots \wedge \xi_{i_n}$
      $\varphi$:=replace $\chi_0$ by $\chi_t$ in $\varphi$
      $\varphi$:=ENF($\varphi$)
    T15: $\chi_0$=$\neg\phi \wedge \xi_1 \cdots \wedge \xi_n$, where $\phi$ is not em-allowed;
      $\alpha$:=ALG$^{cv}$NF(SIMPLIFY($\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$)), where $\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$ is em-allowed
      $\beta$:=ALG$^{cv}$NF($\xi_1 \wedge \cdots \wedge \xi_n$)
      $\chi_t$:=$\neg\alpha \wedge \beta$
      $\varphi$:=replace $\chi_0$ by $\chi_t$ in $\varphi$
    T16 $\chi_0 = R(\tau_1, \ldots, \tau_n) \wedge \xi_1 \wedge \cdots \wedge \xi_m$, where $R(\tau_1, \ldots, \tau_n)$ is not em-allowed;
      $\chi_t := \exists z_1, \ldots, z_n(R(z_1, \ldots, z_n) \wedge z_1 = \tau_1 \wedge \cdots \wedge z_n = \tau_n \wedge \xi_1 \wedge \cdots \wedge \xi_m)$
      $\varphi$:=replace $\chi_0$ by $\chi_t$ in $\varphi$
    T17: $\chi_0 = \xi_1 \wedge \cdots \wedge \xi_m \wedge x \subseteq \{y \mid \phi(y)\}$, where $\phi$ is not em-allowed;
      $\alpha$:=ALG$^{cv}$NF(SIMPLIFY($\phi(y) \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$))
      $\chi_t := \xi_1 \wedge \cdots \wedge \xi_m \wedge x \subseteq \{y \mid \alpha\}$
      $\varphi$:=replace $\chi_0$ by $\chi_t$ in $\varphi$
    T18: $\chi_0 = \psi \wedge x \subseteq \{y \mid \phi(y) \wedge \xi_1 \wedge \cdots \wedge \xi_m\}$, where $\psi$ is not em-allowed;
      $\alpha$:=ALG$^{cv}$NF(SIMPLIFY($\psi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$))
      $\chi_t := \alpha \wedge x \subseteq \{y \mid \phi(y) \wedge \xi_1 \wedge \cdots \wedge \xi_m\}$
      $\varphi$:=replace $\chi_0$ by $\chi_t$ in $\varphi$
  end case
  end while
  return $\varphi$
end
```

**Example 16.** Consider the following formula: $\varphi_1 = (f(y) = z \wedge x + 5 = y) \wedge \exists w(R(x,w) \wedge \neg S(y))$.

As $(R(z,w) \wedge \neg S(y))$ is not em-allowed, we have to apply T13 of the algorithm ALG$^{cv}$NF. We get

$$\varphi_1' = (f(y) = z \wedge x + 5 = y) \wedge \exists w(R(x,w) \wedge \neg S(y) \wedge x + 5 = y).$$

**Example 17.** Consider the following formula:

$$\exists u \exists s(S(u) \wedge R(s) \wedge x \subseteq \{y | (y + s{\cdot}A) \in u{\cdot}C\} \wedge (t \in x \wedge \neg Q(t))).$$

As $((y + s{\cdot}A) \in u{\cdot}C)$ is not em-allowed, we apply the rule T17 to obtain

$$\exists u \exists s(S(u) \wedge R(s) \wedge x \subseteq \{y \mid S(u) \wedge R(s) \wedge (y + s{\cdot}A) \in u{\cdot}C\} \wedge (t \in x \wedge \neg Q(t))),$$

which is in ALG$^{cv}$NF.

**Lemma 4.** *Given an ENF formula $\varphi$ that is em-allowed, ALG$^{cv}$NF($\varphi$) terminates and yields an equivalent formula in Complex Value Algebra Normal Form.*

**Proof.** See Appendix B. □

## 5.4. Transforming ALG$^{cv}$NF formulas into algebraic expressions

This subsection describes the translation of ALG$^{cv}$NF formulas into equivalent complex value algebraic queries. The idea is to translate each constructive sub-formula into an algebraic expression that represents the values of the free variables in the formula that satisfies it.

In general, given an ALG$^{cv}$NF formula $\varphi$ with free variables $x_1, \ldots, x_n$, we shall construct an algebraic expression $E_\varphi$ over attributes $x_1, \ldots, x_n$, such that for each database input $DB$

$$E_\varphi(DB) = \{x_1, \ldots, x_n | \varphi\}(DB).$$

As in the relational model, a crucial aspect in translating an ALG$^{cv}$NF formula $\varphi_1 \wedge \cdots \wedge \varphi_n$ into the algebra is sorting the conjuncts to a modified form so that each conjunct uses only variables from preceding conjuncts. For example, each free variable in a conjunct of the form $\neg\xi$ occurs in some preceding conjunct. Then translation of these modified formulas can be effectively performed sequentially starting from the first conjunct.

**Definition 19.** A formula $\varphi$ is in *modified* ALG$^{cv}$NF if it is in ALG$^{cv}$NF and each polyadic “and” sub-formula, $\varphi_1 \wedge \cdots \wedge \varphi_n$, is ordered and for each $j \in [1,n]$, the prefix $\varphi_1 \wedge \cdots \wedge \varphi_j$ is em-allowed.

If $\varphi$ is a formula in Relational Algebra Normal Form, such an appropriate ordering can be found. There is similar result for formulas in Complex Value Algebra Normal Form.

Before presenting the translation of calculus formulas into equivalent algebraic expressions, we review an algebraic operator called the *replace* operator [1] which will be used in our translation process.

**Definition 20.** If $r$ is a relation and $f$ is a function, then $\rho\langle f\rangle$ is a replace operation defined as

$$\rho\langle f\rangle(r) = \{f(t) \mid t \in r\}.$$

**Example 18.** Suppose that there is a relation $s$ with sort $S : \{\langle A : \mathbf{dom}, B : \{\mathbf{dom}\}\rangle\}$. The relation of sort $\{\langle B : \{\mathbf{dom}\}\rangle\}$, obtained by adding the value of the $A$ component to the $B$ component and deleting the $A$ component, is given by $\rho\langle B \cup \{A\}\rangle(s)$.

Translating an ALG$^{cv}$NF formula into an equivalent algebraic expression can be performed by applying the following method: translate conjunctions into joins or Cartesian products, negations into generalized differences (**diff**) [29], existential quantifiers into projections, inequalities into selections and equalities and arithmetic operations into *appends*. “Append” is a relational operator defined in [27].

**Definition 21.** If $r$ is a relation of $l$-tuples, then the append operator,$\Delta_{\delta(i_1,\ldots,i_k)}(r)$ is a set of $l+1$ tuples, $k \leqslant l$, where $\delta$ is an arithmetic operator on the components $i_1, \ldots, i_k$. The last component of each tuple is the value of $\delta(i_1, \ldots, i_k)$.

**Example 19.** Let the sort of a relation $r$ be $R : \{\langle X : \mathbf{dom}, Y : \mathbf{dom}\rangle\}$. A value of this sort is $\{\langle X : 2, Y : 3\rangle, \langle X : 5, Y : 4\rangle\}$. Then $\Delta_{X+Y=Z}(r) = \{\langle X : 2, Y : 3, Z : 5\rangle, \langle X : 5, Y : 4, Z : 9\rangle\}$.

Attribute renaming is also needed for query translation. A renaming operator for a finite set $U$ of attributes is an expression $\zeta_f$, where $f$ is an attribute renaming for $U$. $f$ is usually written as $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_n$ to indicate $f(A_i) = B_i$.

We first consider atoms that can be translated independently of a surrounding formula.

| Sub-formula to translate | Algebraic expression |
|---|---|
| $R(\vec{t})$ | $\pi_{\vec{x}}(\sigma_\theta(R))$ |
| $x \in t$ | $set\_destroy(E_t)$ |
| $x = t$ | $\{\langle t\rangle\}\vert_{(\mathbf{d},\mathbf{F})}$ i.e., the expression denoting the unary relation containing a single tuple with value $t$ as evaluated under $(\mathbf{d},\mathbf{F})$ |
| $x \subseteq t$ | $powerset(E_t)$ |
| $x \subseteq \{y \mid \phi(y)\}$ | $\rho\langle powerset\rangle(nest_{X=y}(E_\phi))$ |

We next consider non-atomic constructive sub-formulas.

| Sub-formula to translate | Algebraic expression |
|---|---|
| $\xi_1 \vee \cdots \vee \xi_n$ | $E_{\xi_1} \cup \cdots \cup E_{\xi_n}$ |
| $\exists\vec{x}\varphi$ | $\pi_{free(\varphi)-\{\vec{x}\}}(E_\varphi)$ |
| $\neg\xi$ | $\{\langle\rangle\} - E_\xi$ |

Finally, we give the translation for formulas of the form $\varphi_1 \wedge \cdots \wedge \varphi_n$, where $n > 1$. We only consider the case of $\psi \wedge \xi$ here, as it can be easily extended to the form $\varphi_1 \wedge \cdots \wedge \varphi_n$.

| Sub-formula to translate | Algebraic expression |
|---|---|
| $\psi \wedge \xi$ | $\sigma_\theta(E_\psi)$, $\theta$ is a selection list derived from $t_1 = t_2$. if $\xi$ has the form $t_1 = t_2$ or $x \neq t$, where $t_1$ and $t_2$ have only variables in $\psi$. |
| | $\Delta_\delta(E_\psi)$, $\delta$ is a function derived from term $t$. if $\xi$ has the form $t = x$, variable $x$ is not free in $\psi$, and all variables in $t$ are free in $\psi$. |
| | $E_\psi \bowtie E_\xi$ if $\psi$ and $\xi$ are both constructive and have overlapping sets of free variables. |
| | $E_\psi \times E_\xi$ if $\psi$ and $\xi$ are both constructive and do not have overlapping sets of free variables. |
| $\psi \wedge \neg\xi'$ | $E_\psi$ **diff** $E_{\xi'}$ if $free(\xi') \subset free(\psi)$. |
| | $E_\psi - E_{\xi'}$ if $free(\xi') = free(\psi)$. |

Let ALG$^{cv}$NF formula $\varphi$ be fixed. The construction of $E_\varphi$ is inductive, from leaf to root, and can be conducted by the above methods.

We conclude the section by presenting some examples.

**Example 20.** The formula $\varphi \equiv f(x) = y \wedge \neg P(x,z) \wedge g(x,y) = z \wedge R(x)$ can be re-ordered as $\varphi' \equiv R(x) \wedge f(x) = y \wedge g(x,y) = z \wedge \neg P(x,z)$ In turn this can be translated into an algebraic expression as follows:

$$E_1 := R,$$
$$E_2 := \Delta_{f(x)}(E_1),$$
$$E_3 := \Delta_{g(x,y)}(E_2),$$
$$E_4 := E_3 \textbf{ diff } P.$$

**Example 21.** Consider the formula $\varphi = \exists z(r(z) \wedge x \subseteq \{y \mid y \in z\})$ The algorithm $ALG^{cv}NF$ will apply $T_{17}$ to obtain $\varphi' = \exists z(r(z) \wedge x \subseteq \{y \mid r(z) \wedge y \in z\})$ The range formula for $x$ contains the free variables $z$ and $y$. So the type of the corresponding algebra query is a set of pairs, $(z,y)$-values. By the above algorithm (translation into the algebra), we need *nest* and *powerset* operators. Finally, we join $r$ with the algebra query we obtained for the range formula for $x$, and then perform projection for the existential quantifier $z$. The equivalent algebra query is obtained using the program

$$E_1 := r \times \zeta_{z \to y}(set\_destroy(r)),$$
$$E_2 := nest_{X=y}(E_1),$$
$$E_3 := \rho\langle powerset(X)\rangle(E_2),$$
$$E_4 := r \bowtie E_3,$$
$$E_5 := \pi_X(E_4).$$

Based on the translation described in this section, we present the following result.

**Theorem 5.** *For each embedded allowed calculus formula $\varphi$, the translation algorithms described in this section can effectively translate $\varphi$ to an equivalent algebraic expression.*

**Proof.** Based on Lemmas 1–4 and algorithms described in this section, we conclude that formula $\varphi$ can be effectively translated to an equivalent algebraic expression. □

## 6. Relationship between embedded allowed and domain independence

We have presented the translation of em-allowed complex value formulas into equivalent algebra queries in Section 5. In this section, we present the main results of relationship between embedded allowed formulas and domain independence. We first show that every em-allowed formula is embedded domain independent and external-function domain independent. We then prove that all em-allowed queries in $\text{CALC}^{cv} + \mu^+$ are external-function domain independent and continuous.

### 6.1. Embedded and external-function domain independence

Before presenting our main results, we need to review two notions of domain independence. The notion of "embedded domain independence" was proposed to generalize "domain independence" to incorporate external functions [1,13]. The fundamental idea behind this notion is that, for any query $q$, there is a bound on the number of times functions (and their inverses if there exist) can be applied. The answer to $q$ on an input instance $\mathbf{I}$ depends on the *closure* of $adom(q, \mathbf{I})$.

**Definition 22.** Two databases $DB_1 = (\mathbf{d}_1, \mathbf{F}_1, \mathbf{I})$ and $DB_2 = (\mathbf{d}_2, \mathbf{F}_2, \mathbf{I})$ agree on $adom(\mathbf{I}, C_q)$ to level $n$ if

- $adom(\mathbf{I}, C_q) \subseteq \mathbf{d}_1 \cap \mathbf{d}_2$ and
- $\forall x \in dom(\tau_i, term^n(DB)), f_i \in \mathbf{F}_1, f_i' \in \mathbf{F}_2, f_i(x) = f_i'(x)$, i.e., $f_i$ and $f_i'$ agree on any input whose atomic values are in $term^n(DB)$.

We now review the notion of embedded domain independence.

**Definition 23.** A calculus query $q$ is *embedded domain independent at level* $n$ if, for all interpretations $S_1 = (\mathbf{d}_1, \mathbf{F}_1, \mathbf{I})$ and $S_2 = (\mathbf{d}_2, \mathbf{F}_2, \mathbf{I})$ which agree on $atom(\mathbf{I}, C_q)$ to level $n$, $q$ yields the same output on $S_1$ and $S_2$. The query $q$ is *embedded domain independent* if for some $n$ it is embedded domain independent at level $n$.

Next we will review the notion of external-function domain independent queries proposed by Suciu [23]. Let $DB_1$, $DB_2$ be two databases with interpretations $(\mathbf{d}_1,\mathbf{F}_1,\mathbf{I}_1)$, $(\mathbf{d}_2,\mathbf{F}_2,\mathbf{I}_2)$ respectively. A *morphism* $\xi$: $DB_1 \rightarrow DB_2$ is a partial injective function $\xi : \mathbf{d}_1 \rightarrow \mathbf{d}_2$, which can be lifted from the base type to partial functions at any type $\tau$, $\xi_\tau$: $dom(\tau, D) \rightarrow dom(\tau, D')$, such that

- for every $i$, $\xi(R_i)$ is defined and $\xi(R_i) = R'_i$, where $R_i \in \mathbf{I}_1$, $R'_i \in \mathbf{I}_2$, and
- for any $x \in dom(\tau, \mathbf{d}_1)$, if $f'_j(\xi(x))$ is defined then so is $\xi(f_j(x))$ and $f'_j(\xi(x)) = \xi(f_j(x))$, where $f_j \in \mathbf{F}_1, f'_j \in \mathbf{F}_2$.

**Definition 24.** A query $q$ is *external-function domain independent* (*ef-domain independent*) iff for every morphism $\xi : DB_1 \rightarrow DB_2$, if $q(DB_1)$ is defined, then so are $\xi(q(DB_1))$ and $q(DB_2)$, and $q(DB_2) = \xi(q(DB_1))$.

We now state two of the main results of this paper.

**Theorem 6.** *Every em-allowed formula is embedded domain independent.*

**Proof.** By Theorem 5 any em-allowed formula can be effectively translated into an equivalent algebra query. Because the algebra is embedded domain independent, $\varphi$ is embedded domain independent. □

**Theorem 7.** *Every em-allowed formula is ef-domain independent.*

**Proof.** By Theorem 5, every em-allowed formula can be translated into an equivalent complex value algebra (ALG$^{cv}$) query with external functions. All queries in the nested relational algebra $\mathcal{NRA}$ + *fix* are *ef*-domain independent [23]. $\mathcal{NRA}$ is equivalent to ALG$^{cv}$ without powerset with external functions.

Now we prove that the query *powerset* is *ef*-domain independent. Consider two databases $DB_1 = (\mathbf{d}_1, \mathbf{F}_1, \mathbf{I}_1)$ and $DB_2 = (\mathbf{d}_2, \mathbf{F}_2, \mathbf{I}_2)$. For any morphism $\xi : DB_1 \rightarrow DB_2$, for every $i$, $\xi(R_i)$ is defined and $\xi(R_i) = R'_i$, where $R_i \in \mathbf{I}_1$, $R'_i \in \mathbf{I}_2$. $\xi(powerset(q(DB_1))) = powerset(q(DB_2))$, for any query $q$. So *powerset* is *ef*-domain independent. Therefore, every em-allowed formula is *ef*-domain independent. □

If all external functions are computable, it is easy to get the following result.

**Corollary 1.** *Every em-allowed formula defines an ef-domain independent, computable query.*

The relationship between these query classes we have discussed is summarized as follows:

em-allowed → embedded domain independent and

em-allowed → ef-domain independent

The notion of em-domain independence is used *only* in conjunction with query languages without iterations and fails when extended to languages with fix-points. The notion of ef-domain independence is more appropriate for queries with external functions than the notion of em-domain independence. For this reason, we also investigate the aspects of calculus queries with fix-points.

The following theorem investigates the aspects of calculus queries with fix-points. We first review the concept of continuous [24]. Let $\mathcal{DB}$ be a set of databases which share the same relations, $R_1, \ldots, R_k$, and differ only in their domains and their external functions. We say that $\mathcal{DB}$ is *directed* iff it is nonempty and $\forall DB_1 = (d_1, f_1, \ldots f_l, R_1, \ldots, R_k)$, $DB_2 = (d_2, g_1, \ldots, g_l, R_1, \ldots, R_k) \in \mathcal{DB}$, there exists $DB = (d, h_1, \ldots, h_l, R_1, \ldots, R_k) \in \mathcal{DB}$ such that $d_1 \subseteq d$; $d_2 \subseteq d$, and $graph(f_i) \subseteq graph(h_i)$; $graph(g_i) \subseteq graph(h_i)$. We define its *least upper bound* as a structure $\mathcal{M} = \bigcup \mathcal{DB} \overset{\text{def}}{=} (\mathbf{d}, f_1, \ldots, f_l, R_1, \ldots, R_k)$ where $\mathbf{d}$ is the union of the domains of all databases in $\mathcal{DB}$, and the graph of $f_i$ is the union of all graphs of the corresponding functions of the databases in $\mathcal{DB}$.

**Definition 25** [24]. A query $q$ is called *continuous* iff for any directed set of database $\mathcal{DB}$, $\bigcup\{q(DB) \mid DB \in \mathcal{DB}\}$ exists and $q(\bigcup \mathcal{DB}) = \bigcup\{q(DB) \mid DB \in \mathcal{DB}\}$.

Intuitively, the concept of continuity captures the property that the result of a query depends only on a finite approximation of the database. During the computation we can inspect only a finite fragment of a potentially infinite input.

Let $\Omega = \{R_1, \ldots, R_n, f_1, \ldots, f_l\}$ be a signature. $\mathcal{NRA}(\Omega)$ is the nested relation algebra over $\Omega$. Suciu [24] proved that all queries in $\mathcal{NRA}(\Omega) + fix$ are ef-domain independent and continuous. We show that all em-allowed queries expressed in $\text{CALC}^{cv} + \mu^+$ are also *ef*-domain independent and continuous.

**Theorem 8.** *All em-allowed queries in* $\text{CALC}^{cv} + \mu^+$ *are ef-domain independent and continuous.*

**Proof.** We first prove that all em-allowed queries $\in \text{CALC}^{cv}$ are *ef*-domain independent and continuous. As demonstrated in the course of translating em-allowed formulas in $\text{CALC}^{cv}$ into equivalent algebra queries in $\text{ALG}^{cv}(\Omega)$, every em-allowed query in $\text{CALC}^{cv}$ is equivalent to an algebra query in $\text{ALG}^{cv}(\Omega)$. We know that all queries in the nested relational algebra $\mathcal{NRA}(\Omega) + fix$ are *ef*-domain independent and continuous [23]. $\mathcal{NRA}(\Omega)$ is equivalent to $\text{ALG}^{cv}(\Omega)$ without powerset with external functions.

By Theorem 7, the query *powerset* is *ef*-domain independent. As *powerset* is finite, *powerset* is continuous. Therefore $\text{CALC}^{cv}$ is *ef*-domain independent and continuous. $\text{CALC}^{cv}+\mu^+$ is equivalent to $\text{CALC}^{cv}$ [3]. Therefore, every em-allowed queries in $\text{CALC}^{cv}+\mu^+$ is *ef*-domain independent and continuous. □

## 7. Domain independent query programs

There has been considerable development in deductive databases that use the first-order language as a mathematical notation for describing data. Query evaluation in such a model is the process of proving theorems from logical formulas and explicit facts. This section considers the deduction paradigm for complex values. As in deductive databases, there are only certain complex value query programs which may be regarded as reasonable. Since the class of embedded domain independent database query programs is recursively unsolvable, it is desirable to search for recursive subclasses with simple decision procedures. This section also considers this decision problem.

Just as not all calculus queries are reasonable, so not all complex value query programs are reasonable. The set of correct answers to an acceptable query can depend on the language; that is, the answer to a query may not be domain-independent. The following example shows this phenomenon.

**Example 22.** Let $\mathscr{P}$ be the query program:

$$q(a) \leftarrow$$
$$r(y,z) \leftarrow [p(x,z) \wedge z = f(x)] \vee q(y).$$

The set of answers to $\mathscr{P} \cup \{\leftarrow r(y,z)\}$ depends on the interpretation, so $\mathscr{P}$ is not a reasonable query program.

In order to capture the concept of a reasonable recursive program query, the notion of an embedded domain-independent database program is introduced.

Let $C_{\mathscr{P}}$ denote the set of constants appearing in the program $\mathscr{P}$.

**Definition 26.** A database program $\mathscr{P}$ is *embedded domain-independent at level i* if $\text{ans}(\mathscr{P}, A, S_1) = ans(\mathscr{P}, A, S_2)$, for all interpretations $S_1 = (\mathbf{d}_1, \mathbf{F}_1, \mathbf{I})$ and $S_2 = (\mathbf{d}_2, \mathbf{F}_2, \mathbf{I})$ that agree on *atom* $(C_{\mathscr{P}}, \mathbf{I})$ to level *i*, and for all atoms $A$ in $\mathscr{P}$.

Therefore, given a database $\mathbf{I}$ and an interpretation $S$, a program $\mathscr{P}$ is embedded domain-independent if for every atom $A$ in $\mathscr{P}$ and for every interpretation $S'$ which agrees with $S$ on $(C_{\mathscr{P}}, \mathbf{I})$, the set of answers for $A$ is independent of the interpretation $S'$. The decision problem for the class of embedded domain independent programs is now considered. Unfortunately, the class of embedded domain independent programs is recursively unsolvable.

We define the class of ‘em-allowed’ programs and show that every em-allowed program that satisfies certain constraints is embedded domain independent.

**Definition 27.** A rule is *em-allowed* if each variable that appears in the head also appears in the body and the body is em-allowed.

**Definition 28.** A database program $\mathscr{P}$ is *em-allowed* if each clause in $\mathscr{P}$ is an em-allowed formula.

**Example 23.** The following query program is em-allowed.

$$p(a) \leftarrow$$
$$s(x,z) \leftarrow r(x,y) \wedge z \subseteq y \wedge 5 \notin z$$
$$q(x,v) \leftarrow s(x,z) \wedge v = count(z)$$
$$t(x) \leftarrow p(x) \vee q(x,c).$$

We now consider datalog with negation. Adding negation to datalog rules requires defining semantics for negative facts. We use stratified semantics which is defined from a purely computational perspective that involves a syntactic restriction on programs.

**Definition 29.** A *stratification* of a program $\mathscr{P}$ is a sequence of programs $\mathscr{P}_1,\ldots,\mathscr{P}_n$ such that for some mapping $\sigma$ from $idb(\mathscr{P})$ to $[1,\ldots,n]$,

(1) $\{\mathscr{P}_1,\ldots,\mathscr{P}_n\}$ is a partition of $\mathscr{P}$.
(2) For each derived relation $R$, all the rules in $\mathscr{P}$ defining $R$ are in $\mathscr{P}_{\sigma(R)}$ (i.e., in the same program of the partition).
(3) If $R(u) \leftarrow \cdots R'(v)\ldots$ is a rule in $\mathscr{P}$, and $R'$ is an *idb* relation, then $\sigma(R)' \leqslant \sigma(R)$.
(4) If $R(u) \leftarrow \cdots \neg R'(v)\ldots$ is a rule in $\mathscr{P}$, and $R'$ is an *idb* relation, then $\sigma(R)' < \sigma(R)$.

Given a stratification $\mathscr{P}_1,\ldots,\mathscr{P}_n$ of $\mathscr{P}$, each $\mathscr{P}_i$ is called a *stratum* of the stratification. Not every em-allowed database program with stratified negation is embedded domain-independent. The following example exhibits this phenomenon.

**Example 24.** Let $\mathscr{P}$ be an em-allowed stratified $\mathscr{LDL}$ program:

$$q(a) \leftarrow$$
$$q(b) \leftarrow$$
$$r(x,y) \leftarrow r(x,y)$$
$$s(x,\{y\}) \leftarrow r(x,y)$$
$$p(a) \leftarrow \neg q(x) \wedge s(x,z) \wedge x \in z$$
$$t(a) \leftarrow \neg p(a).$$

Suppose $t$ is the target relation (i.e., the query answer relation). Then $\{a\}$ is a set of answers for $t$ if, and only if the domain of the interpretation contains only constants $a$ and $b$. $\square$

We prescribe the following two additional conditions for stratified programs using both negation and functions.

C1. If the rule $p(x) \leftarrow q(y),\ldots,f,\ldots$ is in stratum $\mathscr{P}_{\mathrm{i}}$, where $f$ is a function, and the rules defining $q$ are in some stratum $\mathscr{P}_{\mathrm{j}}$, then $j < i$ if $y$ appears in $bd(f)$, otherwise $j \leqslant i$.
C2. If a generic recursive cycle occurs in stratum $\mathscr{P}_i$, i.e., $p_2(x) \leftarrow p_1(x) \wedge \cdots;\ p_3(x) \leftarrow p_2(x) \wedge \cdots; \cdots p_1(x) \leftarrow p_k(x) \wedge \cdots$ are in stratum $\mathscr{P}_{\mathrm{i}}$, then this stratum must include some predicate $q$ such that the variable $x$ appears in $q$, $q \in \mathscr{P}_{\mathrm{j}}$ and $j < i$.

**Theorem 9.** *Every query expressed in em-allowed stratified Datalog$^{cv}$ satisfying constraints C1 and C2 is embedded domain independent.*

**Proof.** A query is expressible in Datalog$^{cv}$ with stratified negation if and only if it is expressible in CALC$^{cv}$ [1]. As each rule in the program is em-allowed, each variable is range restricted. For each stratum, the rules defining a predicate can be expressed as an *em-allowed* CALC$^{cv}$ formula. Constraints C1 and C2 guarantee that any function will only produce finitely many new values. Therefore the program is embedded domain-independent. □

**Theorem 10.** *Let $\mathscr{P}$ be an em-allowed program, $S$ and $S'$ two interpretations and $Q$ a query $\leftarrow W$. If $\mathscr{P}$ is stratified and satisfies constraints $C1$ and $C2$, and $W$ is domain independent, then $ans(\mathscr{P}, W, S) = ans(\mathscr{P}, W, S')$.*

**Proof.** This follows from the result of Theorem 9 and the definition of an embedded domain independent formula. □

We consider the complex value query languages with embedded-allowed versions. For each language $L$, we denote by $em - L$ the set of em-allowed queries in $L$. We now have the following:

**Theorem 11.** *em-CALC$^{cv}$, stratified em-(Datalog$^{cv}$) and em-COL are equivalent.*

**Proof.** It is well known that a query is expressible in Datalog$^{cv}$ with stratified negation if and only if it is expressible in CALC$^{cv}$ [3]. *COL* is equivalent to *em*-CALC$^{cv}$ for complex objects [1,2]. It implies that their embedded allowed versions also have equivalent expressive power. □

Note that *em*-CALC$^{cv}$ $+ \mu(\mu^+)$ is not equivalent to the above languages since there exist queries in *em*-CALC$^{cv}$ $+ \mu(\mu^+)$ which are not em-domain independent whereas all queries in the above three languages are em-domain independent.

## 8. Em-allowed, safe range and domain independence

This section discusses the relationship between em-allowed, domain independence and safe range in the various calculus-based query languages we have described.

Let $\mathscr{P}$ be a query program, $W$ a query formula and $S$ an interpretation. Then query program $\mathscr{P}$ is *finite* if the query answer $ans(\mathscr{P}, W, S)$ is finite. For CALC$^{cv}$ queries, em-allowed implies embedded domain independence, *ef*-domain independence and the answer is a finite set. It is *not* equivalent to finitely computable query class, as the query $\forall y p(x,y) \wedge z = f(x)$ is finitely computable (under active domain semantics) but not em-allowed. Clearly, the class of embedded domain independent queries is larger than that of em-allowed queries. For example, $F(y,z) = \exists x[p(x,y) \vee q(y)] \wedge z = f(y)$ is embedded domain independent but not em-allowed. Although *not* every domain independent query is em-allowed, the embedded domain independent calculus and em-allowed calculus are equivalent. That is, for every embedded domain independent formula there exists an equivalent em-allowed formula.

For stratified Datalog$^{cv}$ queries, em-allowed does *not* imply embedded domain independence. Em-allowed does not guarantee finite answer as the query $\{p(0) \leftarrow, p(n) \leftarrow p(m) \wedge succ(m) = n\}$ is em-allowed but not finite, where *succ* is a function which maps each number to its succession. However, em-allowed does imply a weaker form of finiteness. A database instance **I** can be regarded as a set of ground atomic formulas for the base predicates of a Datalog$^{cv}$ program $P$. Let $T_P^k$ be the set of facts about derived predicates in $P$ that can be deduced from **I** by at most $k$ applications of the rules in $P$. A Datalog$^{cv}$ program is *weakly finite* if $T_P^k(I)$ is finite for all $k \geqslant 0$ and all databases **I**. Then em-allowed implies weak finiteness. However, em-allowed is *not* equivalent to weakly finite, as the query $\{p(0) \leftarrow m < 0 \wedge m > 0\}$ is weakly finite but not em-allowed.

For fix-point queries, CALC$^{cv}$ $+ \mu(\mu^+)$, em-allowed implies finite but it does not imply embedded domain-independence. By Theorem 8, embedded allowed formulas define *ef*-domain independent queries.

## 9. Conclusion and further research

In this paper we have presented a theory of database queries on the complex value data model.

We have defined a class of reasonable queries and shown that all calculus queries in this class were domain independent. We have also developed an algorithm for translating embedded allowed queries into equivalent algebraic expressions as a basis for evaluating safe queries in all calculus-based query classes. The result of this work would be very useful for query evaluation in object-relational database systems.

Further research on this topic includes the issues of optimization of translated algebraic expressions [20,25], safe queries in the web scenario, and the open problem of whether a broader subclass of domain independent formulas can be recognized efficiently.

## Acknowledgements

This work was supported by a grant (No. 418205) from the Research Grants Council of the Hong Kong Special Administrative Region, China. We would like to thank the anonymous referees for their careful reading of the paper and numerous helpful suggestions.

## Appendix A

We briefly describe the key features of two languages *LDL* and *COL*. The main set construct in *LDL* the grouping construct, which is closely related to the algebraic nest operation. For instance, in the language *LDL*, the rule:

$$S(x, \langle y \rangle) \leftarrow R(x, y)$$

groups in $S$, for each $x$, all the $y$'s related to it in $R$ (i.e., $S$ is the result of the nesting of $R$ on the second coordinate). The grouping construct can be used to simulate negation.

*COL language*

A key feature of the COL language is the use of base and derived data functions. Data functions are used to "name" set of objects: $F(t_1, \ldots, t_n) \ni t_0$ is an atom defining $F$. The term $F(a_1, \ldots, a_n)$ is interpreted as the set of all the objects $a$ such that $F(a_1, \ldots, a_n) \ni a$ hold. A program is a set of clauses of the form: $L \leftarrow L_1, \ldots L_n$, where $L$ is an atom and the $L_i$ are literals. The defined symbol of a clause is either the relation occurring in the head or the function in the left most position (e.g., $F$ in $F(t_1, \ldots, t_n) \ni t_0$). Under some stratification restrictions, the semantics of programs is given by a minimal and justified model that can be computed using a finite sequence of fix-points.

**Example** Consider the following program $\mathscr{P}$:

$$F(x) \ni \langle y, y' \rangle \leftarrow R(x, y, y'),$$
$$S(x, F(x)) \leftarrow R(x, y, y').$$

The predicate $R$ is extensionally defined, whereas the function $F$ and the predicate $S$ are intensionally defined.

## Appendix B

**Lemma 2.** *Given an em-allowed formula $\varphi$, the transformation using rewriting rules $T_1$–$T_7$ yields an equivalent em-allowed simplified formula $\varphi_t$.*

**Proof.** Clearly, both $\varphi$ and $\varphi_t$ satisfy condition (a) of the em-allowed definition. Let $\chi_0$ be a sub-formula that matches a pattern in rewriting rules and let $\chi_t$ be the corresponding transformed sub-formula. Then,

$$bd(\chi_0) = bd(\chi_t); bd(\neg\chi_0) = bd(\neg\chi_t).$$

For the second expression, $bd(\neg\chi_0) = bd(pushnot(\neg\chi_0))$; $bd(\neg\chi_t) = bd(pushnot\ (\neg\chi_t))$. Because $\chi_t$ is the corresponding transformed sub-formula of $\chi_0$, $pushnot\ (\neg\chi_0) = pushnot(\neg\chi_t)$. So $bd(\neg\chi_0) = bd(\neg\chi_t)$. It is trivial that $\chi_t$ also satisfies conditions (b) and (c) of the em-allowed definition. (For example, let $\chi_0 \equiv \neg(t \neq f(x, y))$. $bd(\chi_0) = bd(\neg(t \neq f(x, y))) = bd(pushnot(t \neq f(x, y))) = bd(t = f(x, y)) = bd(\chi_t)$.)

For each sub-formula $x \subseteq \{y \mid \phi(y)\}$ in $\varphi$, Let $\phi_t$ be the simplified formula of $\phi$. As described above, $bd(\phi) = bd(\phi_t)$. The simplified formula of $\varphi$ satisfies condition (d) of the em-allowed definition.

We conclude that the transformation yields an equivalent em-allowed simplified formula. □

**Lemma 3.** *Given an em-allowed formula $\varphi$, the algorithm ENF terminates and yields an equivalent formula in ENF that is em-allowed.*

**Proof.** First we prove that algorithm ENF terminates on all inputs.

(a) By Lemma 7.7 of [13], ENF($\varphi$) terminates if $\varphi$ contains no sub-formula $x \subseteq \{y \mid \phi(y)\}$.
(b) $\varphi$ contains sub-formula $x \subseteq \{y \mid \phi(y)\}$: As ENF($\phi$) also terminates, ENF($\varphi$) terminates.

We now establish that ENF($\varphi$) is em-allowed.

(a) $\varphi$ contains no sub-formula of the form $x \subseteq \{y \mid \phi(y)\}$:Let $\chi_0$ be a sub-formula that at some point in the execution of ENF matches the pattern of original in the transformations $T_8 - T_{12}$ and let $\chi_t$ be the corresponding transformed sub-formula. As in the relational model, either $\chi_0 = pushnot(\chi_t)$ or $\chi_t = pushnot(\chi_o)$, $bd(\chi_0) = bd(\chi_t)$ and $bd(\neg\chi_0) = bd(\neg\chi_t)$. The conditions of the em-allowed property then follow. Applying these transformations preserves the em-allowed property in the context of the complex value model.
(b) $\varphi$ contains a sub-formula of the form $x \subseteq \{y \mid \phi(y)\}$:Let $\chi_t = x \subseteq \{y \mid \mathrm{ENF}(\phi(y))\}$. We prove that all conditions of the em-allowed property are satisfied. By (a), ENF($\phi(y)$) preserves the em-allowed property. $bd(\phi) = bd(ENF(\phi))$. As $free(\phi(y)) = free(\mathrm{ENF}(\phi(y)))$,

$$bd(\chi_0) = [bd(\phi) \cup free(\phi(y)) \to x] = [bd(\mathrm{ENF}(\phi)) \cup free(\mathrm{ENF}(\phi(y))) \to x] = bd(\chi_t).$$

$bd(\neg\chi_0) = bd(\neg\chi_t)$. Conditions (a), (b) and (c) of the em-allowed property are satisfied. Since $\chi_0$ satisfies condition (d) of the em-allowed property, $bd(\phi) \models free(\phi) \to y$. Because

$$bd(\phi) = bd(\mathrm{ENF}(\phi)), bd(\mathrm{ENF}(\phi)) \models free(\mathrm{ENF}(\phi)) \to y.$$

$\chi_t$ satisfies condition (d) of the em-allowed property.

We conclude that ENF($\varphi$) yields an equivalent formula in ENF that is em-allowed. □

**Lemma 4.** *Given an ENF formula $\varphi$ that is em-allowed, ALG$^{cv}$NF($\varphi$) terminates and yields an equivalent formula in Complex Value Algebra Normal Form.*

**Proof.** We first prove that algorithm ALG$^{cv}$NF terminates on all inputs. Given an ENF formula $\varphi$, $N$-em($\varphi$) is the number of non-em-allowed constructive sub-formulas in $\varphi$. We show that each transformation decreases $N$-em($\varphi$).

- T13, T14, T15, or T16 is applied: These transformations decrease $N - em(\varphi)$. See [13].
- T17 is applied:$N$-em($\phi(y) \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$) is clearly less than $N$-em($\phi(y)$). Because the only transformation of SIMPLIFY applicable to $\phi(y) \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$ is T4, we have $N$-em(SIMPLIFY($\phi(y) \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$)) $<$ $N$-em($\phi(y)$). Hence T17 decreases $N$-em($\varphi$).
- T18 is applied: The proof is same as that of T17.

We conclude that ALG$^{cv}$NF terminates on all inputs.

We now show that the em-allowed property is preserved. Transformations T13–T16 preserve the em-allowed property [13]. Hence it suffices to consider the transformations $T17$ and $T18$. We now prove that transformation T17 preserves the em-allowed property. Let

$$\chi_0 = \xi_1 \wedge \cdots \wedge \xi_m \wedge x \subseteq \{y \mid \phi(y)\},$$
$$\chi_t = \xi_1 \wedge \cdots \wedge \xi_m \wedge x \subseteq \{y \mid \phi(y) \wedge \xi_{i1} \wedge \cdots \wedge \xi_{i_k}\}.$$

We verify that all conditions of the em-allowed property are satisfied.

Condition (a):
We prove that $bd(\chi_0) \subseteq bd(\chi_t)$. Let

$$\Sigma_0 = bd(\xi_1) \cup \cdots \cup bd(\xi_m),$$
$$\Gamma_0 = \Sigma_0 \cup bd(x \subseteq \{y \mid \phi(y)\})$$
$$= \Sigma_0 \cup bd(\phi) \cup \{free(\phi) \to x\},$$

$$\begin{aligned}
\Sigma_1 &= bd(\xi_{i1}) \cup \cdots \cup bd(\xi_{i_k}),\\
\Gamma_t &= \Sigma_0 \cup bd(x \subseteq \{y \mid \phi(y) \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k})\}\\
&= \Sigma_0 \cup bd(\phi) \cup bd(\xi_{i1} \wedge \cdots \wedge \xi_{i_k}) \cup \{free(\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}) \to x\}\\
&= \Sigma_0 \cup \Sigma_1 \cup bd(\phi) \cup \{free(\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}) \to x\}\\
&= \Sigma_0 \cup bd(\phi) \cup \{free(\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}) \to x\} \quad (\text{because } \Sigma_1 \subset \Sigma_0).
\end{aligned}$$

Then, $bd(\chi_0) = \Gamma_0$ and $bd(\chi_t) = \Gamma_t$. So, it suffices to show that $\Gamma_0 \subseteq \Gamma_t$.

We prove $(free(\phi(y) \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}) \to x) \models free(\phi(y)) \to x$. $\phi$ is not em-allowed, $\emptyset \nrightarrow free(\phi)$. $\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}$ is em-allowed, so $\emptyset \to free(\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k})$. Therefore it can imply that $free(\xi_{i1} \wedge \cdots \wedge \xi_{i_k}) \to free(\phi)$ must hold. Since $\{free(\xi_{i1} \wedge \cdots \wedge \xi_{i_k})\} \cap \{x\} = \emptyset$, $free(\xi_{i1} \wedge \cdots \wedge \xi_{i_k}) \nrightarrow x$. But $free(\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{i_k}) \to x$. So $free(\phi) \to x$ can be implied.

It is easily seen that: $bd(\neg\chi_0) = \emptyset^{*,\chi_0} \subseteq bd(\neg\chi_t)$. Condition (a) is satisfied.

As $bd(\xi_0) \subseteq bd(\xi_t)$ and $bd(\neg\xi_0) \subseteq bd(\neg\xi_t)$, conditions (b) and (c) are satisfied.

For each sub-formula $x \subseteq \{y \mid \phi(y)\}$, $bd(\phi) \models free(\phi) \to y$. By fd1

$$bd(\phi) \models free(\phi) \to y \Rightarrow free(\phi \wedge \xi_{i_1} \wedge \cdots \wedge \xi_{ik}) \to y.$$

Condition (d) is satisfied.

The proof for transformation T18 is similar to that of T17. We conclude that ALG$^{cv}$NF yields an equivalent em-allowed formula in Complex Value Algebra Normal Form. □

## References

[1] S. Abiteboul, C. Beeri, The power of languages for the manipulation of complex values, The Very Large Data Bases Journal 4 (1995) 727–794.

[2] S. Abiteboul, S. Grumbach, A rule-based language with functions and sets, ACM Transactions on Database Systems 16 (1) (1991) 1–30.

[3] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.

[4] J.M. Almendros-Jimenez, A. Becerra-Teron, A safe relational calculus for functional logic deductive databases, Electronic Notes on Theoretical Computer Science 86 (2003) 168–204.

[5] A. Avron, Constructibility and decidability versus domain independence and absoluteness, Theoretical Computer Science (2008), doi:10.1016/j.tcs.2007.12.008.

[6] A. Avron, Safety signatures for first-order languages and their applications, in: D. Hendricks (Ed.), First-Order Logic Revisited, Logos Verlag, Berlin, 2004, pp. 37–58.

[7] A. Badia, Safety, domain independence and generalized quantification, Data and Knowledge Engineering 38 (2001) 147–172.

[8] C. Beeri, T. Milo, Comparison of functional and predicative query paradigms, Journal of Computer and System Sciences 54 (1997) 3–33.

[9] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, S. Tsur, Sets and negation in a logic database language, in: Proceedings of ACM Symposium on Principles of Database Systems, 1987, pp. 21–37.

[10] V. Breazu-Tannen, P. Buneman, L. Wong, Naturally embedded query languages, in: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, 1992, pp. 140–154.

[11] E.F. Codd, Relational completeness of data base sublanguages, in: R. Rustin (Ed.), Data Base Systems, Prentice-Hall, 1972, pp. 65–98.

[12] R. Demolombe, Syntactical characterization of a subset of domain-independent formulas, Journal of ACM 39 (1) (1992) 71–94.

[13] M. Escobar-Molano, R. Hull, D. Jacobs, Safety and translation of calculus queries with scalar functions, in: Proceedings of ACM Symposium on Principles of Database Systems, 1993, pp. 253–264.

[14] S. Grumbach, V. Vianu, Expressiveness and complexity of restricted languages for complex objects, in: Proceedings of 3rd International Workshop on Database Programming Languages, 1991, pp. 111–122.

[15] S. Grumbach, V. Vianu, Tractable query languages for complex object databases, Journal of Computer and System Sciences 51 (2) (1995) 149–167.

[16] R. Hull, J. Su, Domain independence and the relational calculus, Acta Informatica 31 (1994) 513–524.

[17] G. Jaeschke, H.-J. Schek, Remarks on the algebra on non-first normal form relations, in: ACM Symposium on Principles of Database Systems, 1982, pp. 124–138.

[18] G. Kuper, Logic programming with sets, Journal of Computer and System Sciences 41 (1) (1990) 44–64.

[19] H.-C. Liu, J. Yu, Safe database queries with external functions, in: Proceedings of International Database Engineering and Applications Symposium, Montreal, Canada, 1999, pp. 260–269.

[20] H.-C. Liu, J. Yu, Algebraic equivalences of nested relational operators, Information Systems 30 (3) (2005) 167–204.

[21] J.M. Nicolas, Logic for improving integrity checking in relational data bases, Acta Informatica 18 (1982) 227–253.

[22] R. Ramakrishnan, F. Bancilhon, A. Silberschatz, Safety of recursive horn clauses with infinite relations (extended abstract), in: Proceedings of ACM Symposium on Principles of Database Systems, 1987, pp. 328–339.
[23] D. Suciu, Domain-independent queries on databases with external functions, in: Proceedings of International Conference on Database Theory, Prague, Czech Republic, 1995, pp. 177–190.
[24] D. Suciu, Domain-independent queries on databases with external functions, Theoretical Computer Science 190 (2) (1998) 279–315.
[25] D. Taniar, H.Y. Khaw, H.C. Tjioe, E. Pardede, The use of hints in sql-nested query optimization, Information Sciences 177 (12) (2007) 2493–2521.
[26] R. Topor, Domain independent formulas and databases, Theoretical Computer Science 52 (3) (1987) 281–307.
[27] R. Topor, Safe database queries with arithmetic relations, in: Proceedings of 14th Australian Computer Science Conference, Sydney, Australia, 1991, pp. 1–13.
[28] J. Ullman, Principles of Database and Knowledge-Base Systems, vol. 1, Computer Science Press, 1988.
[29] A. Van Gelder, R. Topor, Safety and translation of relational calculus queries, ACM Transactions on Database Systems 16 (2) (1991) 235–278.