1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

SAFETY AND TRANSLATION OF COMPLEX VALUE CALCULUS QUERIES

Hong-Cheu Liu and Jeffrey X. Yu

Department of Computer Science
The Australian National University, Canberra, ACT 0200, Australia
E-mail: {hcliu, yu}@cs.anu.edu.au

ABSTRACT

The practical integration of user-defined functions in the relational algebra is relatively straightforward. However, it is significantly more difficult to support this in the relational calculus. Since most query languages are calculus-based, translation of such queries with functions into the equivalent algebra queries becomes a serious problem. This paper explores the issue of the semantics of complex value calculus queries in the presence of functions.

The class of domain independent queries is known to be undecidable. We identify two large decidable subclasses of domain-independent formulas with external functions, namely, the *embedded evaluable* and *embedded allowed* formulas. We then define a recursive class of 'embedded allowed' database programs and prove that embedded allowed stratified programs satisfying certain constraints are embedded domain-independent. Finally we develop an algorithm for translating embedded allowed queries into equivalent algebraic expressions as a basis for evaluating safe queries in all calculus-based query classes.

1. INTRODUCTION

Database Management Systems (DBMSs) are widely used to support new applications such as engineering design, image/voice data management and spatial information systems. Complex values (nested relations, complex objects) and functions (system-defined or user-defined) are important for both practical and theoretical purposes. In [4], SQL3 generalizes the relational model into an object model offering abstract data types and therefore allows users to define data types which suit for their applications. Tables may then contain collections of objects. As for supporting the operational behaviours of any user-defined type, query languages are extended in such a way that user-defined functions can be registered in DBMSs such as Informix. For example, the following SQL statement [15]

```
create function vesting (date)
returns float
as external name 'foo'
language C;
```

registers a function called "vesting" on a data type "date". The function returns a float value and is written using the programming language C. The function has been compiled and is kept in the file `foo`. DBMS can then use any user-defined types/functions in the same way as built-in types/functions. Alternatively, programming languages, such as PASCAL/R and Persistent Java, extend imperative languages to incorporate access to a particular database model.

It is well known that some calculus queries cannot be answered sensibly. In the database field, to answer sensibly means that the values of any correct answer lie within the active domain of the query or the input [17]. There are only certain calculus queries (or formulas) which can be regarded as reasonable in this sense. Such queries are called *domain independent* as they yield the same answer no matter what the underlying domain of interpretation. The following are examples of unreasonable query phenomenon. (1) $\{x \mid \neg Movies(\text{"Cries and Whispers"}, \text{"Bergman"}, x\}$ (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 ICS?" is expressed by $\{x \mid \forall y[\neg R(ICS,y) \vee S(x,y)]\}$. (3) $\{x \mid \exists y,z(P(x) \wedge \neg Q(y) \wedge x+y=z\}$. The set of correct answers for each of the above queries depends on the domains of the variables.

The practical integration of user-defined functions in the relational algebra is relatively straightforward. However, it is significantly more difficult to support this in the relational calculus. The main reason is that the semantic of relational algebra queries evaluated on a database instance is independent of the underlying domains while the relational calculus is domain sensitive. Therefore, the semantics of calculus queries in the presence of functions need further investigation.

Most relational query languages such as SQL and QUEL are calculus-based. If the answers of relational calculus queries are possibly infinite sets or even undefined, it will be difficult to specify a well-defined translation procedure in order to translate user queries into the equivalent relational algebra queries based on which the underneath relational database systems can evaluate. In the past, Van Gelder and Topor in [19] identified such problems in SQL and QUEL. Since SQL attempts to incorporate user-defined types and functions, the problems will become more serious.

As domain-independence is undecidable, it is desirable to develop syntactic conditions (called *safety* conditions) that ensure domain independence. The *evaluable* formulas, originally proposed by Demolombe [8] and discussed by Van Gelder and Topor [19] in the context of flat databases, comprise the largest decidable subclass of the domain independent formulas.. The *allowed* formulas proposed by Topor [17] are a strict subclass of the evaluable formulas. Van Gelder and Topor [19] investigate the properties of two such classes and develop algorithms to transform an evaluable formula into an equivalent allowed formula and from there into relational algebra.

Hull and Su study several alternative semantics of the relational calculus and show that they all have the same expressive power [11]. Abiteboul and Beeri [1] define the notion of *bounded-depth domain independence* and show that, with extended interpreted functions and predicates, the algebra, the bounded-depth domain independent calculus, the safe calculus and the datalog-like language have equivalent expressive power. Their paper [1] also provides a translation from safe calculus into the algebra. This translation is based on associating, with each subformula, "range-restriction" for the free variables that are occurring in the formula.

Escobar-Molano, Hull and Jacobs [9] introduce the notion of *embedded domain independence*, generalize the "allowed" criteria to incorporate scalar functions and develop an algorithm for translating these em-allowed queries into the relational algebra. Their translation framework uses *finiteness dependencies* [14] which are analogous to functional dependencies and carry information about how subformulas involving scalar functions can restrict the possible range of variables. Suciu [16] propose a notion of domain independence (called *ef-domain independence*) for queries with external functions. This notion generalizes those of generic and domain independent queries on databases without external functions.

In this paper, we extend the above notions for the complex value data model and explore the issue of the incorporation of external functions into query languages. Our work presented here can be viewed primarily as an extension of work in [9, 19].

The main contributions of this paper are as follows: (1) We generalize the criteria called *evaluable* [8, 19] to incorporate external functions in the context of the complex value model. Two large decidable subclasses of the extended embedded domain independent formulas are identified, namely, embedded evaluable and embedded allowed calculus formulas. (2) We introduce the em-allowed complex value databases, and show that the class of em-allowed stratified database programs satisfying certain constraints is embedded domain-independent and investigate its properties. And (3) we develop an algorithm for translating embedded allowed formulas into complex value algebra expressions. This algorithm can be viewed as a generalization and extension of those presented in [9, 19]. It adopts the notion of finiteness dependency. This approach is different from that of [1] in that we can apply a heuristic method presented in [9] to simplify the computation involving finite dependencies required to implement the translation.

2. PRELIMINARIES

In this section we briefly review some well known concepts of the complex value data model and query languages, and establish some basic terminology. We assume familiarity with the basic notions of relational database theory.

We generally assume that there is only one sort of domain element rather than many (e.g. integer, float, string) since the nature of the elements is irrelevant to this paper. We let **dom** denote a countably infinite set of uninterpreted constants. We focus on a fixed finite set $\mathcal{F}$ of functions and a fixed finite set **Rel** of relation names. Functions are associated with signatures.

The complex value data model allows the application of two basic constructors, *tuple* and *set* constructors recursively. The abstract syntax of sorts of this data model is given by $\tau = \mathbf{dom} \mid < B_1 : \tau, ..., B_k : \tau > \mid \{\tau\}$; where $k \geq 0$ and $B_1, ..., B_k$ are distinct attributes [2].

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

We define $dom(\tau, D)$, for some type τ and a given subset D of **dom** to be: (1) $dom(\mathbf{dom}, D) = D$, (2) If τ is a type, $\{\tau\}$ is a set type with domain: $dom(\{\tau\}, D) = \mathcal{P}_{fin}(dom(\tau, D))$ and (3) If $\tau_1, ...\tau_n$ are types, $< \tau_1, ...\tau_n >$ is a tuple type with domain: $dom(< \tau_1, ...\tau_n >, D) = \{[a_1, ..., a_n] \mid a_i \in dom(\tau_i, D)\}$.

As in the case of the relational data model, query languages have been developed so far from three paradigms, namely: algebraic, logic, and deductive.

The complex value algebra

The complex value algebra is a many-sorted algebra, denoted by ALG^{CV}. The algebra is a functional language based on a family of core operators and a query is an expression in this language to be evaluated in the given databases.

The core operators are the basic set operators, selection, projection, tup_create, tup_destroy, set_create, set_destroy and powerset. A detailed description of these operators can be found in [2].

The complex value calculus

Calculus formulas are constructed from *atoms* of the form

$$R(\tau_1, ..., \tau_n),\ \tau = \tau',\ \tau \in \tau' \text{ or } \tau \subseteq \tau',$$

where $R \in$ **Rel** and the $\tau_i s$, τ and τ' are terms. Formulas are constructed from atomic formulas using the standard

connectives and quantifiers: $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$. We denote the complex value calculus by $CALC^{cv}$.

Example The second component of tuples of relation R:$\{< A, B >\}$ is replaced by its count number:

$$\{x \mid \exists y(R(y) \wedge x.A = y.A \wedge x.B = count(y.B))\}$$

If $\mathbf{d} \subseteq \mathbf{dom}$ then a pre-interpretation is a pair (**d**, **F**). An interpretation is (**d**, **F**, **I**), where $\mathbf{F} \subset \mathcal{F}$, $\mathbf{F} = (f_1, \ldots f_l)$, $f_i s$ are functions; **I** is a database instance.

If σ is a valuation over free(φ), then the notion of the interpretation (**d**, **F**, **I**) satisfying φ under σ, denoted by $\mathbf{I} \models_{(\mathbf{d}, \mathbf{F})} \varphi[\sigma]$, is defined in the usual manner.

A query q is an expression $\{x_1, \ldots, x_n \mid \varphi(x_1, \ldots, x_n)\}$. Let the types of $x_1, \ldots, x_n$ be $\Gamma_1, \ldots, \Gamma_n$. The notion of the answer q(**I**) to the query q on the instance **I** in the pre-interpretation (**d**, **F**) is defined by:

$$q_{(\mathbf{d},\mathbf{F})}(\mathbf{I}) = \{[(v_1, \ldots, v_n] \mid v_i \text{ is of type } \Gamma_i, \mathbf{I} \models_{(\mathbf{d},\mathbf{F})} \varphi(v_1, \ldots, v_n)\}$$

The *active domain* of a database instance **I**, denoted *adom*(**I**), is the set of all constants occurring in **I**. This is defined analogously for formulas φ and queries q. In addition, we use $adom(q, \mathbf{I})$ as an abbreviation for $adom(q)$ $\cup$ $adom(\mathbf{I})$.

Rule-based languages for complex values

Query languages based on the deduction paradigm are extensions of Datalog to incorporate complex values. Those languages are based on the calculus and do not increase the expressive power of the 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 $CALC^{cv}$. A major difference between the various proposals of logic programming with a set construct lies in their approach to nesting: grouping in $\mathcal{LDL}$ [6], data functions in COL [3], and a form of universal quantification in [12]. We briefly review the concept of Datalog for complex values and queries.

Definition A *database clause* (rule) is an expression of the form $p(t) \leftarrow L_1, \ldots, L_n$, where the head p is a derived predicate, and each L_i of the body is a literal. A program $\mathcal{P}$ is a finite set of rules.

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

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

3. DOMAIN INDEPENDENCE

We begin this section with the standard definition of a domain-independent formula. A calculus formula q (with no embedded functions) is *domain independent* if $q_{\mathbf{d}}(\mathbf{I}) = q_{\mathbf{d}'}(\mathbf{I})$ for each input instance **I**, and each pair $\mathbf{d}, \mathbf{d}' \subseteq \mathbf{dom}$. The value of q on **I** can be determined by evaluating q using the finite set $adom(q, \mathbf{I})$ as the underlying domain.

The notion of "embedded domain independence" was proposed to generalize "domain independence" to incorporate functions [9]. 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) can be applied [1]. The answer to q on an input instance **I** depends on the *closure* of $adom(q, \mathbf{I})$. We review this notion for complex objects as follows.

Given a database instance **I** and a query q, let C_q be a set of constants that appear in q. Following [1], we define $term^n(DB)$, for some database DB with interpretation (**d**, **F**, **I**) by: (1) $term^0(DB) \stackrel{\text{def}}{=} atom(\mathbf{I}, C_q)$ (2) $term^{n+1}(DB) \stackrel{\text{def}}{=} term^n(DB) \cup \{atom(f_i(\dot{x})) \mid f \in \mathbf{F}; x \in dom(\tau_i, term^n(DB)), i = 1, \ldots, l\}$, where $atom(\mathbf{I}, C_q)$ are all atomic values appear in the instance **I** and C_q. 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 $atom(\mathbf{I}, C_q)$ to level n if (1) $term^{n+1}(DB_1) = term^{n+1}(DB_2)$ and (2) $\forall x \in dom(\tau_j, term^n(DB))$, $f_i \in \mathbf{F}_1$, $f_i' \in \mathbf{F}_2$, $f_i(x) = f_i'(x)$, i.e., f_i and f_i' agree on any input whose atomic values are in $term^n(DB)$.

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. 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 [16]. 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 $\xi : \mathbf{d}_1 \rightarrow \mathbf{d}_2$ such that (1) 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 (2) for any $x \in dom(\tau_i, \mathbf{d}_1)$, if $f_j'(\xi(x))$ is defined then so is $\xi(f_j(x))$ and $f_j'(\xi(x)) = \xi(f_j(x))$, where $f_j \in \mathbf{F}_1$, $f_j' \in \mathbf{F}_2$. Let us write $e_1 \sqsubseteq e_2$, whenever expression e_1 is undefined or $e_1 = e_2$. A query is a partial function q mapping any database DB with interpretation (**d**, **F**, **I**) to $q_{\mathbf{d},\mathbf{F}}(\mathbf{I}) \in dom(\{\tau\}, \mathbf{d})$, τ is some type for query result. q is *external-function domain independent (ef-domain independent)* iff for every morphism $\xi : S_1 \rightarrow S_2$, $q_{\mathbf{d}_2,\mathbf{F}_2}(\mathbf{I}_2) \sqsubseteq q_{\mathbf{d}_1,\mathbf{F}_1}(\mathbf{I}_1)$.

4. EMBEDDED EVALUABLE AND EMBEDDED ALLOWED FORMULAS

As (embedded) domain-independence is undecidable, it is desirable to identify large decidable subclasses of embedded domain-independent formulas. We propose two such decidable subclasses: embedded evaluable and embedded allowed formulas. To define embedded evaluable we need to define certain relation between variables and (sub)formulas. It is called *constrained* [19]. We propose a procedure to define such a set of constrained variables.

A key element in the notions of evaluable and allowed formulas is the definition of the *bd* function which associates finiteness dependencies with complex value formulas. First we review finiteness dependency (FinD)[9].

A formula φ satisfies the *finiteness dependency* $X \rightarrow Y$, denoted $\varphi \models X \rightarrow Y$, if for each interpretation (**dom**, **F**, **I**) and each $i \geq 0$ there is some $j \geq 0$ such that $\sigma[Y] \subseteq term^{i+j}(\mathbf{I}, C_\varphi)$ whenever σ is a variable assignment for $x_1,...,x_n$ satisfying $\sigma[X] \subseteq term^i(\mathbf{I}, C_\varphi)$ and $\mathbf{I} \models_{\mathbf{d},\mathbf{F}} \varphi[\sigma]$. If Γ is a set of FinDs over a variable set V then the *closure* of Γ over V is

$$\Gamma^{*,V} = \{X \rightarrow Y \mid XY \subseteq V \text{ and } \Gamma \vdash X \rightarrow Y\}$$

For a formula φ, $\Gamma^{*,\varphi}$ is a shorthand for $\Gamma^{*,free(\varphi)}$.

Example Let the sort of relation R be $\{< t : \mathbf{dom}, x : \{\mathbf{dom}\} >\}$. Letting $\varphi \equiv R(t,x) \wedge t \in x \wedge \neg Q(t) \wedge f(t) = y$, it can be shown that $\varphi \models \emptyset \rightarrow tx$, $\varphi \models x \rightarrow t$, $\varphi \models t \rightarrow y$, $\varphi \models x \rightarrow y$, and $\varphi \models \emptyset \rightarrow txy$.

We present the overall definition of *bd*, and then define the notion of constrained relation.

Definition Given a formula φ, $bd(\varphi)$ returns the set of FinDs, as in Figure 1.

In Figure 1, the formulas 1-11 and their associated function *bd* were presented in [9]. We add formulas 12-16 for the complex value model. The operator $\otimes$ is defined as follows: Given the sets $\Gamma_1,...,\Gamma_n$, of FinDs, $\Gamma_1 \otimes ... \otimes \Gamma_n = \{X_1...X_n \rightarrow Y \in \Gamma_i \text{ for } i \in [1,...,n]\}$ [9]

We now define the notions of constrained relation and em-evaluable formulas. To give these definitions, we define the set of *constrained variables* of a formula using the following procedure.

procedure *constrained-variables(ct)*
input: a calculus formula φ
output: a subset of the free variables of φ
begin
(*pred* is a predicate in $\{\in, \subseteq\}$)
(In each of the cases following, X denotes a set of variables that are members of $\vec{\tau}$)
(Y denotes a set of unconstrained variables generated during the process)

$Y := \emptyset$
case φ **of**

$R(\vec{\tau})$	$: ct(\varphi) := X;$
$\xi \wedge \neg R(\vec{\tau})$	$: ct(\varphi) := (ct(\xi) - X) \cup \{z \mid bd(\xi) \models \emptyset \rightarrow z\};$ $Y := Y \cup X - \{z \mid bd(\xi) \models \emptyset \rightarrow z\}$
$\xi \vee \neg R(\vec{\tau})$	$: ct(\varphi) := (ct(\xi) - X); Y := Y \cup X$
$\neg\xi$	$: ct(\varphi) := ct(pushnot(\neg\xi))$, for ξ not of the form $R(\vec{\tau})$
$f(\vec{\tau}) = \tau$	: If τ is a variable and $X \subseteq \{z \mid bd(\varphi) \models \emptyset \rightarrow z\}$, $ct(\varphi) := \tau$
$\tau \; pred \; \tau'$	: If τ is a variable and $Z \subseteq \{z \mid bd(\varphi) \models$ $: \emptyset \rightarrow z\}$, $ct(\varphi) := \tau$ where Z = set of variables that are members of τ'
$\xi_1 \vee \xi_2$	$: ct(\varphi) := ct(\xi_1) \cup ct(\xi_2) - Y$; where ξ_1 and ξ_2 are not of the form $\neg R(\vec{\tau})$
$\xi_1 \wedge \xi_2$	$: ct(\varphi) := \{z \mid bd(\xi_1) \models \emptyset \rightarrow z\} \cup$ $\{z \mid bd(\xi_2) \models \emptyset \rightarrow z\} \cup$ $(ct(\xi_1) \cup ct(\xi_2) - Y)$, where ξ_1 and ξ_2 are not of the form $\neg R(\vec{\tau})$
$\exists x\xi$	$: ct(\varphi) := ct(\xi) - \{x\}$
$\forall x\xi$	$: ct(\varphi) := ct(\xi) - \{x\}$

end

We say that a variable x is *constrained* in a formula φ if $x \in ct(\varphi)$.

Consider a fixed database DB and variable x of type τ_i. Intuitively, the fact that x is constrained in a formula φ tell us that if $\varphi(x,\vec{y})$ is true, then either (1) $x \in dom(\tau_i, term^n(DB))$ or (2) $\varphi(x,\vec{y})$ is true for *all* values of x, i.e., $x \in dom(\tau_i, \mathbf{dom})$.

Example Consider the following formula

$$\varphi \equiv (P(x,y) \vee Q(y)) \wedge x \in y \wedge (R(y,u) \vee \neg S(y)).$$

Let $A \equiv (P(x,y) \vee Q(y))$, $B \equiv x \in y$, $C \equiv (R(y,u) \vee \neg S(y))$. By using *bd* functions listed in Figure 1, we get $bd(\varphi) \models \emptyset \rightarrow xy$. Then by using the *constrained-variables* procedure, we get (1) $ct(A) = \{xy\}$, (2) $ct(A \wedge B) = \{z \mid bd(A) \models \emptyset \rightarrow z\} \cup \{z \mid bd(B) \models \emptyset \rightarrow z\} \cup (ct(A) \cup ct(B) - Y) = \{y\} \cup \emptyset \cup (\{x,y\} \cup \{x\} - \emptyset) = \{x,y\}$, and (3) $ct((A \wedge B) \wedge C) = \{z \mid bd(A \wedge B) \models \emptyset \rightarrow z\} \cup \{z \mid bd(C) \models \emptyset \rightarrow z\} \cup (ct(A \wedge B) \cup ct(C) - Y) = \{x,y\} \cup \emptyset \cup (\{x,y\} \cup \{u\} - \{y\}) = \{x,y\} \cup \{x,u\} = \{x,y,u\}$.

Definition A formula φ is em-evaluable if:

(a) $bd(\varphi) \models \emptyset \rightarrow free(\varphi)$;

(b) for each sub-formula $\exists\vec{x}\psi$ of φ, $\vec{x} \subseteq ct(\psi)$;

(c) for each sub-formula $\forall\vec{x}\psi$ of φ, $\vec{x} \subseteq ct(\neg\psi)$.

Definition A formula φ is embedded allowed (em-allowed) if:

(a) $bd(\varphi) \models \emptyset \rightarrow free(\varphi)$;

(b) for each sub-formula $\exists\vec{x}\psi$ of φ, $bd(\psi) \models free(\exists\vec{x}\psi) \rightarrow [\vec{x} \cap free(\psi)]$;

	φ	$bd(\varphi)$
1	$R(\tau_1,...,\tau_n)$	$\{\emptyset \to X\}^{*,\varphi}$ where X = set of variables that are members of $\{\tau_1,...,\tau_n\}$
2	$\neg R(\vec{\tau})$	$\emptyset^{*,\varphi}$
3	$\neg\xi$	$bd(pushnot(\neg\xi))$ for ξ not of the form $R(\vec{\tau})$
4	$f(\tau_1,...,\tau_n)=\tau$	$\emptyset^{*,\varphi}$ if τ is not a variable, or τ is a variable occurring in one of $\tau_1,...,\tau_n$
5	$f(\tau_1,...,\tau_n)=\tau$	$\{X \to \tau\}^{*,\varphi}$ if τ is a variable not occurring in any of $\tau_1,...,\tau_n$, where X = set of variables occurring in $\tau_1,...,\tau_n$
6	$x=y$	$\{x \to y, y \to x\}^{*,\varphi}$
7	$\tau_1 \neq \tau_2$	$\emptyset^{*,\varphi}$
8	$\xi_1 \wedge ... \wedge \xi_n$	$(bd(\xi_1) \cup ... \cup bd(\xi_n))^{*,\varphi}$
9	$\xi_1 \vee ... \vee \xi_n$	$(bd(\xi_1) \otimes ... \otimes bd(\xi_n))^{*,\varphi}$
10	$\exists\vec{x}\xi$	$(bd(\xi)$ - all FinD in which some variable in $\vec{x}$ occurs$)^{*,\varphi}$
11	$\forall\vec{x}\xi$	$(bd(\xi)$ - all FinD in which some variable in $\vec{x}$ occurs$)^{*,\varphi}$
12	$x \subseteq \{y \mid \phi(y)\}$	$\{X \to x\}$ if x is a variable not occurring in ϕ, X = set of variables occurring in ϕ.
13	$\tau \in \tau'$	$\{X \to \tau\}^{*,\varphi}$ if τ is a variable not occurring in τ', where X = set of variables occurring in τ'
14	$\tau \in \tau'$	$\emptyset^{*,\varphi}$ if τ is not a variable or τ is a variable occurring in τ'
15	$\tau \subseteq \tau'$	$\{X \to \tau\}^{*,\varphi}$ if τ is a variable not occurring in τ', where X = set of variables occurring in τ'
16	$\tau \subseteq \tau'$	$\emptyset^{*,\varphi}$ if τ is not a variable or τ is a variable occurring in τ'

Figure 1. The function bd.

(c) for each sub-formula $\forall\vec{x}\psi$ of φ, $bd(\neg\psi) \models free(\forall\vec{x}\psi) \to [\vec{x} \cap free(\psi)]$

Theorem 1 *Every em-allowed formula is em-evaluable.*

Proof We consider the condition (*b*) of em-allowed definition. As for each sub-formula $\exists\vec{x}\psi$, $bd(\varphi) \models free(\exists\vec{x}\psi) \to [\vec{x} \cap free(\psi)]$, ψ must not be only of the form $\neg R(\vec{\tau})$. If $\vec{x}$ occurres in the form $\neg R(\vec{x})$, it must also occurres in some other form of $R(\vec{\tau})$, $f(\vec{\tau}) = \tau$, τ *pred* τ'. Therefore, by constrained-variable procedure we can get $\vec{x} \subseteq ct(\varphi)$, which is satistied the condition (b) of em-evaluable. Similarly, the condition (c) of em-allowed holds, then the condition (c) of em-evaluable holds as well.□

Example The following formula is em-evaluable.

$$\varphi(y,z) = \exists x[(p(x,y) \vee q(y)) \wedge z = f(y)]$$

Let $A = [(p(x,y) \vee q(y)) \wedge z = f(y)]$. We have $bd(\varphi) \models \emptyset \to yz$ and $ct(A) = \{x,y,z\}$, as required for φ to be em-evaluable. $bd(A) \models \emptyset \to yz$, so φ is not em-allowed.

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

$bd(A) = \{\emptyset \to zu, zu \to y, zuy \to x\}^{*,A}$. $bd(\varphi) = \{\emptyset \to y, \emptyset \to x\}^{*,\varphi}$ We have $bd(\varphi) \models \emptyset \to xy$ (*i.e.*, $\emptyset \to free(\varphi)$); $bd(A) \models \emptyset \to zu$, and $\emptyset \to zu \vdash xy \to zu$ (*i.e.*, $free(\exists z\exists uA) \to \{z,u\}$). So $\varphi(x,y)$ is em-allowed.

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* ensure that each variable is range restricted, in the sense that relative to the 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 $x = f(x_1,...,x_k)$ 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 is em-allowed.

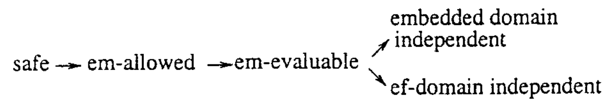
Theorem 2 *Every em-allowed formula is embedded domain independent.*

Proof Since the complex value algebra ALG^{cv} with external functions is embedded domain independent, the proof is demonstrated in the course of translating em-allowed formulas into equivalent algebra queries; see the Section 6. □

Theorem 3 *Every em-allowed formula is ef-domain independent.*

Proof By Theorem 2 and Section 6, 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 [16]. $\mathcal{NRA}$[1] is equivalent to ALG^{cv} without powerset with external functions. By definition, it is easy to check that the powerset query is *ef*-domain independent. Therefore, every em-allowed formula is *ef*-domain independent. □

By adopting the algorithm (**con-to-gen**) described in [19] with minor modification, every em-evaluable formula can be translated into em-allowed formula in the context of the complex value model. We summarize the relationship between these classes as follows:



5. DOMAIN-INDEPENDENT DATABASE PROGRAMS

Just as not all calculus queries are reasonable, so not all complex value database 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. We give two examples.

Example Let $\mathcal{P}$ be the database program:

$$\begin{array}{rcl} q(a) & \leftarrow & \\ r(x,y) & \leftarrow & [p(x,z) \wedge z = f(x)] \vee q(y) \end{array}$$

The set of answers to $\mathcal{P} \cup \{\leftarrow r(x,y)\}$ depends on the interpretation, so $\mathcal{P}$ is not a reasonable database.

Example Let $\mathcal{P}$ be the database program:

$$\begin{array}{rcl} p(\{a\}) & \leftarrow & \\ q(x) & \leftarrow & q(x) \\ r(x) & \leftarrow & q(x) \wedge p(z) \wedge x \notin z \end{array}$$

Let Q be the query $\leftarrow r(x)$. Then, if a is the only constant in the domain of an interpretation, there are no answers for $\mathcal{P} \cup \{Q\}$. But, if the domain contains any constant $b \neq a$, then $\{x/b\}$ is an answer for $\mathcal{P} \cup \{Q\}$. So $\mathcal{P}$ is not a 'reasonable' database program.

[1] See [7] for a detailed description.

We introduce the notion of an embedded domain-independent database program in order to capture the concept of a 'reasonable' recursive program query.

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

Definition A database program $\mathcal{P}$ is *embedded domain-independent at level i* if ans($\mathcal{P}$, A, S_1) = ans($\mathcal{P}$, 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*($C_{\mathcal{P}}$, **I**), and for all atoms A in $\mathcal{P}$.

Therefore, given a database **I** and an interpretation S, a program $\mathcal{P}$ is embedded domain-independent if for every atom A in $\mathcal{P}$ and for every interpretation S' which agrees with S on ($C_{\mathcal{P}}$, **I**), the set of answers for A is independent of the interpretation S'.

We now consider the decision problem for the class of embedded domain independent programs. Unfortunately, the class of embedded domain independent programs is recursively unsolvable. As for deductive databases, it is desirable to search for subclasses with simple decision procedures. We define the class of 'em-allowed' programs and show that every em-allowed program that satisfies certain constraints is embedded domain independent.

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

Example The following database program is em-allowed.

$$\begin{array}{rcl} p(a) & \leftarrow & \\ s(x,z) & \leftarrow & r(x,y) \wedge z \subseteq y \wedge 5 \notin z \\ q(x,v) & \leftarrow & s(x,z) \wedge v = count(z) \\ t(x) & \leftarrow & p(x) \vee q(x,c) \end{array}$$

Not every em-allowed database program with stratified negation is embedded domain-independent. The following example exhibits this phenomenon.

Example Let $\mathcal{P}$ be an em-allowed stratified database program:

$$\begin{array}{rcl} q(a) & \leftarrow & \\ q(b) & \leftarrow & \\ r(x,y) & \leftarrow & r(x,y) \\ s(x,\{y\}) & \leftarrow & r(x,y) \\ p(a) & \leftarrow & \neg q(x) \wedge s(x,z) \wedge x \in z \\ t(a) & \leftarrow & \neg p(a) \end{array}$$

Suppose t is the selected derived 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.

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

C1 If the rule $p(x) \leftarrow q(y), \ldots, f(\ldots), \ldots$ is in stratum $\mathcal{P}_i$ and the rules defining q are in some stratum $\mathcal{P}_j$, then $j < i$ if y appears in $bd(f)$, otherwise $j \leq i$.

C2 If the rule $p(x) \leftarrow p(x) \wedge \ldots$ is in stratum $\mathcal{P}_i$ then it must include some predicate q such that the variable x appears in q, $q \in P_j$ and $j < i$.

We denote the set of em-allowed stratified programs satisfying the above two constraints as $Datalog^{cv}_{em-strat}$.

Theorem 4 *Every query expressed in $Datalog^{cv}_{em-strat}$ is embedded domain-independent.*

Proof (sketch) 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 a *safe* $CALC^{cv}$ formula. Constraints C1 and C2 guarantee that any function will only produce finitely many new values. Therefore the program is embedded domain-independent. □

Theorem 5 *Let $\mathcal{P}$ be an em-allowed program, S and S' two interpretations and Q a query $\leftarrow W$. If $\mathcal{P}$ is stratified and satisfies constraints C1 and C2 (i.e., $\mathcal{P} \in Datalog^{cv}_{em-strat}$), and W is domain independent, then $ans(\mathcal{P}, W, S) = ans(\mathcal{P}, W, S')$.*

Proof This follows from the result of Theorem 4 and the definition of an embedded domain independent formula. □

6. EVALUATION

We describe a non-trivial generalization of the algorithm of [9] for translating embedded allowed complex value formulas into equivalent algebra queries. Our algorithm consists of the following four steps.

1. Replace all sub-formulas of the form $\forall y(y \in x \rightarrow \varphi(y))$ by $x \subseteq \{y \mid \varphi(y)\}$. Next replace any remaining sub-formula of the form $\forall\varphi$ by $\neg\exists\neg\varphi$. Rename the quantified variables if necessary.
2. Transform the em-allowed formula F obtained in step 1 into an equivalent formula F' in Existential Normal Form (ENF).
3. Put the formula F' into an equivalent complex value algebra normal form ψ ($ALG^{cv}NF$).
4. Translate ψ into an equivalent algebra expression E_ψ.

Step 1 is accomplished using the transformations mentioned above. We briefly describe Steps 2 to 4.

Step 2: It is convenient to think of a calculus formula in terms of its parse tree. The leaves of the tree are atomic formulas. There is a sub-formula which corresponds to each internal node labelled by $\vee, \wedge, \neg, \exists x$ or a sub-formula $x \subseteq \{y \mid \varphi(y)\}$. We introduce the concept of Existential Normal Form.

Definition A formula F is in *Existential Normal Form* (ENF) if and only if:

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

In order to satisfy the conditions of ENF, it is necessary to transform in the sub-formulas that violate these conditions. This can be done by using the transformation rules, T_8 - T_{12}, stated in [9].

Example 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 φ' $= (((f(x) = y \wedge x \in z) \vee T(x)) \wedge S(z)$, which is in ENF.

Step 3: We start by defining the concepts of a maximum sub-formula and complex value algebra normal form, then we give a necessary transformation rule not presented in the relational model.

Definition A sub-formula G of a formula F is *maximum* if either G is F; or G's root is $\wedge$; or G is a child node of $\exists, \vee, \neg$ or a range formula $x \subseteq \{y \mid \varphi(y)\}$.

Our aim is to transform a given em-allowed query into an equivalent query, all of whose maximum subqueries can in fact be translated. During this step, the function bd is crucial to decide whether a sub-formula is em-allowed.

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

In addition to the rules T_{13} to T_{16} stated in [9], we need the following rule to transform ENF em-allowed formula to complex value algebra normal form.

Rule T_{17}: $\xi_1 \wedge \ldots \wedge \xi_m \wedge x \subseteq \{y \mid \varphi(y)\} \rightarrow \xi_1 \wedge \ldots \wedge \xi_m \wedge x \subseteq \{y \mid \varphi(y) \wedge \xi_{i_1} \wedge \ldots \wedge \xi_{i_k}\}$ where $\varphi(y)$ is not em-allowed, but $\varphi(y) \wedge \xi_{i_1} \wedge \ldots \wedge \xi_{i_k}$ is em-allowed.

Example Consider the following formula $\exists u \exists s(S(u) \wedge R(s) \wedge x \subseteq \{y \mid (y + s.A) \in u.C\} \wedge (t \in x \wedge \neg Q(t)))$. We apply the above rule to obtain $\exists u \exists s(S(u) \wedge R(s) \wedge x \subseteq \{y \mid$

[2] In every sub-formula $\exists\vec{x}\varphi$, each x_i is actually free in φ and the polyadic operators $\wedge$, $\vee$, $\exists$ are flattened. See [9] for detailed description.

$S(u) \wedge R(s) \wedge y + s.A \in u.C\} \wedge (t \in x \wedge \neg Q(t)))$ which is in $ALG^{cv}NF$.

Step 4: Translating an $ALG^{cv}NF$ formula into an equivalent algebra expression can be performed by applying the following method: conjunctions are translated into joins or Cartesian products, negations into generalised differences (**diff**) [19] , existential quantifiers into projections, inequalities into selections and equalities and arithmetic operations into *appends* which are described below.

Append is a relational operator which is defined in [18]. We shall denote it by Φ. Suppose r is a relation of l-tuples, then $\Phi_{g(i_1,\ldots,i_k)}(r)$ is a set of $l+1$ tuples, $k \leq l$, where g is an arithmetic operator on the components $i_1,\ldots,i_k$. The last component of each tuple is the value of $g(i_1,\ldots,i_k)$.

Example Consider the following formula $\exists z \exists u (R(z) \wedge S(u) \wedge t \in \{y \mid y + z.A \in u.C\} \wedge \neg Q(t))$. Transforming it into $ALG^{cv}NF$ yields $\exists z \exists u (R(z) \wedge S(u) \wedge t \in \{y \mid R(z) \wedge S(u) \wedge y + z.A \in u.C\} \wedge \neg Q(t))$

The equivalent algebra query is obtained using the following program

$E_1 = tup_create_{C'}(set_destroy(tup_destroy\ (\pi_C(S))))$;

$E_2 = \pi_{A'C'}(\sigma_{C' \in C}(S \times E_1))$;

$E_3 = E_2 \times (\pi_A(R))$; $\quad E_4 = \Phi_{g(2,3)}(E_3)$;

$E_5 = nest_{C' Ay \to X}(E_4)$; $\quad E_6 = \pi_{A',X.y}(E_5)$;

$E_7 = replace < [A', X, Z = (rename_{y \to t}(X)\ \mathbf{diff}\ Q)] > (E_6)$

$E_8 = \pi_{X,Z}(E_7)$

where $g(2,3)$ = column 2 − column 3. Note that E_2 is equivalent to $unnest_C(S)$. A detailed description of the complex value algebra including *set_destroy*, *tup_create* and *replace* can be found in [2].

7. CONCLUSION

Translation of relational calculus queries that support both user-defined functions and complex values into the corresponding relational algebra queries is challenging, because the class of domain independent is known to be undecidable even for DBMSs that don't support user-defined functions and complex values. In this paper, we identify two large decidable subclasses of embedded domain-independent formulas, namely, the embedded evaluable and embedded allowed formulas. The question: can a broader subclass of embedded domain independent formulas, be recognized efficiently remains open.

We define a recursive class of "embedded allowed" database programs and prove that embedded allowed stratified programs satisfying certain constraints are embedded domain-independent.

We also investigate the issue of how to implement complex value calculus queries with the incorporation of functions. A rather involved construction has been shown for translating embedded allowed calculus formulas into the algebra. The algorithm presented here is still open to optimization. A formal analysis of the complexity of the complete algorithm also needs detailed study.

REFERENCES

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

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

[3] S. Abiteboul and S. Grumbach. A Rule-Based Language with Functions and Sets. In *ACM Transactions on Database Systems*, **16**(1):1-30, 1991.

[4] D. Beech. Collections of Objects in SQL3. In *Proc. of Int. Conf. on Very Large Data Bases*, pp. 244-255, 1993.

[5] C. Beeri and T. Milo. Comparison of Functional and Predicative Query Paradigms. In *Journal of Computer and System Sciences*, **54**, pp. 3-33, 1997.

[6] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. of ACM Symp. on Principles of Database Systems*, pp. 21-37, 1987.

[7] V. Breazu-Tannen, P. Buneman and L. Wong. Naturally embedded query languages. In *Proc. of 4th Int. Conf. on Database Theory,*, pp. 140 - 154, 1992.

[8] R. Demolombe. Syntactical Characterization of a Subset of Domain-Independent Formulas. In *Journal of ACM*, **39**(1):71-94, 1992.

[9] M. Escobar-Molano, R. Hull and D. Jacobs. Safety and Translation of Calculus Queries with Scalar Functions. In *Proc. of ACM Symp. on Principles of Database Systems*, pp. 253-264, 1993.

[10] S. Grumbach and V. Vianu. Tractable Query Languages for Complex Object Databases. In *Journal of Computer and System Sciences*, **51**(2):149-167, 1995.

[11] R. Hull and J. Su. Domain Independence and the Relational Calculus. *Acta Informatica*, **31**, pp. 513-524, 1994.

[12] G. M. Kuper. Logic Programming with Sets. *Journal of Computer and System Sciences*, **41**, pp. 44- 64, 1990.

[13] H.-C. Liu, Jeffrey X. Yu and C. Chirathmajaree. On Translation of Complex Value Calculus Queries with Arithmetic Operators. In *Proc. of Int. Symp. on Cooperative Database Systems for Advanced Applications*, Japan, pp. 492 - 495, 1996.

[14] R. Ramakrishnan, F. Bancilhon and A. Silberschatz. Safety of recursive horn clauses with infinite relations. In *Proc. of ACM Symp. on Principles of Database Systems*, pp 328 - 339, 1987.

[15] M. Stonebraker. *Object-Relational DBMSs - the next great wave*. Morgan Kaufmann Publisher, Inc. 1996.

[16] Dan Suciu. Domain-Independent Queries on Databases with External Functions. In *Proc. of Int. Conf. on Database Theory,*, pp. 177-190, 1995.

[17] R. Topor. Domain independent formulas and databases. *Theoretical Computer Science*, **52**(3):281-307, 1987.

[18] R. Topor. Safe Database Queries with Arithmetic Relations. In *Proc. of 14th Australian Computer Science Conference*, pp. 1-13, 1991.

[19] A. Van Gelder and R. W. Topor. Safety and Translation of Relational Calculus Queries. In *ACM Transactions on Database Systems*, **16**(2):235-278, 1991.