Data Construction with Recursive Set Expressions *

J. Eder† A. Rudloff‡ F. Matthes‡ J. W. Schmidt‡

Institute for Statistics and Comp. Science†
University of Vienna
Liebiggasse 4
A - 1010 Vienna, Austria

Department of Computer Science‡
University of Hamburg
Schlüterstr. 70
D - 2000 Hamburg 13, FRG

Abstract

In this paper we present a conceptually rather conservative approach to data deduction. Instead of introducing new language constructs we stay within the conventional relational framework while exploiting it further by making better use of current language technology. Applying naming, typing and binding to queries and relations leads to a language that gains expressiveness from its orthogonality rather than from extensiveness.

As a framework for our presentation we use DBPL, a strongly typed database programming language based on the relational calculus and on Modula-2. In DBPL, data construction is expressed declaratively through set-oriented expressions which can be abstracted by parameterization and by naming, thus allowing powerful recursive constructor definitions. In the paper, a model-theoretic constructor semantics is defined in two steps: parameter substitution in *constructor definitions* leads to *constructor instances* which are then evaluated in a second step. The first step can be regarded as logic program generation, the second step as program evaluation.

We show that for the general case no procedure exists that evaluates an arbitrary set of parameterized constructors and is guaranteed to terminate. However, we are able to classify constructors and give an evaluation algorithm which terminates for interesting subclasses. Finally, we solve an example from the literature showing that NP-complete problems can be solved by constructors from a subclass which is decidable. This implies that decidable DBPL constructors are strictly more expressive than stratified Datalog.

1 Introduction

In the setting of a database programming language, queries are considered as a specific class of expressions that compute set-valued results from their contributing operands. In this sense, relational queries are mappings from a database state, represented by a collection of relation variables, into relation values. Compared with, e.g. arithmetic expressions, such query expressions are quite restricted since query languages do not allow

*This work was supported by the Deutscher Akademischer Austauschdienst, DAAD, by the Austrian Fonds zur Förderung der wissenschaftlichen Forschung (contract P6772P) and by the European Commission under ESPRIT Basic Research Action FIDE (contract 3070).

the use of arbitrary computable functions. The main motivation behind such restrictions is the use of queries for abstract and yet efficient access to large and shared data spaces [Cod70]. Therefore, query languages restrict themselves for the most part to operations for constructing (by Cartesian product), compressing (by predicative selection), and slicing (by projection) relational data spaces. However, even for relationally complete query languages as obtained by orthogonal combinations of the above operations, there exist desirable results that cannot be reached by expressions of finite size [AU79].

Thus, many proposals have been offered to extend the expressiveness of query languages, most of which can be discussed in the framework of Datalog [CGT89]. Datalog is a language based on function-free Horn clauses and draws additional expressive power primarily from recursion.

There are, nevertheless, several restrictions to the Datalog family that triggered an ongoing activity regarding "Stratified Datalog" extensions [BNR87], [LNP et al. 88], [Dah87], [KP88].

One severe restriction of Datalog is the lack of polymorphism. In Datalog one cannot define, for example, a generic set of rules for the transitive closure of arbitrary binary relations. Instead, one has to define such rules separately for each relation, leading to programs that are redundant with respect to bijective renaming of predicates. From a software engineering point of view such program redundancy is a major drawback often referred to as "text editor polymorphism".

From a modelling point of view, Datalog is restricted by being based on a "flat data model". Some extensions proposed make use of functions and sets to handle more complex objects. These sets, however, are not typed and have to be equipped by user-provided operations (instead of generic ones).

A third restriction is the limited expressiveness of Datalog, which, although exceeding that of relational complete languages, is still below that of first-order languages.

In this paper we present a conceptually rather conservative approach to data deduction. Instead of experimenting with a variety of language extensions and new computational models, we stay within the conventional relational framework and exploit it further by making full use of current language technology. Here the term language technology refers to mechanisms for naming, typing and binding of queries and relations, together with the design principle of achieving expressiveness by combinability rather than by extensiveness.

One major motivation behind our project is to exploit fully the benefits of DBPL's characteristics such as strong and static typing, type-completeness and orthogonality. Furthermore, we strove to keep under tight control the consequences of advanced data construction on our database system implementation and its operational support (e.g. query optimization, transaction management and overall system architecture).

The paper is organized as follows. In section two we give a short introduction of the relevant part of the database programming language DBPL [SEM88] [MRS89], specifically, the concept of a relation constructor [JLS85]. We de-emphasize the procedural part of the language and concentrate on types and expressions, in particular on expression naming, typing and binding.

In sections three and four we describe formal semantics to the resulting typed and parameterized query calculus. We have to deal with two kinds of recursion: first, the class

of statically defined, non-parameterized, mutually recursive constructors with their standard fixpoint semantics. Second, there are the dynamically created constructor instances based on actual parameters that are supplied at constructor invocation time. In particular, the invocation of a constructor may be parameterized by a term that, recursively, depends on the constructor at hand.

Section five demonstrates by an extended example the increase in expressiveness gained by applying current language technology to the conventional relational data model. The concluding section comments further on this first experience with our approach and on current research.

2 An Introduction to DBPL

We base our informal introduction into the DBPL language on an example from combinatorial circuits. The current section introduces DBPL types by providing the representation of an electronic circuit (see Fig. 1) and it discusses DBPL query expressions by evaluating some circuit data. Section five concentrates on a more elaborate problem introduced by [BI90] and solves it by recursive set expressions.

2.1 Types

DBPL's contributions to typing and persistence can be characterized by the principles of type completeness and orthogonal persistence.

DBPL is type complete in the sense that a user of the language can exploit the entire type space defined by base types (with the exception of data of type *reference*, see [MS89]) and by arbitrary nestings of type constructors including relations. In our example (see Fig. 2) this leads to a definition of CircuitType as a relation type based on GateType records that have relation-valued attributes for outgoing connections. Orthogonal persistence refers to DBPL's ability to make variables of any type long-lived. In our example we exploited this property by including into the persistent Database Definition Module, Topology, not only the relation variable, circuit, but also the base type variable, highestGateIdUsed. DBPL, being based on the systems programming language Modula-2, supports the engineering of large software systems. The module concepts, for example, not only allow programs to be partitioned and their interfaces to be controlled but also separates and hides implementations from definitions. In our circuit example we may want to separate the definition of circuit topology from applications that use topology data to derive state-oriented properties of a circuit (Fig. 3).

2.2 Expressions

Relations can be evaluated by two classes of declarative expressions: by Boolean expressions using logical quantifiers and variables bound to relations, and by access expressions that denote those relation elements which verify some Boolean expression.

Boolean expressions are highly desirable for bulk data evaluation since, on the one hand, they provide the conceptual basis for associative data selection and integrity control while,

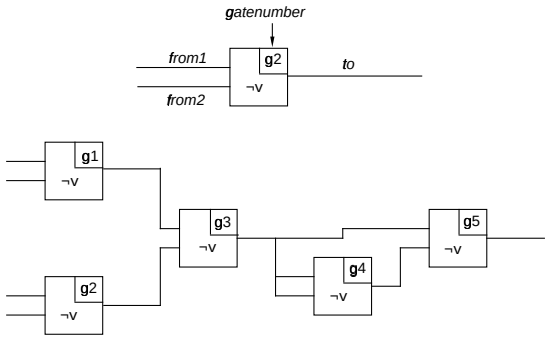

Figure 1: Gates and Circuits

```
DEFINITION MODULE Topology;
TYPE GidType = (g1, g2, g3, g4, g5,  ..., open);
     GateIds = RELATION OF GidType;
     GateType = RECORD id: GidType; from1, from2: GidType;
                       to: GateIds; END;
     CircuitType = RELATION id OF GateType; VAR circuit: CircuitType;
     highestGateIdUsed: GidType;
END Topology.
```

Figure 2: Type Definitions for Circuit Topology

on the other hand, they constitute an abstract interface to the more procedural parts of problem solution in terms of conditionals and iterators of the database programming language. Examples in our topology database are the following expressions that test whether there exists (at least one) gate with an open input

```
SOME c IN circuit ({c.from1, c.from2} = {open, open})
```

or whether gate identifiers are used consistently

```
ALL c IN circuit (c.id ≤ highestGateIdUsed).
```

Access expressions abstract from the details of bulk data access and presentation. The following example of a (selective) access expression,

```
EACH c In circuit: {c.from1, c.from2} = {open, open},
```

denotes those elements of our circuit relation that have open inputs.

The elements denoted by an access expression can be presented to its environment in two different ways. First, we can use access expressions as parameters for relation constructors:

```
DEFINITION MODULE  States; IMPORT Topology;
TYPE GateIn = RECORD id: Topology.GidType; in1, in2: BOOLEAN; END;
     GateOut= RECORD id: Topology. GidType; out: BOOLEAN; END;
     InState = RELATION id OF GateIn;
     OutState = RELATION id OF GateOut;
     InSpace = RELATION OF InState;
END States;
```

Figure 3: Type Definitions for Circuit States

```
CircuitType{EACH c In circuit: {c.from1, c.from2} = {open, open}}.
```

In this way a *relation value* is returned n-elements-at-a-time. Secondly, access expressions can be used also as *iterators* in a loop definition, e.g.

```
FOR EACH c In circuit: {c.from1, c.from2} = {open, open} DO ...
```

thus providing a one-element-at-a-time interface to relational data.

In full generality, DBPL makes use of the expressiveness of a relationally complete, calculus-based query language: the (constructive) access expression

```
cons(r1, ..., rn) OF EACH r1 IN R1, ... EACH rn IN RN: pred(r1, ..., rn),
```

returns a result constructed by some element-constructor, cons, over those elements r1, ..., rn of R1, ..., Rn that verify the selection predicate. Notice that access expressions can form lists provided the contributing expressions construct union-compatible results.

A final example makes use of relation nesting and returns pairs of gate identifiers that are connected by the circuit:

```
{g.id, g'.id} OF EACH g IN circuit, EACH g' IN g.to: TRUE
```

2.3 Constructors

Applying language technology [IBM90] to our relational extensions essentially results in providing appropriate means for the naming, typing and binding of relational types, variables, expressions, etc. In the following we will concentrate on naming and parameterization of access expressions. Similar to functions a *Constructor* associates a name with some body (which is an access expression list) and introduces typed by-value parameters together with a result type. For the subsequent formalization of constructor semantics we restrict ourselves to a single relational parameter and to records as relation elements (ruling out arrays and variants); furthermore, we disallow subrange and enumeration types.

When specifying their semantics it helps to recognize that constructors are used on three different levels.

1. By a **constructor definition** we introduce a mapping from a name to a (possibly parameterized) access expression.

```
CONSTRUCTOR  InitIn:InState;
BEGIN {c.id, FALSE, FALSE} OF
        EACH c IN circuit:{c.from1,c.from2}={open,open}  END InitIn;

CONSTRUCTOR Higher ON (parmH: InState): InState;
BEGIN EACH g IN parmH: SOME g' IN parmH (g.id > g'.id) END Higher;
```

   The first definition associates a name, InitIn, with an access expression that constructs for all gates in the circuit with open input a triple composed of their gate identifier and two times the value FALSE. The second definition introduces a name, Higher, together with a parameterized access expression that selects from an arbitrary relation of type InState those elements that have an identifier higher than that of the minimal element.

2. By a **constructor instance** we mean the result of substituting the formal parameter of a constructor definition by an actual one.

```
 Higher({{g1, FALSE, FALSE}, {g2, FALSE, FALSE}})
```

   for example, identifies the following constructor instance

```
EACH g IN {{g1, FALSE, FALSE}, {g2, FALSE, FALSE}}:
  SOME g' IN {{g1, FALSE, FALSE}, {g2, FALSE, FALSE}} (g.id > g'.id).
```

3. Finally, by a **constructed relation** we mean the extensional relation value gained by mapping a constructor instance to the corresponding relation from the result type value set. In our example, relation construction, i.e.,

```
InState{EACH g IN {{g1, FALSE, FALSE}, {g2, FALSE, FALSE}}:
  SOME g' IN {{g1, FALSE, FALSE}, {g2, FALSE, FALSE}} (g.id > g'.id)}
```

   results in the constructed relation {{g1, FALSE, FALSE}}.

A DBPL constructor has, just like a function, also a type. A constructor type is defined by the type of its parameter and by its result type.

```
TYPE HigherType = CONSTRUCTOR ON (InState): InState;
```

For the purpose of this paper the use of constructor types is restricted to ON-parameters.

2.4 Deductive DBPL and Datalog

The rest of this paper will concentrate on the semantics of constructors which we consider to be the deductive subset of DBPL. After enumerating some of the major differences between our approach and Datalog [CGT90], we will define a declarative semantics and develop an evaluation algorithm for deductive DBPL. Some comparative features of DBPL and Datalog to keep in mind:

- DBPL is based on the relational tuple calculus while Datalog is an extension of the domain calculus.
- DBPL is a strongly typed language while original Datalog is untyped. Typed Datalog versions are a recent development [LNP et al. 88], [YS87], [Frü90].

  Being typed puts DBPL constructors into the framework of many-sorted logic. This strong typing, however, also restricts the universe of DBPL.
- Deductive DBPL is based on two specific functors: the tuple construction and the set constructor. Both are restricted through strong typing, and, therefore, cannot be compared in terms of expressiveness with functions in Horn Languages.
- Through the orthogonal combination of tuple and set functors, DBPL supports relations with relation-valued components (NF^2 relations) nested to any depth. In a DBPL program, all relations can be treated uniformly, be it a base relation or a component relation. In some Datalog extensions complex objects can be represented through functor terms and sets. However, since these sets are not typed, they cannot be dealt with like relations.
- A single DBPL constructor corresponds to the set of all Datalog rules having the same predicate in the head.
- DBPL constructors can be parameterized.
- DBPL constructors use explicit quantification, existential and universal. The order of quantifiers matters. In Datalog there is no explicit quantification and implicit quantification prefixes rules by default.
- Finally, since constructors in DBPL are deeply integrated into a complete database programming language with persistence, transactions and query optimization, they are fully supported by up-to-date database technology.

3 Model-Theoretic Semantics

We define the semantics of DBPL's deductive part declaratively in the form of a mapping from a program to a result relation. A program consists of type definitions, base relations, constructors, and a relation expression based on a constructor instance.

A constructor is considered as a template for constructor instances. If a constructor is not parameterized it is equal to its constructor instance. For parameterized constructors we derive constructor instances by substituting the formal parameter through a base relation,

a constructor instance, or an extensionally defined relation (to represent component relations). A constructor instance can be regarded as a rule (better a set of rules) in terms of logic programming.

The very idea for formulating a model-theoretic semantics is that we have to consider the constructor instances in our universe and not only the ground terms. This means that we have to give two interpretations:

- the set of constructor instances which is determined by the program;
- the set of tuples, which constitutes the result of a relation expression.

To attribute meaning to DBPL programs three steps are required. First, we define the set of constructor instances which are implied by the program. Second, we define the extensions of all constructor instances. Finally, we define the result of the given relation expression. Note, the DBPL program without a relation expression has already an associated meaning, as it stands for a (possibly infinite) set of extensional relations, which are intensionally defined by the DBPL program. A relation expression can be considered as a goal in terms of logic programming or as a query against a base or derived relation in terms of database languages.

In this first definition we make the restriction that no negation and no universal quantification may appear in the constructor definition. These cases will be dealt with in one of the following sections. (They are solved through stratification.)

3.1 DBPL Types

For the purpose of this paper we restrict the DBPL types [MS89] as follows:

1. Basetypes
2. Recordtypes
   If $a_1, \ldots, a_n$ are names and $t_1, \ldots, t_n$ are types, then $t = [a_1 : t_1, \ldots, a_n : t_n]$ is a record-type with the name t.
   With each record type t we associate a functor f_t.
3. Relationtypes
   Let t be a type. r = relation of t defines the relation-type r. With each relation type r we associate a constant $\perp_r$ and a binary functor con_r with the following properties: Let A be a functor-term with con_r as principal functor. For all proper terms t_1, t_2: $con_r(t_1, con_r(t_2, A)) = con_r(t_2, con_r(t_1, A))$, and $con_r(t_1, con_r(t_1, A)) = con_r(t_1, A)$.

3.2 DBPL Extents

In the *DBPL-Universe* we define all possible extents of types. If T is a type of a DBPL program, then there is a corresponding set S_T in the DBPL-Universe such that:

- if T is a base type, then S_T consists of all instances of this base type;

- if $T = [a_1 : T_1, \ldots, a_n : T_n]$ is a record type, then $\forall t_1 \in S_{T1}, \ldots, \forall t_n \in S_{Tn}$: $f_T(t_1, \ldots, t_n)$ is in S_T;
- for all record types R there is a constant $\perp_R$ (the empty relation) in S_R;
- if T is a type and R is of type relation of T, and $r \in S_R$, and $t \in S_T$, then $\exists r' \in S_R$ with $r' = con_R(t, r)$;
- S_T is minimal.

The minimality requirement demands that for each relation only one representative of equal con-terms is contained in S_T.

The *Constructor-Instance-Base CI* consists of all constructor instances which can be derived from the constructors of the program, the base relations, and the relations in the DBPL-Universe.

1. For each constructor C of type T without parameters there is a constructor instance C in CI.
2. If C(X) is a constructor of type T with the formal parameter X of type T', and D is the name of a constructor instance of type T' or the name of a base relation of type T' , then C(D) is a constructor instance of type T, and $C(D) \in CI$. C(D) stands for a constructor instance which is derived by substituting the formal parameter X of the constructor C with the actual parameter D.
3. If C(X) is a constructor of type T with the formal parameter X of type T', and D is a ground term in $S_{T'}$ of the DBPL-Universe, then C(D) $\in CI$.

   C(D) stands for a constructor instance which is derived by substituting the formal parameter X by the actual parameter D, where D is an extensionally defined relation. If D = D' (equality of con_T - terms is different from syntactic equality) then C(D) = C(D').

Note that point 2 is similar to predicate substitution [BI90]. Point 3 is a new approach which can be considered as *extensional substitution*. Note that the name of a constructor instance contains the name of its actual parameter. Through our definition of S_T for relation types T we can use the elements of S_T as names for relations and in particular as names for actual parameters.

The *DBPL-Base* consists of a set B_R for each identifyable relation R of the DBPL program. This set describes all possible tuples of which the extension of the relation can consist.

1. If R is a base relation of type relation of T and $t \in R$, then $t \in B_R$.
2. If C is the name of a constructor instance of type relation of T, and $t \in S_T$, then $t \in B_C$.
3. If R is the name of a type (R = relation of T), then for all $r \in S_R$: for all $t \in r \rightarrow t \in B_R$.
   This third type of relation is extensionally defined. Note the name of such a relation is a function term (i.e. a con_r-term). This kind of relation is required for handling relation valued attributes and for substituting formal parameters with component relations of tuples from other relations.

3.3 DBPL Models

We now consider two different kinds of substitutions: substitution of parameters in constructors yielding constructor instances, and the substitution of (range-restricted) variables in constructor instances with values (tuples).

A set I is a *subbase* of the DBPL-Base of a program P, iff for all sets B_R of the DBPL-Base there is a set B_R^I which is a subset of B_R. So all subbases have the same number of sets (with the same subscripts) as the DBPL-Base.

An *interpretation* I is a subbase of the DBPL-Base.

A model has to be consistent with base relations and extensionally defined relations (i.e. relations whose names are relation constants, i.e. con - terms). An interpretation I is consistent for extensional relations, iff for all tuples t in the name of the relation r, $t \in B_r^I$. It is consistent for base relations, iff for all base relations R: $B_R^I = R$.

A DBPL program is *satisfied* by an interpretation, if all constructor-instances C in CI, i.e. all access expressions in C are satisfied. An access expression, A, is satisfied if for all valid substitutions of variables in A (range-restricted and existentially quantified ones) with values of the associated interpretation sets the result, as defined by the element-constructor, is in the set B_C^I of the interpretation.

A substitution is defined valid, iff

1. for all variables t ranging over relation R, the substituting value is in the set B_R of the interpretation;
2. the predicate (without range restrictions) evaluates to *true*.

Note, that by substituting variables with values, we also substitute all component relations which depend on that variable. Through our definition of DBPL-Base, we can be sure that these relations have their corresponding extension sets in the interpretation.

A consistent interpretation which satisfies a DBPL program is called a *DBPL-Model*.

The *intersection* of two models is the set which contains the intersections of all sets of the two models. Since the intersection of two models is a model, there exists a unique least model for a positive DBPL program. This least model is the meaning of a DBPL program.

Further, we are now in a position to define the value of a relation expression, $RelType\{C\}$, as the set of all t in B_C of the least model of P.

3.4 Relevant Constructor Instances

For defining the extensions of a relation expression (goal) we need only a subset of CI, i.e. the *set of relevant constructor instances* RCI for a goal G, defined by:

1. G $\in$ RCI
2. If H $\in$ RCI and H' appears in H (as range relation) then H' $\in$ RCI.
3. RCI is the minimal set of constructor instances for which 1. and 2. holds.

On this set of relevant constructor instances we define the dependency graph of constructor instances in the usual way [CGT90].

3.5 Negation and Universal Quantification

Negation in front of evaluable predicates (after a quantified range expression) causes no problems with our semantics. Such a use of negation is of kind $\exists X notp(X)$ while negated predicates in Datalog are of kind $not \exists X p(X)$. We can view a $notp(X)$ in the evaluable predicates as just another evaluable predicate. Since all variables are bound in preceeding range expression, the safety of such expressions is guaranteed.

For defining a semantics for programs with negation or universal quantification in the range expressions, we use a stratification approach. First we make the observation that negated existence quantifiers and universal quantifiers do not change the set of relevant constructor instances for a given goal. In the definition of the set of relevant constructor instances we look at the range relations only and not at the quantification. Therefore, we don't have to extend the definition of semantics for the inference of constructor instances.

For the derived set of relevant constructor instances we can derive the dependency graph and extend it with labels for edges <C, C'> if C' is used in C as range relation with universal quantification or negated existence quantification. We define stratification thas found in [CGT90] [Naq89]. Only stratified programs will have a result, others will be regarded as inadmissable programs.

4 Operational Semantics

The evaluation of constructors consists of two parts: deriving the set of relevant constructor instances resulting in a first-order program and evaluating this program, (i.e. deriving the extensions of constructor instances). However, these two parts cannot be accomplished generally in two successive phases, since the set of relevant constructor instances may be infinite (recursive nesting of constructors) or depend on actual parameters being subrelations of extensions of constructor instances.

4.1 Constructor Dependencies

The dependency graph of constructor instances may be infinite due to nesting of constructors. We will show, however, that we can distinguish some patterns which cause this infinity. By analyzing these patterns we are also able to discuss whether we can identify a finite set of constructor instances which yields the same result as the set of relevant constructor instances.

We start by analyzing the relationships between constructors and the structure of infinite sets of constructor instances.

Definition 4.1 A constructor C is called *parameter consuming* if it uses the parameter as range relation.

Definition 4.2 (Adorned Constructor Dependency Graph (ACDG))
The nodes of the *Adorned Constructor Dependency Graph (ACDG)* are names of the constructors of the DBPL program (we will only distinguish between the names of the

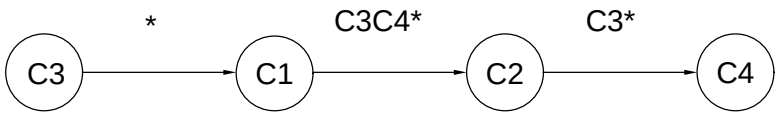

Figure 4: Adorned Constructor Dependency Graph

constructors and the constructors themselves where it is necessary and not obvious from the context). Let C and C' be constructors. The arc <C, C'> is in the graph, if C' is the principal constructor of a range relation in C. If C has a parameter, say X, and this parameter is used in the range relation which has C' as principal constructor then this arc is labeled with an '*' preceeded with all names of constructors which appear in the range relation inside C'. (Note, between C and C' there may be several arcs with different labels.) Furthermore, we mark the parameter consuming nodes.

Since all the nodes of the graph are constructors, and the number of outgoing arcs of a node is less or equal to the number of range relations in that constructor, a finite DBPL program has a finite ACDG.

Definition 4.3 (*-path and label of a *-path) A **-path* is a path in the adorned dependency graph which consists only of arcs marked with *. The *label* of such a path is derived the following way: Let <C, C'> be a path with label L*, and <C',C"> be a path with label K*, then there is a path <C, C"> with label KL* (*concatenation of labels*).

Example:

```
CONSTRUCTOR C1 ON (X): ...;  BEGIN EACH t IN {C2(C3(C4(X)))}:... END C1;

CONSTRUCTOR C2 ON (X): ...; BEGIN EACH t IN {C4(C3(X))}: ... END C2;

CONSTRUCTOR C3 ON (X): ...; BEGIN EACH t IN {C1(X)}: ... END C3;

CONSTRUCTOR C4(X): ...; BEGIN EACH t IN X: ... END C4;
```

The graph in Fig. 4 shows clearly where the infinity of the set of relevant constructor instances comes from.

A parameter may be used in three different ways:

1. as a range relation;
2. as a parameter for a constructor (i.e. instantiating a constructor with the parameter);

3. as a parameter for a constructor which is used as parameter (i.e. instantiating a constructor with a constructor instance which is derived by instantiation of a constructor with the parameter), or further nesting.

It is easy to see that an infinite set of relevant constructor instances is due to the third use of a parameter. Such an appearance of a parameter causes an increase in the level of constructor nesting. Now we can decide whether a program may lead to an infinite number of constructor instances by solely looking at the dependency graph for constructors.

Definition 4.4 A constructor C is *parameter dependent*, if it is parameter consuming or there is a *-path to a parameter C' which is parameter consuming and all constructors of the label of the *-path <C, C'> are parameter dependent.

This definition is motivated by the observation that there may exist constructors with parameters which do not actually make use of the extensions of an actual parameter but use it solely for the purpose of constructor instance naming.

Theorem 4.1 If a constructor C is not parameter dependent, then for all relations R and Q: C(R) = C(Q).

The proof of this theorem is obvious. □

Definition 4.5 A constructor C is parametric dependent on a constructor C' (C :→C') with label L, iff

1. there is a *-path from C to C' with label L, or
2. there exists a parameter consuming constructor C", and there exists a *-path from C to C" with label L_1C'L_2 ,and all constructors in L_1 are parameter dependent and L = L_2, or
3. there exist constructors C', C", C"' being parameter consuming,and there is a *-path from C to C" with label L_1C"'L_2, and all constructors in L_1 are parameter dependent, and C"' is parametric dependent on C' L_3, and L equals the concatenation of L_3 and L_2.

Definition 4.6 A constructor C' is *nesting-dependent* on a constructor C (C+ →C'), iff it is parametric dependent with a label different from *.

Nesting-dependent is stricter than parametric dependent. Nesting-dependent implies that the level of nesting of constructor instances for parameters is increased. A consequence of this is theorem 4.2. If a constructor C is parametric dependent on C', it means that for each instance of C (with actual parameter P) there has to be an instance of C' with P as innermost parameter.

In our example above, C1 is nesting-dependent on C1 with label C3*, on C2 with label C3C4*.

Theorem 4.2 The set of relevant constructor instances is infinite, iff there is a constructor C which is nesting-dependent on itself.

The *fan-out* of a constructor C is the set of all constructors which are needed for evaluating a instance of C independent of its parameter. It is the set of constructors instances which appear in the set of relevant constructor instances of any goal with C as principal constructor.

4.2 Case Analysis

We classify DBPL programs in two dimensions: the nesting of constructors (unnested, layered nested, nested) and the use of subrelations as parameters (no, layered, free). Of all combinations we discuss the following:

1. **Unnested Constructors, No Subrelations**

   If no constructor is nesting-dependent on itself and no component relation is used as a parameter, the set of all relevant constructor instances is finite and can be determined before any tuple of the extensions of these constructor instances is evaluated. Starting from the goal we can construct a complete and finite dependency graph before evaluating a single value (extension) of any constructor instance. In a second phase these constructor instances can then be evaluated (i.e. the extensions of the constructor instances are computed). Compared with logic programming, both phases can be characterized as first-order deductions. In the first phase the result of the deduction process is a first-order program which is evaluated in the second phase.

   All Datalog programs can be mapped into this class of DBPL programs. Furthermore, this class offers in principle enough expressiveness for polymorphism as demanded in the introduction.

2. **Unnested Constructors, Layered Subrelations**
   The constructors are not recursively nested and the constructor instances can be layered in a way, that if a constructor instance X uses a component of a constructor instance Y, then Y is in a lower layer than X (i.e. Y does not depend on X).
   Example:

```
CONSTRUCTOR UpperLevel ON (X : ...): ...;
BEGIN ..., ... OF
   EACH ll IN {LowerLevel}, EACH l IN {UpperLevel(ll.subrel)} :...
   (* ll.subrel is a relation valued attribute *)
END UpperLevel;

CONSTRUCTOR LowerLevel : ...;
BEGIN ..., EACH l IN {LowerLevel(X)}: p(l) END LowerLevel;
```

   For such programs we perform the two phases of the first case in each of the layers. However, to deduct the constructor instances of a layer i, it is necessary to know not only the constructor instances of the previous layers but also their extensions.

3. **Unnested Constructors, Free Subrelations**
   The constructors are not recursively nested, but subrelations are used freely as actual parameters. (There are at least constructor instances X and Y, X has a subrelation of Y as actual parameter, and Y depends on X.) The number of constructor instances is finite, given the number of tuples of each constructor instance is finite.

   In this case the evaluation procedure has to keep track of values and constructor instances in parallel. An exhaustive analysis of such programs is the subject of ongoing research.

4. **Layered Constructors**
   The set of relevant constructor instances is infinite due to recursive nesting of constructors through parameters. Therefore, there exists at least one constructor C which is nesting-dependent on itself with label L_C.

   For case 4 it is required that the constructor-instances can be layered in a way that all instances of such self-dependent constructors C are in a higher layer than all instances of the constructors in the respective label L_C. Here it is be possible that the infinite set of relevant constructor instances can partioned into a finite set of equivalence classes of extensionally equal constructor instances.

   An evaluation algorithm for this kind of program will be discussed in the following section.

5. **Nested Constructors**
   Constructors are nesting dependent on themselves and cannot be layered as in case 4. We cannot give an evaluation algorithm which terminates in general for this class of programs. However, the evaluation may terminate if access expressions which contain nested constructors as range relations need not be further expanded because a different (finitely evaluable) range relation of this access expression is empty.
   Example:

```
CONSTRUCTOR C ON (X):...;
BEGIN ..., EACH t IN {Rest(X)}, EACH x IN {C(C(Rest(X)))}, END C;

CONSTRUCTOR Rest ON (X):...;
BEGIN ..., EACH t IN X : SOME s IN X: t > s.END Rest;
```

   Since Rest is defined in a way that it can be evaluated independently of C, and since the cardinality of Rest(X) is strictly less than the cardinality of X, there exists $i \in N$ with $Rest^i(X) = \emptyset$. All constructor instances with $Rest^j(X), i \leq j$ as range relation in an acess expression can be simplified. However, it seems that a finite evaluation of programs of this class can only be detected through semantical constructor analysis.

4.3 Layered Constructors

In case 4 we consider programs with an infinite set of relevant constructor instances which can be partitioned in a finite set of equivalence classes. This infinity stems from

recursive nesting of constructor instances through parameters. Nevertheless, it may be possible to evaluate such a program. In principle (if no component relations are used as parameters) we could first derive the constructor instances and then their extensions as in case 1. However, since the set of relevant constructor instances is infinite, this is not an adequate evaluation procedure. For case 4 a different evaluation strategy can be applied. It avoids infinity by deriving constructor instances and their extensions in parallel.
Example:

```
CONSTRUCTOR C ON (X: R): R;
BEGIN ..., EACH t IN {C(C1(X))}: ... END C;

CONSTRUCTOR C1 ON (X:R) : R;
BEGIN ... END C1;
```

This example shows that the number of relevant constructor instances for a goal $\{C(A)\}$ is the set of all $C1^i(A)$, and $C(C1^i(A))$, $i \in N$. This set is infinite. However, if it is possible to evaluate $C1^i$ independent of $C1^j, i < j$ (i.e. in the dependency graph there is no path from $C1^i$ to any $C1^j, i < j$), and for some $i < j : C1^i = C1^j$ (extensional equality, i.e. $C1^i$ and $C1^j$ have the same set of tuples), we can then define equivalence classes on these constructor instances and replace each constructor instance by the one that has the smallest exponent (representative of the equivalence class). This way the DBPL program can be evaluated in finite time. Note further that, if the number of possible values of type R in the example above is finite, it is always possible to find a partition into a finite set of equivalence classes.

In a more formal way we can define a DBPL program to be of *class layered constructors*, iff for all constructors C the following holds: C is not nesting dependent on C with a label which includes a constructor that has C in its fan-out. This definition means that if there is a cycle which causes an infinite set of constructor instances, then the constructors accumulated in that cycle do not take part in the cycle. Therefore, it is possible to define a layering that evaluates the parameters first and then the instances of the constructors in the cycle.

For evaluating such programs we avoid the creation of an infinite set of constructor instances by deriving an equivalent finite set of constructor instances. This is always possible because of the following theorem 4.3, which based on the notion of extensional equality.

Definition 4.7 Two constructor instances C and D of type T are extensionally equal, if {C} = {D}, i.e. if they evaluate to the same set of tuples.

This definition implies:

Theorem 4.3 For all types T, all constructors C with parameter of type T, all base relations R and Q of type T: if R = Q holds, then also C(R) = C(Q).

The proof is based on the following arguments:

- if C does not depend on its parameter then the exchange of the actual parameter is merely a renaming of constructor instances;

- if C does depend on its parameter, in each dependent constructor instance where this parameter is used as range relation, exactly the same substitutions are valid, if R and Q are extensionally equal.

□

Theorem 4.3 holds more generally since in a layered evaluation, each relation of layer i can be treated as base relation in layers greater than i.

Theorem 4.4 Consider two constructor instances C and D of type T. If the body of C can be derived by replacing range relations in D by (extensionally) equal relations from lower layers than those of C and D, then {C} = {D} i.e., C and D are extensionally equal.

It is easy to see that by replacing a range relation by another relation which has exactly the same tuples, the valid substitutions remain the same and, therefore, the tuples of the constructor instance. □

To avoid inifinite production of constructor instances we need to close the dependency graph by introducing a new cycle. The following constructors exemplify this technique:

```
CONSTRUCTOR C ON (X: R): R;
BEGIN ..., EACH t IN {C(C1(X))}: ... END C;

CONSTRUCTOR C1 ON (X:R) : R;
BEGIN ... END C1;
```

We will evaluate the goal C(Q). Further we assume that C1(Q) = C1(C1(Q)). Since all instances of C1 are in a lower layer than instances of C, we can treat them like base relations and we will name C1(Q) as A and C1(C1(Q)) as B. Now we will show that replacing the constructor instance C(A) through the constructor instance C'(A) yields the same result.

```
CONSTRUCTOR-INSTANCE C'(A):R;
BEGIN ..., EACH t IN {C'(A)}: ... END C';
```

Since A = B we know that {C(A)} = {C(B)}. It is clear to see that all models for C(A) are also models for C'(A), and vice versa, since all valid substitutions of C(A) and C(B) are equal. This means that we can replace the infinite set of constructor instances induced by C(Q) by a finite one, the dependency graph of which has a cycle.

By straight forward extension this method can also be applied to the more general "power-i-case" where cycles consist of more than one node.

4.3.1 Evaluation Principles

In the following we present and discuss an algorithm which is able to evaluate a goal given by a (possibly recursive) constructor instance (up to now only cases 1 and 4 of section

4.1 are allowed). Our intent in showing this algorithm is to demonstrate the existence of such an evaluation method. We do not claim it efficient.

We begin by considering case 1. Here, the constructor instance dependency graph (CDG) could be built before any evaluation is done. Thus a goal could be evaluated by performing the following steps:

1. Constructing the CDG by expanding the goal.
2. Determining the strong components of the CDG (using some standard algorithm [Baa88]).
3. Checking the admissibility according to the use of negation and universal quantification. Therefore, we have to confirm as certain, if all constructor instances of a strong component use the constructor instances of the same strong component not under negation or universal quantification. (Remember that the partition of the CDG corresponds to a stratification.) If the CDG is not admissible we can stop here because the underlying program has no semantics.
4. Topological sorting of the strong components (using some standard algorithm).
5. Parallel evaluation of the constructor instances of the single strong components [Tar55] (in the topological order of the strong components). For this every known algorithm for evaluating recursive queries could be used, e.g. the differential fixpoint iteration [GKB87].

We now consider case 4. The problem is that in general the CDG is infinite and can only be made finite by recognizing the equivalence classes. To do this, however, we have to evaluate the corresponding constructor instances. That means that we cannot divide the evaluation process in the two succeeding steps which construct the CDG and evaluate all constructor instances.

These two steps have to be merged in order to obtain a terminating evaluation. Here the main problem is the recognition of equivalent nodes in the constructor dependency graph. Equivalence is defined as follows:

Constructor Equivalence:

After each evaluation of a constructor instance C_1C_2 compare its value and type with that of already evaluated instances C_2 (note, that C_2 is a postfix of C_1C_2). If there is one that is equal then C_1C_2 and C_2 are considered equivalent. Building equivalence classes has the following three consequences on the CDG:

1. All edges pointing to C_1C_2 (the C_i are identifiers of constructor instances) can be replaced by edges pointing to C_2 (meaning that no further expansion of C_1C_2 is necessary).
2. If constructor instances of the form $C_3C_1C_2$ and C_3C_2 exist, all edges pointing to $C_3C_1C_2$ can be replaced by edges pointing to C_3C_2.

3. For every newly identified constructor instance C_1C_2 it must be checked whether there exists already a postfix C_p of C_2 with the same value for its constructed relation. If this is the case and if C_1C_p exists, no new constructor instance C_1C_2 will be created and all references to C_1C_2 are redirected to point at C_1C_p. This also means that no further expansion of C_1C_2 is allowed.

Building equivalence classes guarantees termination only if every strong component is detected and evaluated in finite time. Otherwise we may follow an infinite path in the CDG without recognition of postfixes. Under these circumstances it is not possible to use a depth first search algorithm, instead the search has to be breadth first. Before any expansion of constructor instances for the next level the full CDG has to be searched for completed subgraphs. (A subgraph is completed if for every constructor instance in the subgraph the used constructor instances are also contained in the subgraph.) The completed subgraphs can be evaluated following alternative 1 of our case analysis followed by an equivalence class test for the newly evaluated constructor instances. The implementation of consequence 3 of equivalence class formation can be supported by maintaining a set of pairs $< a, b >$ of constructor instances, so that b is a postfix of a and both constructed relations are equal.

5 Example: Solving a NP-Complete Problem with DBPL Constructors

Bonner and Imielinski [BI90] demonstrate the expressive power of predicate substitution by solving a NP-complete problem that cannot be solved by function-free Horn logic. They sketch a Datalog program with substitution that determines for a given combinational circuit whether its output can possibly have the value true. In the example now presented we essentially follow their presentation and discuss a detailed solution of that problem based on DBPL constructors.

Referring back to the DBPL examples of section 2 we provide a relational representation of an arbitrary circuit topology (see Fig. 1) and its relevant states by the two modules, Topology and States (Figs. 2, 3). Based on this representation we solve the problem in three steps:

1. Construction of the input space:

   Based on the circuit representation captured by the actual state of the module Topology (i.e., by the value of its relation circuit, see Fig. 2) we start by giving a circuit input vector, InitIn, that initializes all open gates by setting their inputs to FALSE. The recursive constructor, Inc, then systematically derives all possible inputs vectors by increasing and carrying over the values of its individual components (with the help of the two constructors, Low and Higher).

```
CONSTRUCTOR  InitIn: InState;
BEGIN {c.id, FALSE, FALSE} OF EACH c IN circuit:
           {c.from1, c.from2} = {openIn, openIn} END InitIn;
```

```
CONSTRUCTOR Low ON (parmL: InState): InState;
BEGIN EACH g IN parmL: ALL  g' IN parmL (g.id <= g'.id) END Low;

CONSTRUCTOR Higher ON (parmH: InState): InState;
BEGIN EACH g IN parmH: SOME g' IN parmH (g.id > g'.id) END Higher;

CONSTRUCTOR Inc ON (parmI: InState): InState;
BEGIN {g.id, NOT g.in1, g.in2} OF EACH g  IN {Low(parmI)}: NOT g.in1,
      {g.id, NOT g.in1, NOT g.in2} OF EACH g  IN {Low(parmI)}: g.in1,
      EACH g IN {Higher(parmI)}:
         SOME g' IN {Low(parmI)} (NOT(g'.in1 AND g'.in2)),
      EACH g IN {Inc({Higher(parmI)}:
         SOME g' IN {Low(parmI)} (g'.in1 AND g'.in2) END Inc ;
```

The types required by the above constructors are given in the module States (see Fig. 3, section 2). An instance of type InState, for example, can represent an arbitrary input vector to the open gates. (Note, that the arity of that vector is variable and depends on the circuit at hand.)

Finally, the recursive constructor AllIns covers the entire input space represented by type InSpace which is a set of InState vectors.

```
CONSTRUCTOR AllIns: InSpace;
BEGIN {InitIn}, {Inc(a)} OF EACH a IN {AllIns}: TRUE END AllIns;
```

2. Construction of the output states:

   In a second step, circuit output is deduced for a given input state. The recursive constructor, Deduce, steps through the circuit topology and derives level by level the output states of the individual gates. Note, in our example we restricted the circuit to consist of NOR-gates only.

```
CONSTRUCTOR Deduce ON (parmD: InState) : OutState;
BEGIN {g.id, NOT (g.in1 OR g.in2)} OF EACH g IN parmD: TRUE,
     {g.id, NOT (g'.out OR g''.out)} OF
           EACH g IN circuit, EACH g', g'' IN {Deduce(parmD)}:
               ({g.from1, g.from2} = {g'.id, g''.id}) END Deduce;
```

   The constructor, Deduce, uses (Boolean) expressions for result construction. This goes beyond the DBPL subset formalized above, however, our version of the example can be shown to be just a shorthand that conceptually does not leave the framework of this paper.

3. Evaluation of the output space:

   Finally, we solve the problem by evaluating the circuit input space and the gate output spaces. The following quantified query expression tests for the existence of a positive circuit output. The query evaluates to TRUE if and only if for some vector, a, from the circuit's input space some output, o, can be deduced which has the value TRUE and refers to a gate with an open output line:

```
SOME a IN {AllIns} (SOME o IN {Deduce(a)}
        (o.out AND (circuit[o.id].to = {open}))
```

Note the close interplay between the various language levels of DBPL. Instead of forcing the entire solution into, say, a single set expression and a test for emptiness (which we could have done in DBPL), we make use of DBPL's conceptual richness and exploit quantification, set construction and selection, parameterization, etc.

6 Conclusion

Our approach to data deduction differs substantially from that followed by other projects which either extend Horn logic, Prolog-like languages, or include some extra operator with fixpoint semantics into relational query facilities.

Instead, we applied to DBPL a good language design tradition which recommends that language users be allowed to name and classify (by types, parameters etc.) all constructs considered relevant. There is no doubt that associative query expressions for abstract access to bulk data are the most important single language facility for any data-intensive application.

DBPL's parameterized and possibly recursive constructors prove to be a powerful deductive query language that is shown to exceed stratified Datalog. Since, furthermore, DBPL constructors have the advantage of being deeply integrated into a fully-fledged database programming language and system, we have the possibility of studying very carefully the implications of query facility extension on both applications and implementations.

Practical experience with the use of DBPL in lab classes demonstrates that students quite willingly accept the clear and improved interaction of a database model in the 000form of "typed relational sets" and "declarative set expressions" with other concepts needed and found in procedural languages, like strong typing, fine-grained scoping and dynamic parameterization. However, while set-oriented expressions are readily used for the more standard data retrieval and manipulation tasks, experience also seems to indicate that there is a tendency to fall back to procedural solutions for more complex tasks, e.g. by embedding set expressions into iterators or recursive procedures.

The reason behind such user behavior may originate from the fact that simple query expressions can still be understood by refering to an operational semantics in terms of set construction by loops, conditionals and assignments. However, this view becomes less appropriate for complex queries such as recursive ones that require a more abstract understanding in terms of model-theory and fixpoint semantics.

For designers and teachers it is definitely quite a challenge to reconcile this kind of "abstraction mismatch" inside their languages, a problem hard to overcome without leaving the framework of traditional computer languages (and definitely ruling out the "PL1 + SQL + "*" "-approach). We are convinced that only advanced language technology with higher-order and polymorphic functions, taxonomic typing systems and reflection will form the appropriate basis for next-generation database programming languages [Car89] [MS90] [SFS90].

References

[AU79] A. Aho and J. Ullmann. Universality of data retrieval languages. *ACM Symp. on Principles of Programming Languages*, 1979.

[Baa88] S. Baase. *Computer Algorithms - Introduction to Design and Analysis.* Addison-Wesley Publishing Company, 2nd edition, 1988.

[BNR87] C. Beeri, S. Naqvi and R. Ramakrishnan. Sets and negation in a logic database language (LDL). In: *Proceedings ACM Symposium Principles of Database Systems*, 1987.

[BI90] A.J. Bonner and T. Imielinski. The Reuse and Modification of Rulebases by Predicate Substitution. In: *F. Bancilhon, C. Thanos, and D. Tsichritzis (Eds.): Advances in Database Technology - EDBT'90 (Proceedings)*, Springer-Verlag, 1990.

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

[CGT89] S. Ceri, G. Gottlob and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 1989.

[CGT90] S. Ceri, G. Gottlob and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Databanks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Dah87] E. Dahlhaus. Skolem Normal Forms Concerning the Least Fixpoint. In *E. Börger (Ed.): Computation Theory and Logic* volume 270 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[Frü90] Th. Frühwirth. Types in logic programming. PhD-Thesis, Technical University of Vienna, 1990.

[GKB87] U. Güntzer, W. Kiessling and R. Bayer. On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration. In: *Proceedings 3rd International Conference on Data Engineering*, pp. 120 – 129, Los Angeles, February 1987.

[IBM90] In: A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, August 1990.

[JLS85] M. Jarke, V. Linnemann and J.W. Schmidt. Data Constructors: On the Integration of Rules and Relations. In: *11th Intern. Conference on Very Large Data Bases, Stockholm*, August 1985.

[LNP et al. 88] E. Lambrichts, P. Nees, J. Paradaens, P. Peelman and L. Tanca. MilAnt: An extension of Datalog with complex objects, functions, and negation, 1988.

[KP88] Ph. G. Kolaitis and C.H. Papadimitriou. Why not negation by Fixpoint? In: *Proceedings ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1988.

[MRS89] F. Matthes, A. Rudloff, and J.W. Schmidt. Data- and Rule-Based Database Programming in DBPL. Esprit Project 892 WP/IMP 3.b, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, March 1989.

[MS89] F. Matthes and J.W. Schmidt. The Type System of DBPL. In: *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, pp. 255–260, June 1989.

[MS90] F. Matthes and J. W. Schmidt. Database Application Systems: Types, Kinds and Other Open Invitations. In these proceedings, 1990.

[Naq89] S.A. Naqvi. Stratification as a Design Principle in Logical Query Languages. In: *Proc. of the 2nd Workshop on Database Programming Languages, Salishan Lodge, Oregon*, June 1989.

[Prz88] T. C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In: Jack Minker (Ed.), *Foundations of Deductive Databases*, pp. 193 – 216. Morgan Kaufmann Publishers, 1988.

[SEM88] J.W. Schmidt, H. Eckhardt and F. Matthes. DBPL Report. DBPL-Memo 111-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, West Germany, 1988.

[SFS90] L. Stemple, D. Fegaras, T. Sheard and A. Socorro. Exceeding the Limits of Polymorphism in Database Programming Languages. In: *Advances in Database Technology, EDBT '90*, volume 416 of *Lecture Notes in Computer Science*, pp. 269–285. Springer-Verlag, 1990.

[Tar55] A. Tarski. A Lattice Theoretical Fixpoint Theorem and its Applications. *Pacific J. Mathematics*, 5(2):285–309, June 1955.

[YS87] E. Yardeni and E. Shapiro. A Type System for Logic Programming. In: E. Shapiro (Ed.),*Concurrent Prolog: Collected Papers*, Vol. 2, MIT-Press, 1987.