Constructive second-order proofs in logical databases.

Michel de Rougemont

European Computer-Industry Research Centre.
Arabellastr. 17
8000 Munchen 81, West-Germany

Abstract: The constructive second-order proofs that we study are associated with inductive definitions on classes of finite structures, where each structure represents a database state. To an inductive definition of a predicate P on a class K corresponds a *uniform proof* of P i.e a function which for each structure U defines a proof of P in U

These proofs use computations on sets and the finiteness of the structures in a fundamental way, and hence differ from first-order proofs We show the non-monotonicity of this calculus, and mention the constructivity of some of its intensional properties (time and space complexities)

1. Introduction

A *logical database* is a finite structure U, defined by successive expansions of a database DB with new inductively defined relations, functions and functionals. In this paper, we concentrate on expansions obtained with inductive relational queries only, but the notion of constructible second-order proofs can be generalised to logical databases built with functions and functionals.

With a boolean inductive query Q on a class K of finite structures U we associate a *uniform proof* i.e. a function which to each structure U of a class K defines a proof of Q or $\neg Q$ in U (depending if the interpretation of Q in U, $|Q|^U$, is true or false). Uniform proofs are constructible objects in the following sense: let Q be an inductive query on the class K, and let $K_1=\{U_1=(U,|Q|^U : U \in K\}$. Then to an inductive query Q_1 on the class K_1 there corresponds a uniform proof, built from the uniform proof of Q on K.

Inductive queries exactly capture the notion of polynomial time computability (as a function of the size of the data) [9], and these uniform proofs hence capture the notion of effectiveness (computability in polynomial time). These proofs use computations on sets and the finiteness of the structures in a fundamental way. They are second-order proofs, but only capture the constructible part of second order logic on finite structures. We show the non-monotonicity of the corresponding calculus, and indicate that some of its intensional properties are also constructive.

In the second section, we review the definition of an inductive query, and the differences between logic programming and inductive definability. We give some examples and then describe compilation techniques. In the third section we introduce the notion of a uniform proof, and in the fourth section we study properties of uniform proofs, namely non-monotonicity and intensionality.

2. Inductive definability.

2.1 Notations: We assume that data is given as sets of tuples defining relational sets $\underline{R}_1,\ldots,\underline{R}_k$. $\underline{R}_j(a_1,\ldots,a_j)$ iff $<a_1,\ldots,a_j>$ is a tuple of arity j in the set $\underline{R}_j$, where $a_1,\ldots,a_j \in D$, for a finite set D. A *database* is a relational structure $DB= <D, \underline{R}_1,\ldots,\underline{R}_k>$ and a *database schema* is the class K of all finite relational structures DB of similar signature. A *logical database* (or *knowledge-base*) is a logical expansion of a database, i.e. a structure $U=<D, \underline{R}_1,\ldots,\underline{R}_k, R_1,\ldots,R_l, f_1,\ldots,f_m, 0,1>$, where $R_1,\ldots,R_l$ are relations on D, $f_1,\ldots,f_m$ are functions on D, and 0 and 1 are two distinguished elements of D; in the complexity arguments, $n=|D|$; a *knowledge schema* is a class K of all finite structures U of similar signature. For a knowledge-base U, $\underline{R}_1,\ldots,\underline{R}_k$ are *base relations*, whereas $\underline{R}_1,\ldots,\underline{R}_k, R_1,\ldots,R_l$ are *explicit relations* For a class K, let $L_1(K)$ be classical first-order language with equality, and $L_2(K)$ be the second-order language Let $S_1,\ldots,S_m$ be new relational symbols in the language $L_2(K)$.

Definition:[5] an *inductive system S with parameters* is a sequence of first-order formulas $[F_1,\ldots,F_k]$ in the language $L_1(K) \cup\{S_1,\ldots S_m\}$, where each S_i occurs positively and where the arity of S_i is equal to the number of free variables in F_i.

We write a system as:

$$S_1(x_1\ldots x_{d_1} ; y_1,\ldots,y_p)<= F_1(x_1\ldots x_{d_1} ,S_1\ldots S_k;y_1,\ldots,y_p)$$

$$\ldots \qquad\qquad \ldots$$

$$S_m(x_1\ldots x_{d_m} ; y_1,\ldots,y_p)<= F_m(x_1\ldots x_{d_m} ,S_1\ldots S_k;y_1,\ldots,y_p)$$

The parameter variables are kept constant in a system. If for $i=1,\ldots m$ $d_i \leq d$, we say the system is of *dimension d*. If each F_i is in disjunctive normal form, i.e. $F_i<=>F_{i1}\vee\ldots F_{iq}$, each F_{ij} is the *j-th component* of F_i. The fixed-point semantic associate for each structure U, the simultaneous fixed-point $|S_1|^U,\ldots,|S_m|^U$, and we define the notion of an inductive query.

Definition- |1, 2| **A query** $Q(x_1..x_q)$ is *inductive on a clan* K if there exists a system with parameters such that for all U: $|Q(x_1...x_q)|^U <=> |S_1(x_1...x_{d_1};x_{d_1+1},...,x_q)|^U$.

2.2. Inductive definability and logic programming.

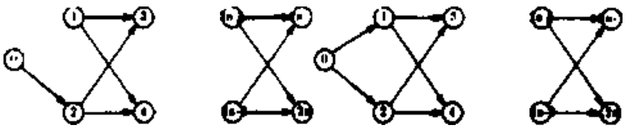
In classical logic programming data and logical definitions are treated as first-order axioms and a query is solved by attempting to find some first-order proof of Q in the theory defined by the axioms If the data changes, the theory changes and so will the proof of Q

In the inductive definability framework, we set a fundamentally different formalism as we distinguish between data and logical definitions The data determines the finite relational structure U of a class K. and the logical definitions are taken as inductive definitions relative to K To a class K we associate a *class of theory,* namely the set of true first-order formulas of U (Th(U)), for each U If the data changes, the structure changes within the same class K, but the the class of theory does not change.

We only solve inductive queries on the class K Given an inductive query Q and a structure U, we solve the query Q and produce a proof of Q or -Q in U To the query Q corresponds a *uniform proof* i.e a function which for each structure U associates a proof of Q or -Q in U. Although we will obtain different proofs for each structure U (database state), they will all correspond to the same uniform proof.

Let us illustrate this fundamental difference with a simple classical example A more detailed analysis is made in |7,. We adopt the Prolog notations: in the inductive framework, "•" replaces the symbol "■-".

Example:Consider two classes of acyclic directed graphs



The class of graphs $G_1(n)$, and $G_2(n)$.

Each finite graph is defined as a structure $G_i(n)=<\{0,1,...,2n\}, \underline{e}>$, i=1 or i=2, where $\underline{e}(a,b)$ <=> there is an edge from a to b. The transitive closure of a graph is a binary relation tc, such that tc(a,b)<=> there is path from a to b. This new relation tc can be defined as a logic program and as an inductive definition:

- **Logic program:(tc on $G_2(n)$)**

```
tc(X , Y) :- e(X,Y)
          ;  e(X,Z), tc(Z,Y).
e(0,2).
e(2,3)
..
..
..
e(2n-2,2n).
```

- **Inductive definition:**

```
tc(X ; Y) : e(X,Y )
          ; e(X,Z), tc (Z ; Y).
```

The inductive definition defines tc(X,Y) on the class of finite graphs (in particular on the classes $G_1(n)$ and $G_2(n)$), where Y is a parameter. in the logic program, tc is defined by 2 rules, and the inductive definition of tc has £ *component**. Let us analyse the q u **tc(0,1)** i n both formalisms.

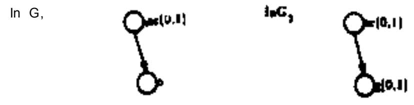
The inductive definition of tc will be compiled into code independent of the data (C-code, then machine code), using call-by-value for the parameter variables, and a call-by-sets for the recursion variables (see next section). In the case of tc(0,l), the computation leads to computing **$\{a/|tc^1(a;1)|^U,...,\{a/|tc^i(a,1)|^U\},...$, i.e unary sets until the closure ordinal λ (less then 2n) is reached. Then we check if $0 \in \{a/|tc^\lambda(a;1)|^U$.**

The query **tc(0,1)** evaluates to false o **$G_1(n)$** n d to true on $G_2(n)$ Let us compare the proofs generated by these two approaches

Logic programs: **On $G_1(n)$, tc(0,1)** evaluates to *no,* using negation by failure |3|, in time $0(2^n)$, as all the paths from 0 are considered **On $G_2(n)$, tc(0,1)** evaluates to *yet,* but the complexity analysis depends on the position of the new atomic clause "**e(0,1).**' If this new clause is inserted before the clause "e(0,2) ", the complexity is constant (one step). If it is inserted after the clause "e(0,2)^n, the complexity will be $0(2^n)$ The worst-case analysis is therefore $0(2^n)$ If we add data (for example ^{n}e(0,l)"), the theory changes, and the proofs change Each proof of tc(0,l) is specific to a given finite graph

Inductive definability: For all constants o and **β**, the uniform proof of **tc(α,β),** is a constructive second-order proof that paraphrases the compilation of the inductive definition It could be represented as *The inductive set* $A=\{a : |tc(a,\beta)|^U\}$ *hat been computed. If* $\alpha \in A$ *then* yes, *otherwise no.* We obtain the same uniform proof independently of the graphs As we compute a unary set, the relativized complexity is O(n) It can be pre-computed at compile-time, and is the basis of the intensional analysis.

On the graphs **$G_1(n)$, tc(0,1)** evaluates to *no,* using the implicit *negation by inductive cloture*[2]. We use the symbol o to represent this negation On the graphs **$G_2(n)$, tc(0,1)** evaluates to *yet* The following labeled trees represent these proofs:



We show that the constructive calculus associated with the inductive definability framework is non-monotonic. The notion of stratification jlOj in logic programming is a step towards inductive definability, and towards a distinction between the structure (the data) and the logical definitions.

[2]The set $A=\{a : |tc(a,1)|^{G_1(n)}\}$ has been computed. $0 \notin A$ therefore no.

2.S. The compilation of inductive queries.

Let K be a class of logical databases. Each explicit relation R is compiled into two C-functions. For simplicity, let us suppose R binary, and assume that a type-definition set has been defined that includes the data-structures for base relations (GRIDS).

```
Rb(x,y)                     set *Rf(x,y)
char *x,*y;                 char *x, *y;
{....}                        {.. }
```

The C-function R_B assumes that x and y are known strings a and b representing elements of the domain, and returns 1 if $|R(a,b)|^U$ and 0 if $|\neg R(a,b)|^U$, whereas the function R_f assumes that at least one of the variables is unknown or free (value nil), and returns a pointer to a set.

2.3.1. Compilation rule: [6]

Let $|S_1, .. ,S_k|$ be a system defining R

- For dimension 0 inductions, generate S_{ib} and S_{if} for $i=1,...,k$, following the cases: Selection, Projection, Join, Intersection, Union, Expansion
- For inductions of positive dimension d, compile S., and $S_{,f}$ as before Pass the parameter by value, and compute the inductive sets $\{a/|S_1{}^i(a;b)|^U\}, ..,\{a/|S_k{}^i(a;b)|^U\}$ until the closure ordinal is reached Project on the recursive variables if known.

This computation rule defines R_b and R_f, and works for existential induction (Horn-Clauses), and Universal inductions. In this last case, we compute a set and check that its cardinality is equal to the cardinality of the finite domain We also compile -^R by generating $\neg R_b(x)$ and act $*\neg R_f(x)$ (using the complement operator on the finite domain D_3.

2.3.2. Witnewet.

Suppose R is defined on a class of structures where S and T are explicit by the component "$R(x,y) \cdot S(x,z), T(z,y)$." The variable *i* is quantified existentially. In order to prove $|R(a,b)|^U$ we have to exhibit an element c in D such that $|S(a,c)|^U$ and $|T(c,b)|^U$ The computation of R_b will call for a selection and then a join-operation. We will compute |S(a,s)| , and check for each element d of that set if $|T(d,b)|^U$. The first d that we find with this property is *the witness* of $|R(a,b)|^U$.

In an induction of positive dimension, a new element in the inductive set S at stage i, uses a witness at stage i-1. These witnesses can be stored together with the inductive sets, at no extra cost.

[3]This coastruction is inefficient at soon as the arity of R is 2, but the intentional analysis will allows us to know in advance that it it inefficient

3. Uniform proofs

Let Q be an inductive query of arity j on a class K where all strings are constant symbols. We associate to the boolean query $Q(a_1,...,a_j)$ on U a labeled proof-tree, with $Q(a_1,...,a_j)$ as label of the root if $|Q(a_1,...,a_j)|^U$ and with $\neg Q(a_1,..,a_j)$ as label of the root if $|\neg Q(a_1,...,a_j)|^U$. The *uniform proof* or *constructible second-order proof* is the function that associates a labeled proof-tree with each structure U.

Definition: A *labeled tree in* U is a finite tree where leaves are labeled with $\underline{R}(b_1,...,b_j)$, $\forall x T(x,b_1,...,b_j)$ or $\circ$ and where nodes are labeled with $<R(b_1,...,b_j), i>$, where $R(b_1,...,b_j)$ is in $L_1(U)$ and i is an integer, such that:

- Each leaf is labeled with $R(e_1,...,e_j)$ for an explicit R, with $\forall x T(x,b_1,...,b_j)$ for an explicit set-relation T or with $\circ$.
- If a node is labeled $<R(b_1,b_2), i>$, and its children are labeled $<R_1(b_1,c)>, i_1>$, $<R_2(c,b_2), i_2>$, then the i-th component of the inductive definition of R is of the form "$R(x,y) : R_1(x,z),R_2(z,y)$" [4]
- If a node is labeled with $<R(b_1,..,b_j), i>$ and its child with $\forall x T(x,b_1, .,b_j)$, then the i-th component of the inductive definition of R is " $R(y) . \forall x\ T(x,y)$".

Definition: An *effective proof of* $Q(a_1,...,a_j)$ *in* U is a labeled tree in U whose root is labeled with $Q(a_1,...,a_j)$ and an *effective proof of* $\neg Q(a_1,...,a_j)$ *in V* is a labeled tree in U whose root is labeled with $\neg Q(a_1,...,a_j)$.

We write $U|\!-_e Q$ and $U|\!-_e \neg Q$ to denote effective proofs of Q and $\neg Q$ in U.

Definition: A *uniform proof* of a query Q on a class K is a function, computable in polynomial time, that associates an effective proof of Q or $\neg Q$ in U,, with each structure U of K,

Theorem 1: If Q is an inductive query on a class K, then for all U in K, $U \models Q$ iff $U|\!-_e Q$, and $U \models \neg Q$ iff $U|\!-_e \neg Q$

Theorem 2: A query Q is computable in polynomial time iff there is a uniform proof of Q.

Theorem 3: If a class K_1 is an inductive expansion of a class K, and if there is a uniform proof of Q on K., then there is a uniform proof of Q on K.

Theorem 1 is implicit with our definition of |- . Theorem 2 comes from the equivalence between inductiveness and being uniformly provable. Theorem 3 is the recursion theorem rephrased in this context.

[4]Similar cases handle the components with explicit negation, and allow nodes to be marked negatively $(<\neg R(b_1,b_2),i>)$.

4. Properties of uniform proofs.

We show that the calculus associated with inductive definitions is *non-monotonic*: the addition of data or logical definitions changes the provability of a query.

4.0.1. Change of data or transaction.

Consider the class of directed graphs augmented with tc, i.e $G=< D, \underline{e}, tc, 0>$ and the following inductive definition: "$P <=> \forall x(x=0 \vee tc(x,0))$."

To P corresponds a uniform proof of P. Let $G_2(n)$ be a graph, as in section 2. $G_2(n) \models P$ and therefore $G_2(n) \vdash_e P$. Suppose we add a new edge $<2n+1, 2n+2>$ to the graph $G_2(n)$ (making a transaction on the data), and define $G_3(n)$ as the new disconnected graph. $G_3(n) \models \neg P$ and therefore $G_3(n) \vdash_e \neg P$

4.0.2. Change of a logical definition.

The non-monotonicity is simply based on the constructive negation, as the following example (adapted fom [4]) shows

- Consider the relational schema Parent(NAME,NAME) and Origin(NAME, ADDRESS), i.e <Joe, Adam> is a tuple of Parent meaning that Adam is the parent of Joe, and <Joe, Antarctic> is a tuple of Origin meaning that Joe lives in the Antarctic. Let DB=<Bird, D_2, Parent, Origin, C> be the finite structure with domain Bird and D_2, base relations Parent and Origin given by the following tables, and an infinite set of constants C. (Bird is the set of individuals {Tweety, Bill, Joe, Adam} and D_2 is the set of individuals addresses {Marineland, Antarctic, Adelie})

Parent

Tweety	Bill
Bill	Joe
Joe	Adam

Origin

Tweety	Marineland
Bob	Antarctic
Adam	Adelie

Let **K** be the class of structures DB, and let us define Anc on K (as tc in section 2)

Anc(x,y) : Parent(x,y)
          ; Parent(x,z), Anc(z,y).

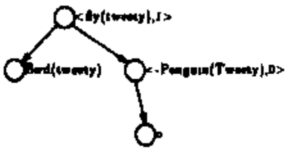
Let $\mathbf{K}_1$ be the class of structures U_1= <Bird, D_2, Parent, Origin, Anc,C>, and let define Penguin on $\mathbf{K}_1$

Penguin(x) : Anc(x,y), Origin(y,"Antarctic").

Let $\mathbf{K}_2$ be the class of structures U_2= <Bird, D_2, Parent, Origin, Anc, Penguin, C> and let define fly:

fly(x) : Bird(x), ¬Penguin(x).

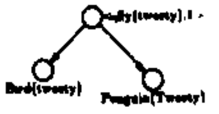
Consider the query fly(tweety) on U_2. The effective proof in DB is:

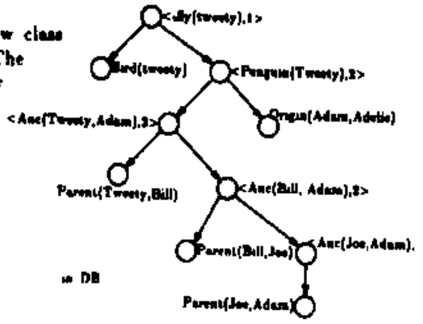


If we find out that Penguins can also originate from Adelie, we modify the definition of Penguin(x) with.

Penguin(x) : Anc(x,y), Origin(y, "Antarctic")
          ; Anc(x,y), Origin(y, "Adelie")

We deal with a new class of structures $\mathbf{K}_3$. The effefective proofs of ¬Fly(Tweety) are:





Proof in U_3 and in DB

4.1. Intensionality.

In [8], we showed how to define constructively two intensional functions time.Q and space.Q, defining the worst-case time and space complexities of an inductive query Q, as a function of the size of the data. The average complexities can be approximated with similar methods, using statistical information associated with base relations. The uniform proofs that we introduced share these intensional properties, as an effective proof is built by solving the query. In practice given a query Q, one computes time.Q (negligeable complexity), anf if the result is greater than say 20 seconds, a process is forked, and a new query can be taken. Once the background process terminates, the constructive proof is passed to the parent process through a pipe.

References

[1] Barwise J., Moschovakis Y.
Global Inductive Definability
Journal of Symbolic Logic 43(3):521-534, 1978.

[2] Chandra A., Harel D
Structure and complexity of relational queries
Journal of Computer and System Sciences 25(1):99-128, 1982

[3] Lloyd J W.
Foundations of Logic Programming
Springer-Verlag, 1984

[4] McCarthy J.: Applications of Circumscription to Formalizing Common-Sense Knowledge.
Artificial Intelligence 28(1):89-116, 1986

[5] Moschovakis Y.
Elementary Induction on Abstract Structures.
North-Holland, 1974

[6] de Rougemont M.
The Computation of Inductive Queries by machines.
In *Logic, Language and Computation.* ASL, 1985.
Abstract in *Journal of Symbolic Logic* 51(3): 839, 1986.

[7] de Rougemont M.
Logic on finite structures and logic programming.
Computers and Artificial Intelligence 5(6), 1986.

[8] de Rougemont M.
The intensional compilation of logic programs.
In *European ASL-meeting, Hull..* ASL, 1986.

[9] Sazonov V.
Polynomial Computability and Recursivity in finite domains
Elektronische Info. und Kybernetik, (16):319-323, 1980.

[10] Sebelik J., Stepanek P.
Horn clause programs suggested by recursive functions.
In Logic program workshop., 1980.