A Coq Formalization of the Relational Data Model*

Véronique Benzaken[1], Évelyne Contejean[2], and Stefania Dumbrava[1]

[1] Université Paris Sud, LRI, France
[2] CNRS, LRI, Université Paris Sud, France

Abstract. In this article, we propose a Coq formalization of the relational data model which underlies relational database systems. More precisely, we present and formalize the data definition part of the model including integrity constraints. We model two different query language formalisms: relational algebra and conjunctive queries. We also present logical query optimization and prove the main "database theorems": algebraic equivalences, the homomorphism theorem and conjunctive query minimization.

1 Introduction

Current data management applications and systems involve increasingly massive data volumes. Surprisingly, while the amount of data stored and managed by data engines has drastically increased, little attention has been devoted to ensure that those are reliable. Obtaining strong guarantees requires the use of formal methods and mature tools. A very promising approach consists in using proof assistants such as Coq [10]. Among such systems, relational database management systems (RDBMS) are the most widespread, motivating our choice to focus on the formalization of the relational data model.

The relational model serves different related purposes: it allows to *represent* information through *relations*, to *refine* the represented information by further restricting it through *integrity constraints*. It also provides ways to *extract* information through *query languages* based on either algebra (relational algebra queries) or first-order logic (conjunctive queries). Two different equivalent versions of the relational model exist: the *unnamed* and the *named* ones. In the *unnamed* setting, the specific attributes of a relation are ignored: only the arity (*i.e.,* the number of attributes) of a relation is available to query languages. In the *named* setting, attributes are viewed as an *explicit part* of a database and are used by query languages and integrity constraints. In practice, systems such as Oracle, DB2, PostgreSQL or Microsoft Access, rely on the *named* version of the model. They do it for several reasons. First, because for modeling purposes, names carry much more semantics than column numbers. Second, query optimizers do exploit auxiliary data structures such as indexes natively based on attributes names for physical optimization purposes (exploiting different algorithms and/or indexes). We thus chose to formalize this version.

* This work is partially supported by ANR project Typex n^o 11BS0200702.

1.1 Related Work

The first attempt in that direction is found in Gonzalia [5, 6]. This work investigates different formalizations of the *unnamed* version of the model and only addresses data definition and relational algebra aspects. A more recent formalization is found in Malecha *et al.*, [8] which addresses the problem of designing a fully verified, lightweight *implementation* of a relational database system. The authors prove that their implementation meets the specification, all the proofs being written and verified in the Ynot [3] extension of Coq. However, they implemented only a *single-user* database system. Both works chose a very unrealistic data model, the unnamed version, and gave only a *partial* modeling, insofar as conjunctive queries, serious optimization techniques, as well as integrity constraint aspects, are left for future work. Any modeling aiming at being a realistic full-fledged specification of the relational model has to include all of them.

1.2 Contributions

Our long term purpose is to *prove* that *existing* systems conform to their specifications and to *verify* that programs that make intensive use of database queries are correct and *not to implement* a RDBMS in Coq. So, the first essential step is to formalize the relational model of data. We first formalize the data definition part of the model, relational algebra and conjunctive queries. Since the latter play a central role in optimization, as they admit an *exact* equivalent optimized form, we provide both a formal specification and a certified version of the algorithm that translates relational algebra queries into conjunctive queries. We then present logical query optimization, for both query formalisms, and prove the main "databases theorems": algebraic equivalences, the homomorphism theorem and conjunctive query minimization. We also provide a certified algorithm for such a minimization. As integrity constraints are central to database technology at the design level, to build relation structures (called *schemas* in the database dialect) enjoying good properties, and at the compiler and optimizer levels, to generate optimizations, we thus specify the integrity constraints part of the relational model. We model functional and general dependencies which are considered by the database community as the most important constraints. We then deal with the problem of implication, *i.e.*, inferring all logically implied constraints from a given set. Inferring *all* constraints is important, as these are intensively used for design and optimization purposes, and since, in the absence of some of them, the compiler could miss further optimizations. We provide a formal Coq proof that the inference system for functional dependencies, *a.k.a.,* Armstrong's system, is sound, complete and terminating. We formalize and prove the correctness of the procedure for deducing new general dependencies, *a.k.a.,* the *chase*. The informal presentation of all concepts is directly taken from reference textbooks on the topic [1, 11, 9]. Our formalization achieves a high degree of abstraction and modularity and is, to our knowledge, the most realistic mechanization of classic database theory to date. A last but not least contribution is that through this formalization process, we also bring insight both to the

database and to the Coq communities. To this extent, we emphasize, throughout this article, the subtleties, which were hidden (or even missing) in textbooks. Finally, we would like to stress that this formalization step is not a mere Coq exercise of style but a needed phase in the realistic verification of full fledged database management systems, and there is no way around it.

1.3 Organization

In Section 2, we present and formalize the *named version* of the relational model, in Section 3, we model relational algebra, and conjunctive queries, we formalize and extract an algorithm translating algebra queries into conjunctive ones. In Section 4, we present the logical query optimization and prove the main "database theorems": algebraic equivalences, tableaux query minimization and the homomorphism theorem. We discuss and formalize the constraint part of the model in Section 5. We conclude by drawing lessons and giving perspectives in Section 6.

2 Data Representation

Intuitively, in the relational model, data is represented by tables (relations) consisting of rows (tuples), with uniform structure and intended meaning, each of which gives information about a specific entity. For example, assuming we want to describe movies, we can represent each movie by a tuple, whose fields (attributes) could be the movie title, its director and one of its actors. Note that we will have as many rows as there are actors for a given movie. Then, assuming that movies are screened in specific locations, these could be described by a theater name, an address and a phone number. Last, one should be able to find in the Pariscope journal[3] which theater features which movie on which schedule.

2.1 Attributes, Domains, Values

Quoting [1], a set *attribute* of (names for) attributes is fixed and equipped with a total order $\leq_{\mathbf{att}}$. When different attributes should have distinct domains (or types), a mapping, *dom*, from *attribute* to *domain* is assumed. Further, an infinite set *value* is fixed. Usually, the set of attributes is assumed to be countably infinite but in our formal development this assumption was not needed. We also assume several distinct domain names (*e.g.*, "string", "integer"), which belong to set *domain*. In the database context, *domain* corresponds to the Coq notion of type. In order to have a decidable equality, we rather used our own type : Type. In our setting, *dom* is called type_of_attribute. Each value has a formal type (obtained by the function type_of_value) which belongs to type. All these assumptions are gathered in a Coq record Tuple.Rcd, whose contents will be enriched throughout this section.

[3] The Pariscope is a Parisian journal for advertising cultural events.

```
Module Tuple.
Record Rcd : Type :=
 mk_R {
  (* Basic ingredients,
     attributes, types and values *)
  attribute : Type;
  type : Type;
  value : Type;
  (* Typing attributes and values. *)
  type_of_attribute : attribute →
type;
  type_of_value : value → type;
  (* Default Values. *)
  default_value : type → value;
  ... }.
End Tuple.
```

We illustrate these definitions with our running movie example. Recall that our purpose is *not* to store an actual database schema or instance in Coq, rather the following example is intended to be a *proof of concept*.

```
Inductive attribute :=
  | Title | Director | Actor | Theater
  | Address | Phone | Schedule.
Inductive type :=
  | type_string | type_nat | type_Z.
Inductive value :=
  | Value_string : string → value
  | Value_nat : nat → value
  | Value_Z : Z → value.
Definition type_of_attribute x :=
match x with
  | Title | Director | Actor | Theater
  | Address | Phone ⇒ type_string
  | Schedule ⇒ type_nat
end.
Definition type_of_value v :=
  match v with
  | Value_string _ ⇒ type_string
  | Value_nat _ ⇒ type_nat
  | Value_Z _ ⇒ type_Z
  end.
```

There is also a more generic modeling for attributes, and in that case, for the sake of readability, we could use the Coq notations shown in [2].

2.2 Tuples

In the named perspective, tuples are characterized by their relevant attributes (for example {Title, Director, Actor} for movies). We call this the tuple's *support*. Following textbooks, we naturally model support's by finite sets. To this end, we mainly used Letouzey's MSet library [7]. To be as modular as possible, we still dissociate the specification of finite sets from the implementation. The specification is given by a record Fset.Rcd parametrized by the elements' type and contains a comparison function elt_compare. From now on, we use the notation $\stackrel{set}{=}$ to denote set equivalence, which is actually the usual mathematical extensional equality for sets: $\forall s\ s', s \stackrel{set}{=} s' \iff (\forall e, e \in s \iff e \in s')$. For the sake of readability, the usual sets operators will be denoted with their usual mathematical notations ($\cap$, $\cup$, $\setminus$, $\in$,...). Extending the record Tuple.Rcd, we further assume:

```
Module Tuple.
Record Rcd : Type := mk_R {
  (** Basic ingredients,
      attributes, domains and values *)
  ...
  A : Fset.Rcd attribute;
  (** tuples *)
  tuple : Type;
  support : tuple → set A;
  dot : tuple → attribute → value;
  mk_tuple : set A → (attribute → value) → tuple;
  support_mk_tuple_ok :
   ∀ V f, support (mk_tuple V f) =set= V;
  dot_mk_tuple_ok :
   ∀ a V f, a ∈ V → dot (mk_tuple V f) a = f a;
  FTuple : Fset.Rcd tuple;
  tuple_eq_ok : ∀ t1 t2 : tuple,
      (Fset.elt_compare FTuple t1 t2 = Eq) ⟷
       (support t1 =set= support t2 ∧
       ∀ a, a ∈ (support t1) → dot t1 a = dot t2 a)
}
```

where A models finite sets of attribute's. We still keep the type of tuples abstract and assume the existence of two functions: support, returning the relevant tuple attributes, and dot, the associated field extraction. These functions allow to characterize tuple equivalence (tuple_eq_ok) since a tuple t behaves as the pair (support t, dot t). Further, we assume the existence of the mk_tuple function which builds tuples. This and the previous modeling of attribute induce a notion of tuple

well-typedness. A tuple t is well-typed if and only if for any attribute a in its support the type of the value t.a is the type of attribute a:

```
Definition well_typed_tuple (t : tuple) := ∀ a, a ∈ (support t) → type_of_value (dot t a) = type_of_attribute a.
```

However and surprisingly, such a notion was useless to prove all the results stated in theoretical textbooks. This is an a posteriori justification of the *relevance* of the assumption that it suffices to use a unique domain for values. The previously presented record Tuple.Rcd captures exactly the abstract behavior of tuples *i.e.,* the needed properties for proving all the theorems presented hereafter. To illustrate the generality and flexibility of our specification, we give, in [2], different possible implementations for tuples. All of them satisfy the required properties and are orthogonal to the implementation of attributes. Among others, one implements tuples as pairs containing a set of attributes and a function, and sticks to the abstract definition. For instance, another one implements tuples as association lists between attributes and values.

2.3 Relations, Schemas and Instances

A distinction is made between the *database schema*, which specifies the structure of the database, and the database *instance*, which specifies its actual content: sets of tuples. In textbooks, each table is called a *relation* and has a *name*. A set relname of relation names, equipped with a suitable comparison function specified by ORN, is thus assumed. The structure of a table is given by a relation name and a *finite* set of attributes: its *sort*. The relation name, together with its sort, is called the *relation schema*. A *database schema* is a non-empty finite set of relation schemas. We choose to model database schemas with a function basesort which associates to each relname its sort. We adopted this representation because it is the most *abstract* and makes no further choices on any *concrete implementation* (*e.g.*, association lists or finite maps or even functions) of function basesort.

```
Module DatabaseSchema.
Record Rcd attribute (A : Fset.Rcd attribute) : Type :=
mk_R { (** names for relations *) relname : Type; ORN : Oset.Rcd relname; basesort : relname → set A}.
End DatabaseSchema.
```

More precisely, the basesort function will be used to relate the support of tuples (in the instance they belong to) and the structure of the corresponding relation name.

```
Definition well_sorted_instance (I : relname → setT) :=
    ∀ (r : relname) (t : tuple), t ∈ (I r) → support t =set= basesort r.
```

It is important to mention that, in all our further development, the notion of well-sorted instance resulted *central to the correctness* of many theorems.

3 Queries

Queries allow the extraction of information from tables. The result of a query is also a table or a collection of tables. Information extraction is usually performed by a query language, the standard being SQL or QBE. All these languages rely

on a more formal basis: *relational algebra* or *first-order logic*. Both formalisms are based on the notion of tuples. Thus, we assume the existence of a record T of type Tuple.Rcd, for representing tuples, as well as of a record DBS of type DatabaseSchema.Rcd, for representing base relations. Moreover, we assume that T and DBS use the same representation, A, for finite sets of attributes. This is achieved by parameterizing DBS by (A T). For the sake of readability, we shall omit all extra (implicit) record arguments and denote by setA and setT finite sets of attributes and tuples respectively.

3.1 Relational Algebra

Relational algebra consists of a set of (algebraic) operators with relations as operands. The algebra we shall consider in this article is the *SPJRU(ID)*, where S stands for selection, P for projection, J for natural join, R for renaming and last U for union. Though intersection (I) and difference (D) are not part of the SPJRU minimal algebra, we decided to include them at this point, as they are usually part of commercial query languages. In the context of the named version, the natural way to combine relations is the natural join, whereas in the unnamed one it is the Cartesian product. The complete definition of queries is given in Figure 1. In our development, we chose, as far as possible, *not to embed proofs* in types. Hence, types are much more concise and readable.

```
Inductive query : Type :=
  | Query_Basename : relname → query
  | Query_Sigma : formula → query → query
  | Query_Pi : setA → query → query
  | Query_NaturalJoin : query → query → query
  | Query_Rename : renaming → query → query
  | Query_Union : query → query → query
  | Query_Inter : query → query → query
  | Query_Diff : query → query → query
with variable : Type :=
 | Var : query → varname → variable
with term : Type :=
  | Term_Constant : value → term
  | Term_Dot : variable → attribute → term
with atom : Type :=
 | Atom_Eq : term → term → atom
 | Atom_Le : term → term → atom
with formula : Type :=
  | Formula_Atom : atom → formula
  | Formula_And : formula → formula → formula
  | Formula_Or : formula → formula → formula
  | Formula_Not : formula → formula
  | Formula_Forall : variable → formula → formula
  | Formula_Exists : variable → formula → formula.
```

Fig. 1. Queries

Syntax Base relations are queries. Concerning the selection operator, in textbooks, it has the form $\sigma_{A=a}$ or $\sigma_{A=B}$, where $A, B \in \mathit{attribute}$ and $a \in \mathit{value}$. The notation $A = a$ ($A = B$ resp.,) is improper and corresponds to $x.A = a$ ($x.A = x.B$ resp.,) where x is a free variable. Given a set of tuples $\mathcal{I}$, with the same support S, we shall call S the *sort* of $\mathcal{I}$. The selection applies to any set of tuples $\mathcal{I}$ of sort S, (with $A, B \in S$) and yields an output of sort S. The semantics of the operator is $\sigma_f(\mathcal{I}) = \{t \mid t \in \mathcal{I} \wedge f\{x \to t\}\}$ where $f\{x \to t\}$ stands for "t satisfies formula f", x being the only free variable of f. Formula

satisfaction is based on the standard underlying interpretation. Since, in another context (database program verification) we use general first-order formulas, we chose to model selection's (filtering) conditions with them, rather than restricting ourselves to the simpler case found in textbooks. We first introduce names for variables:

```
Inductive varname : Set := VarN : N → varname.
```

Then formulas are built in the standard way from equality and inequality atoms which compare either constants or tuples' field extractions. However, one should notice that variables are used to denote tuples in the output of specific queries, therefore containing information about the query itself. For example variable x below is intended to represent any tuple in the Movies relation while formula f corresponds to x ∈ Movies ⇒ x.Director = "Fellini".

```
Notation x := (Var (Query_Basename Movies) (VarN 0)).
Definition f := (* x ∈ Movies ⇒ x.Director = "Fellini" *)
 (Formula_Atom (Atom_Eq (Term_Dot x Director) (Term_Constant (Coq_string "Fellini")))).
```

The projection operator has the form $\pi_{\{A1,\ldots,An\}}$, $n \geq 0$ and operates on all inputs, $\mathcal{I}$, whose sort contains the subset of attributes $W = \{A_1, \ldots, A_n\}$ and produces an output of sort W. The semantics of projection is $\pi_W(\mathcal{I}) = \{t|_W \mid t \in \mathcal{I}\}$ where the notation $t|_W$ represents the tuple obtained from t by keeping only the attributes in W. Remember that setA denotes finite sets of attributes and embeds as an implicit argument (A T), the record representing all types and operations on finite sets. Depending on the actual implementation of sets, this definition may contain some proofs in the setA data type. For instance the proof that a set is an AVL tree may be part of the type. The natural join operator, denoted $\bowtie$, takes arbitrary inputs $\mathcal{I}_1$ and $\mathcal{I}_2$ having sorts V and W, respectively, and produces an output with sort equal to $V \cup W$. The semantics is, $\mathcal{I}_1 \bowtie \mathcal{I}_2 = \{t \mid \exists v \in \mathcal{I}_1, \exists w \in \mathcal{I}_2, t|_V = v \wedge t|_W = w\}$. When $sort(\mathcal{I}_1) = sort(\mathcal{I}_2)$, then $\mathcal{I}_1 \bowtie \mathcal{I}_2 = \mathcal{I}_1 \cap \mathcal{I}_2$, and when $sort(\mathcal{I}_1) \cap sort(\mathcal{I}_2) = \emptyset$, then $\mathcal{I}_1 \bowtie \mathcal{I}_2$ is the cross-product of $\mathcal{I}_1$ and $\mathcal{I}_2$. The join operator is associative and commutative. An attribute renaming for a finite set V of attributes is a one-one mapping from V to *attribute*. In textbooks, an attribute renaming g for V is specified by the set of pairs $(a, g(a))$, where $g(a) \neq a$; this is usually written as $a_1 a_2 \ldots a_n \rightarrow b_1 b_2 \ldots b_n$ to indicate that $g(a_i) = b_i$ for each $i \in [1, n], n \geq 0$. A renaming operator for inputs over V is an expression ρ_g, where g is an attribute renaming for V; this maps to outputs over $g[V]$. Precisely, for $\mathcal{I}$ over V, $\rho_g(\mathcal{I}) = \{v \mid \exists u \in \mathcal{I}, \forall a \in V, v(g(a)) = u(a)\}$. We made a different more abstract choice to model this operator. To avoid proofs in types, we made no assumptions on the "renaming" function except for its type attribute → attribute in the inductive definition. However, the one-to-one assumption will explicitly appear as an hypothesis for some theorems. Set operators can be applied over sets of tuples, $\mathcal{I}_1, \mathcal{I}_2$, with the same sort. As standard in mathematics, $\mathcal{I}_1 \cup \mathcal{I}_2$ (resp. $\mathcal{I}_1 \cap \mathcal{I}_2, \mathcal{I}_1 \setminus \mathcal{I}_2$) is the set having this same sort and containing the union (resp., intersection, difference) of the two sets of tuples. Sort compatibility constraints are absent in our modeling so as to avoid proofs and will be enforced in the semantics part.

Semantics We present our Coq modeling of query evaluation. We, hence, have to explicitly describe constraints about sorts, which were, deliberately, left out of the query syntax. For base queries, the sort corresponds to the basesort of the relation name, for selections, the sort is left unchanged and for joins, the sort is as expected the union of sorts. The cases which are of interest are projections, renaming and set theoretic operators. For projections, rather than imposing that the set W of attributes on which we project, be a subset of the sort of q1, we chose to define the sort of Query_Pi W q1 as their intersection (W ∩ sort q1). For renaming, we check that the corresponding function rho behaves as expected, *i.e.*, that it is a one-to-one mapping over attributes in q1; otherwise the sort of the query is empty. Last, for set theoretic operators, if the input's sorts are not compatible, the sort of the query is empty. This is formally defined by:

```
Fixpoint sort (q : query) : setA := match q with
  | Query_Basename r ⇒ basesort r
  | Query_Sigma _ q1 ⇒ sort q1
  | Query_Pi W q1 ⇒ W ∩ sort q1
  | Query_Join q1 q2 ⇒ sort q1 ∪ sort q2
  | Query_Rename rho q1 ⇒
    let sort_q1 := sort q1 in
      if one_to_one_renaming_bool sort_q1 rho
      then fset_map A A rho sort_q1
      else ∅
  | Query_Union q1 q2 | Query_Inter q1 q2
  | Query_Diff q1 q2 ⇒
    let sort_q1 := sort q1 in
      if sort_q1 =?^set sort q2
      then sort_q1
      else ∅
end.
```

At this point we are ready to interpret queries. We first assume an interpretation for base relations. When we shall prove the usual structural equivalence theorems (Section 4) for query optimization, we shall impose that queries' results are *well-sorted*. This means that all tuples in an instance or query evaluation must have the *same* support which is the sort of the query. This property is inherited from base instances as stated below:

```
Lemma well_sorted_query : ∀ (I : relname → setT), well_sorted_instance I →
    ∀ (q : query) (t : tuple), t ∈ (eval_query I q) → support t =^set sort q.
```

Query evaluation is inductively defined from a given interpretation I for base relations. We sketch its structure (the complete definition of eval_query is given in [2]) in order to emphasize the fact that the same tests as for sorts, are performed. For example, for renaming, if the corresponding function is not suitable, the query evaluates to the empty set of tuples.

```
Fixpoint eval_query I (q : query) : setT := match q with
  | Query_Basename r ⇒ I r
  | Query_Sigma f q1 ⇒ ...
  | Query_Pi W q1 ⇒ ...
  | Query_Join q1 q2 ⇒ ...
  | Query_Rename rho q1 ⇒
    let sort_q1 := sort q1 in
      if one_to_one_renaming_bool sort_q1 rho
      then ...
      else ∅
  | Query_Union q1 q2 ⇒
      if sort q1 =?^set sort q2
      then (eval_query I q1) ∪ (eval_query I q2)
      else ∅
  | Query_Inter q1 q2 ⇒ if sort q1 =?^set sort q2 ...
  | Query_Diff q1 q2 ⇒ if sort q1 =?^set sort q2 ...
end.
```

Our definition enjoys the standard properties stated in all database textbooks which are expressed in our framework by the following lemmas. We only present some of them ; the full list, as well as the complete code, is given in [2]. In particular, the way terms, atoms and formulas are interpreted is detailed. For the sake of readability we used some syntactic sugar, such as $\stackrel{I}{=}$, $\in_I$, as well as f {x → t}, for the interpretation of formula f under assignment x → t.

```
Notation query_eq q1 q2 := (eval_query I q1 =^set eval_query I q2).

Infix " =^I= " := query_eq.
Notation "t '∈_I ' q" := t ∈ (eval_query I q).
Lemma mem_Basename : ∀ I r t, t ∈_I (Query_Basename r) ⟷ t ∈ (I r).

Lemma mem_Inter : ∀ I q1 q2, sort q1 =^set sort q2 → ∀ t, t ∈_I (Query_Inter q1 q2) ⟷ (t ∈_I q1 ⋀ t ∈_I q2).
Lemma mem_Sigma : ∀ I, well_sorted_instance I → ∀ f x q t, set_of_attributes_f f ⊆ sort q →
    Fset.elements FV (free_variables_f f) = x :: nil →
    (t ∈_I (Query_Sigma f q) ⟷ (t ∈_I q ⋀ f {x → t} = true)).
Lemma mem_Pi : ∀ I, well_sorted_instance I →
  ∀ W q t, t ∈_I Query_Pi W q ⟷ ∃t', (t' ∈_I q ⋀ t =^t mk_tuple (W ∩ sort q) (dot t')).
Lemma mem_Join : ∀ I, well_sorted_instance I → ∀ q1 q2 t,
  t ∈_I Query_Join q1 q2 ⟷
    ∃t1, ∃t2, (t1 ∈_I q1 ⋀ t2 ∈_I q2 ⋀ (∀ a, a ∈ sort q1 ∩ sort q2 → dot t1 a = dot t2 a) ⋀
    t =^t mk_tuple (sort q1 ∪ sort q2) (fun a ⇒ if a ∈_? (sort q1) then dot t1 a else dot t2 a)).
Lemma mem_Rename : ∀ I, well_sorted_instance I → ∀ rho q, one_to_one_renaming (sort q) rho →
    ∀ t, t ∈_I (Query_Rename rho q) ⟷ (∃ t', t' ∈_I q ⋀ t =^t rename_tuple rho t').
Lemma NaturalJoin_Inter : ∀ I, well_sorted_instance I → ∀ q1 q2, sort q1 =^set sort q2 →
    Query_NaturalJoin q1 q2 =^I= Query_Inter q1 q2.
```

Those lemmas highlight the heterogeneous nature of relational operators. In order to prove that they enjoy their usual semantics, on the one hand, the purely set theoretic ones, only need sort compatibility conditions, on the other hand, the database ones need well-sortedness. Interestingly, the lemma, NaturalJoin_Inter, bridging both worlds, needs both.

3.2 Conjunctive Queries

In this context, the query language is slightly different. Rather than relying on algebraic operators, queries are expressed by logical formulas of the form $\{(a_1, \ldots, a_n) \mid \exists b_1, \ldots \exists b_m, P_1 \wedge \ldots \wedge P_k\}$, where the a_i, b_i denote variables which will be interpreted by values and where P_i's denote either equalities or membership to a base relation. For example the query: "Which of "Fellini" 's movies are played at the cinema "Action Christine" ?" expressed by the following relational algebra expression:

$$\pi_{\{\text{Title, Director, Actor}\}}(\sigma_{\substack{\text{x.Director="Fellini"} \wedge \\ \text{x.Theater = "Action Christine"}}}(\texttt{Movies} \bowtie \texttt{Pariscope}))$$

will be:

$$\left\{(t,d,a) \mid \begin{array}{l} \exists th, \exists t', \exists s, \texttt{Movies}(t,d,a) \wedge \texttt{Pariscope}(th,t',s) \\ \wedge t = t' \wedge d = \text{"Fellini"} \wedge th = \text{"Action Christine"} \end{array}\right\}$$

Quoting [1], "if we blur the difference between a variable and a constant, the body of a conjunctive query can be seen as an instance with additional constraints". This leads to the notion of extended tuples mapping attributes to either *constants* or *variables*. Hence, a *tableau* over a schema is defined exactly as was the notion of an instance over this schema, except that it contains extended tuples. A *conjunctive query* is simply a pair *(T, s)* where T is a tableau and s, an extended tuple called the *summary* of the query. Variables occurring in s are called *distinguished variables* or *distinguished symbols* in textbooks. The

summary s in query (T, s) represents the answer to the query which consists of all tuples for which the pattern described by T is found in the database. This formulation of queries is closest to the QBE visual form. Equality conditions are embedded in the tableau itself as shown by the following example:

Title	Director	Actor	Theater	Schedule	
t	''Fellini''	a			Movies
t			''Action Christine''	s	Pariscope
t	d	a			summary

Syntax The formal way to "blur" the differences between variables and constants (value's in our modeling) is achieved by embedding them in a single Coq type tvar.

```
Inductive tvar : Type := Tvar : nat → tvar | Tval : value → tvar.
Inductive trow : Type := Trow : relname → (attribute → tvar) → trow.
```

Notice that a row, modeled by type trow, is tagged by a relation name (its first argument) and gathers variables and constants thanks to its second argument. For instance the first row of the above query is:

```
Trow Movies (fun a : attribute ⇒ match a with | Title ⇒ Tvar 0 | Director ⇒ Tval ''Fellini'' | ... end)
```

A tableau is a set of trow's. This set is built using a comparison function similar to the one for tuples. Next a summary is tagged by a set of relevant attributes and maps attribute's to tvar's. Last a conjunctive query consists of a tableau and a summary.

```
Notation setR := (Fset.set (Ftrow T DBS)).
Definition tableau := SetR.
Inductive summary : Type := Summary : setA → (attribute → tvar) → summary.
Definition tableau_query := (tableau * summary)
```

Semantics Let us grasp, through our former example, the semantics of such queries. This query is expressed by the summary

```
Summary (mk_set A (Title :: Director :: Actor :: nil))
 (fun a : attribute ⇒ match a with | Title ⇒ Tvar 0 | Director ⇒ Tvar 1 | ... end)
```

and its result consists in the set of movies

```
mk_set A ((mk_movie "Casanova" "Fellini" "Donald Sutherland") ::
          (mk_movie "La strada" "Fellini" "Giulietta Masini") :: nil)
```

This set is computed by composing the summary function with some mappings from variables in the tableau rows to values, hence mapping summaries to tuples. Thus, we first need to define the notion of valuation which, as usual, maps variables to values. More precisely, in our case, as we embedded variables and constants in a single abstract type tvar, and because variables are characterized by their nat identifier, the type of valuation is nat→ value. Hence applying a valuation (thanks to apply_valuation) on constants consists in applying the identity function.

```
Definition valuation := nat → value.
Definition apply_valuation (ν : valuation) (x : tvar) : value := match x with | Tvar n ⇒ ν n | Tval c ⇒ c end.
Notation "ν '⟦' x '⟧'" := (apply_valuation ν x).
```

Valuations naturally extend to trow's and summary's, yielding tuples, and to tableaux, yielding sets of tuples.

```
Definition apply_valuation_t (ν : valuation) (x : trow) : tuple :=
 match x with Trow r f ⇒ mk_tuple (basesort r) (fun a ⇒ ν ⟦f a⟧) end.
Notation "ν '⟦' x '⟧_t'" := (apply_valuation_t ν x).
Definition apply_valuation_s (ν : valuation) (x : summary) : tuple :=
  match x with Summary V f ⇒ mk_tuple V (fun a ⇒ ν ⟦f a⟧) end.
Notation "ν '⟦' x '⟧_s'" := (apply_valuation_s ν x).
```

Given a query (T, s), its result on instance $\mathcal{I}$ is given by $\{t \mid \exists \nu, \nu(T) \subseteq \mathcal{I} \wedge t = \nu(s)\}$ where ν is a valuation. In our development, we characterize this set by the predicate is_a_solution I (T, s), where $\stackrel{t}{=}$ denotes the equivalence of tuples.

```
Inductive is_a_solution (I : relname → setT) : tableau_query → tuple → Prop :=
  | Extract : ∀ (ST : tableau) (s : summary) (ν : valuation),
      (∀ (r : relname) (f : attribute → tvar), (Trow r f) ∈ ST → ν ⟦Trow r f⟧_t ∈_I (Query_Basename r)) →
      ∀ (t : tuple), t =^t ν ⟦s⟧_s → is_a_solution I (ST, s) t.
```

3.3 From Algebra Queries to Conjunctive Queries

The two formalisms presented are not exactly equivalent except in the case where relational queries are only built with selections, projections and joins. In this case there is an *apparently straightforward* way to construct the corresponding conjunctive query. We give hereafter the algorithm found in [11] as it is presented. If we try to apply this algorithm on the following relational ex-

Given an SPJ algebraic expression, a conjunctive query equivalent to this expression is inductively constructed using the following rules. The base case consists in a relation $r(A_1, \ldots, A_n)$ the corresponding tableau consists in a single row and summary which are exactly the same with one variable for each A_i. Assume that we have an expression of the form $\pi_W(E)$ and that we have constructed (T, s) for E, then to reflect the projection, all the distinguished variables that are not in W are deleted from s. For selections $\sigma_f(E)$ where f is either of the form $A = B$ or $A = c$, in the former case, the distinguished symbols for columns A and B in the summary and the tableau are identified, in the latter, the distinguished variable for A is replaced by c. For joins $E_1 \bowtie E_2$, it is assumed without loss of generality that if both (T_1, s_1) and (T_2, s_2) have distinguished symbols in the summary column for attribute A then those symbols are the same, but that otherwise (T_1, s_1) and (T_2, s_2) have no symbols in common. Then the tableau for $E_1 \bowtie E_2$ has a summary in which a column has a distinguished symbol a if a appears as a distinguished symbol in that column of s_1 or s_2 or both. The new tableau has as rows all the rows of T_1 and T_2.

Fig. 2. Ullman's book algorithm presentation

pression $\sigma_{A=B}(r) \bowtie \sigma_{B=C}(r)$, we obtain for E_1 and E_2 the following tableaux: $\frac{x_1\ x_1\ x_2\ \mathrm{r}}{x_1\ x_1\ x_2}$ and $\frac{y_1\ y_2\ y_2\ \mathrm{r}}{y_1\ y_2\ y_2}$. Given those two tableaux whatever renaming we choose to apply to the second one as stated in [11] there is no way to be in the situation described by the algorithm, *i.e., if both (T_1, s_1) and (T_2, s_2) have distinguished symbols in the summary column for attribute A then those symbols are the same*. We fixed this source of incompleteness by using *unification* instead of renaming. If we unify the two summaries of our example, we obtain

$x_2 \mapsto x_1; y_1 \mapsto x_1; y_2 \mapsto x_1$ yielding the tableau $\dfrac{x_1\ x_1\ x_1\ \mathrm{r}}{x_1\ x_1\ x_1}$ which indeed corresponds to what is expected in terms of semantics. The unify function, given in [2], is readable but the proofs of its soundness (the result of unify is a unifier) and completeness (whenever there is a unifier, unify finds it) took more than 4000 lines of code. Thanks to it we are able to express the translation algorithm, also given in [2], which is sound and complete and handles all SPJ queries. If the selection condition is a conjunction of equalities, a preprocessing step, expand_query, transforms it into a sequence of selections whose conditions are equalities. The translation yields either an equivalent query, EmptyRel when the original query has no solution or NoTranslation when the input query is not SPJ. The translation algorithm relies on several auxiliary functions. The first one, fresh_row n r, is used for the base case and generates a row, Trow r fr, tagged by relation name r. Function fr maps attributes to fresh variables starting from index n. The second one rename t1 t2 is used for selections with condition t1 = t2 and returns either None if t1 and t2 are distinct constants or Some rho where rho is a substitution which replaces one of the t1 and t2 by the other one, avoiding to replace a constant by a variable. In that case rho is applied to the whole tableau. The only case where unify is needed is for joins. In this case the translation is applied to both operands and then compatibility on common attributes is ensured by applying the resulting substitution to the whole query. The following lemma states that the algorithm behaves as expected.

Our formalization helped us in making precise the exact behavior of the translation algorithm. In the informal presentation taken from textbooks, an *underlying assumption* is made about *freshness of variables* for the base case, which is quite tedious to handle at the formal level. To our knowledge our algorithm is the *first one* for such a translation which is formally specified and fully proved.

```
Lemma algebra_to_tableau_expand_is_complete :
 ∀ (q : query) (n : nat) (I : relname → setT),
   well_sorted_instance I →
   match algebra_to_tableau (S n) (expand_query q) with
     | TQ _ Ts ⇒
         ∀ t, is_a_solution I Ts t ⟷ t ∈ (eval_query I q)
     | EmptyRel ⇒ ∀ t, t ∈ (eval_query I q) → False
     | NoTranslation ⇒ translatable_q q = false
   end.
```

4 Logical Optimization

4.1 Optimizing Relational Algebra Queries

Query optimization exploits algebraic equivalences. Such equivalences are found in all textbooks and in particular in [9]. We list the most classical ones hereafter.

$$\sigma_{f_1 \wedge f_2}(q) \equiv \sigma_{f_1}(\sigma_{f_2}(q)) \quad (1) \qquad \pi_{W_1}(\pi_{W_2}(q)) \equiv \pi_{W_1}(q) \quad \text{if } W_1 \subseteq W_2 \quad (5)$$

$$\sigma_{f_1}(\sigma_{f_2}(q)) \equiv \sigma_{f_2}(\sigma_{f_1}(q)) \quad (2) \qquad \pi_W(\sigma_f(q)) \equiv \sigma_f(\pi_W(q)) \quad \text{if } \mathcal{A}tt(f) \subseteq W \quad (6)$$

$$(q_1 \bowtie q_2) \bowtie q_3 \equiv q_1 \bowtie (q_2 \bowtie q_3) \quad (3) \qquad \sigma_f(q_1 \bowtie q_2) \equiv \sigma_f(q_1) \bowtie q_2 \quad \text{if } \mathcal{A}tt(f) \subseteq sort(q_1) \quad (7)$$

$$q_1 \bowtie q_2 \equiv q_2 \bowtie q_1 \quad (4) \qquad \sigma_f(q_1 \nabla q_2) \equiv \sigma_f(q_1) \nabla \sigma_f(q_2) \quad \text{where } \nabla \text{ is } \cup, \cap \text{ or } \setminus \quad (8)$$

All these have been formally proved and their formal statements are given in [2].

Although not technically involved, all the proofs relied on the assumption that instances are *well sorted*. To illustrate this, we give the formal statement of (7).

```
Lemma Sigma_NaturalJoin_comm : ∀ I, well_sorted_instance I → ∀ f q1 q2, set_of_attributes_f f ⊆ sort q1 →
 Query_Sigma f (Query_NaturalJoin q1 q2) ≡^I Query_NaturalJoin (Query_Sigma f q1) q2.
```

4.2 Optimizing Conjunctive Queries

For the algebraic queries that are expressible by a conjunctive query, there exists an exact optimization technique. In this case, query optimization is based on the following consideration: the number of lines in the tableau corresponds to the number of joins (plus one) in the relational expression. Therefore, the optimization consists in reducing this number of lines. This is achieved through the notions of tableaux containment and equivalence and finally through a minimality condition. More precisely, let (T_1, s_1) and (T_2, s_2) be two conjunctive queries, (T_1, s_1) is contained in (T_2, s_2) written $(T_1, s_1) \subseteq (T_2, s_2)$ iff (T_1, s_1) and (T_2, s_2) have the same set of attributes, and, for all relations' instances, solutions of (T_1, s_1) are included in the set of solutions of (T_2, s_2). This inclusion relation naturally induces an equivalence. $(T_1, s_1) \equiv (T_2, s_2)$ iff $(T_1, s_1) \subseteq (T_2, s_2)$ and $(T_2, s_2) \subseteq (T_1, s_1)$. This is formalized in Coq by:

```
Definition is_contained_instance I Ts1 Ts2 := ∀ (t : tuple), is_a_solution I Ts1 t → is_a_solution I Ts2 t.
Definition is_contained Ts1 Ts2 := ∀ I, is_contained_instance I Ts1 Ts2.
Definition are_equivalent Ts1 Ts2 := is_contained Ts1 Ts2 ⋀ is_contained Ts2 Ts1.
```

These semantical notions can be checked syntactically relying on tableaux's substitutions. A (tableau) substitution θ is a mapping from variables to variables or constants. The following database theorem expresses this syntactical characterization of containment.

Theorem 1 (Tableaux Homomorphism). *If (T_1, s_1) and (T_2, s_2) are conjunctive queries, $(T_1, s_1) \subseteq (T_2, s_2)$ iff there exists a substitution tableau θ such that for all line t tagged by relation name r in T_2, $\theta(t)$ occurs tagged by r in T_1, and $\theta(s_2) = s_1$. θ is called a tableau homomorphism from (T_2, s_2) to (T_1, s_1).*

We first give the definition of substitution in our setting and then formally define the application of a substitution to a variable. This notion extends to trow's and summary's. Then we provide the formal definition of tableau homomorphism and state the homomorphism theorem.

```
Definition substitution := nat → tvar.
Definition apply_subst_tvar (θ : substitution) (x : tvar) := match x with Tvar n ⇒ θ n | Tval _ ⇒ x end.
Notation "θ '[' x ']_v'" := (apply_subst_tvar θ x).
Definition tableau_homomorphism (θ : substitution) Ts2 Ts1 :=
match Ts1, Ts2 with (T1, s1), (T2, s2) ⇒ (fset_map Ftrow Ftrow (fun t ⇒ θ [t]_t) T2) ⊆ T1 ⋀ θ [s2]_s ≡^s s1
end.
Theorem Homomorphism_theorem :
   ∀ Ts1 Ts2, (∃ θ , tableau_homomorphism θ Ts2 Ts1) ⟷ is_contained Ts1 Ts2.
```

We briefly sketch the proof of the homomorphism theorem. Interestingly in textbooks a lot of material is hidden. Namely, the notion of *fresh constants* is central to the proof in order to be able to define a list of such *distinct* fresh constants for each variable present in the query. We assume therefore

```
Hypothesis fresh : (Fset.set Ftvar) → value.
Hypothesis fresh_is_fresh : ∀ lval, (Tval (fresh lval)) ∈ lval → False.
```

This implies that domains are *infinite*. Based on fresh constants we define a variable assignment μ from variables to new fresh abstract constants on (T_1, s_1). We then show that μ is a solution of (T_1, s_1) w.r.t. the interpretation I which contains exactly $\mu(T_1)$. Thanks to the definition of tableaux containment, μ is a solution of (T_2, s_2) w.r.t. I. Hence there is an assignment ν which corresponds to a solution of (T_2, s_2), $\nu(s_2) = \mu(s_1) \wedge (\forall t_2\ r, t_2 : r \in T_2 \Rightarrow \nu(t_2) \in I(r))$, that is $\nu(s_2) = \mu(s_1) \ \ \wedge \ \ (\forall\ t_2\ r, \ \ t_2 : r \in T_2 \Rightarrow \exists\ t_1, \ \ t_1 : r \in T_1 \wedge \nu(t_2) = \mu(t_1))$. By construction μ admits an inverse function defined over the variables of (T_1, s_1). What remains to show is that $x \mapsto \mu^{-1}(\nu(x))$ is an homomorphism from (T_2, s_2) to (T_1, s_1). The main difficulties encountered in Coq were to properly define the notion of query solution, to build the variable assignment μ as a function from the fresh function and to prove that μ is injective.

At this point, based on the homomorphism theorem, given a conjunctive query, we shall explicitly construct an equivalent minimal one. Indeed another, well known, database theorem states that for each conjunctive query there exists a minimal equivalent query among its sub-queries. A sub-query of (T, s) is simply (T', s) such that $T' \subseteq T$. Hence, the optimization process consists in inspecting all equivalent sub-tableaux and among those keeping a minimal one.

```
Definition min_tableau Ts Ms :=
  are_equivalent Ts Ms ⋀ (∀ Ts', are_equivalent Ts Ts' → cardinal (fst Ms) ≤ cardinal (fst Ts')).
Lemma tableaux_optimisation : ∀ T s, {T' | min_tableau (T, s) (T', s)}.
```

More precisely, the corner stone of the algorithm is to find an homomorphism from the initial tableau to a given sub-tableau. To do so we used a function abstract_matching. All further details are given in [2]. Not only do we *prove* this result but we also provide a *certified algorithm* to build this minimal tableau both in Coq and by *extraction* from tableaux_optimization in OCaml.

5 Integrity Constraints

Constraints are captured by the theory of dependencies which deal with the semantics of data. For example, returning to our running example, we may know that there is only one director associated with each movie title. Such properties are called *functional dependencies* because the values of some attributes of a tuple uniquely determine the values of other attributes of that tuple. Let us further assume that we have another relation: *Showings(Theater, Screen, Title, Snack)* which contains tuples *(th, sc, ti, sn)* if the theater *th* is showing the movie *ti* on the screen *sc* and if the theater *th* offers snack *sn*. Intuitively, one would expect a certain independence between the *Screen-Title* attributes, on the one hand, and the *Snack* attribute, on the other, for a given value of *Theater*. For example, if *(Action Christine, 1, Casanova, Coffee)* and *(Action Christine, 2, M, Tea)* are in *Showings*, we also expect *(Action Christine, 1, Casanova, Tea)* and *(Action Christine, 2, M, Coffee)* to be present. Such dependencies are called tuple generating dependencies. Functional and tuple generating dependencies

fall under the wider class of *general dependencies* which we model and that also capture inclusion dependencies which correspond to foreign key constraints in real systems. First we introduce functional dependencies, then we present the class of general dependencies. An important problem concerning dependencies is that of the so called *logical implication*: given a set of constraints, what other constraints could be inferred ? Armstrong's system that allows to deduce, in the functional case, all dependencies implied by a given set, is sound, complete and terminating. We then detail the chase, a procedure that allows to infer general dependencies, and prove its soundness.

5.1 Functional Dependencies

A *functional dependency (fd)* expresses a constraint between schema attribute sets. Specifically, given a database schema R, an instance r of R and attribute sets V and W (in the sort of R), a functional dependency $V \hookrightarrow W$ over r, denoted $r \models V \hookrightarrow W$, *holds* if $\forall t_1\ t_2, t_1 \in r \Rightarrow t_2 \in r \Rightarrow t_1|_V = t_2|_V \Rightarrow t_1|_W = t_2|_W$.

Let F be a set of functional dependencies over a given schema R. A functional dependency $d = X \hookrightarrow Y$ is *semantically implied* by F, denoted $F \models d$, if $\forall r : R, (r \models F \Rightarrow r \models d)$. This is formally defined in Coq by:

```
Inductive fd : Type := FD : setA → setA → fd.
Notation "V '↪' W" := (FD V W).
Definition fd_sem (ST : setT) (d : fd) := match d with | V ↪W ⇒
∀ t1 t2, t1 ∈ ST → t2 ∈ ST → (∀ x, x ∈ V → dot T t1 x = dot T t2 x)
          → ∀ y, y ∈ W → dot T t1 y = dot T t2 y
end.
```

Armstrong's inference system $\mathcal{A}$ is modeled via the dtree inductive definition, representing a derivation tree, whose branches are the axioms above and the D_ax rule, for deriving dependencies already in the context and where setF denotes the type of sets of dependencies.

```
Inductive dtree (F : setF) : fd → Type :=
 | D_Ax : ∀ X Y, (X ↪Y) ∈ F → dtree F (X ↪Y)
 | D_Refl : ∀ X Y, Y ⊆ X → dtree F (X ↪Y)
 | D_Aug : ∀ X Y Z XZ YZ, XZ =set (X ∪ Z) → YZ =set (Y ∪ Z) → dtree F (X ↪Y) → dtree F (XZ ↪YZ)
 | D_Trans : ∀ X Y Y' Z, Y =set Y' → dtree F (X ↪Y) → dtree F (Y' ↪Z) → dtree F (X ↪Z).

Theorem Armstrong_soundness : ∀ F d (t : dtree F d) ST, (∀ f, f ∈ F → fd_sem ST f) → fd_sem ST d.
```

This theorem is formally proven by an easy induction on the derivation tree. The completeness proof borrows from [11] the central idea of building a *model* M. Given a set of dependencies F and a set of attributes X, M consists of two tuples t0 and t1, which only agree on the closure attribute set $[X]_F^+$. The constructive proof of completeness is simply based on the fact that if $F \models X \hookrightarrow Y$, since M is a model of F, then M is a model of $X \hookrightarrow Y$.

```
Lemma Armstrong_completeness : ∀ U F X Y, X ⊆ U → Y ⊆ U → (∀ ST, (∀ t, t ∈ ST → support T t =set U)
  → (∀ f, f ∈ F → fd_sem ST f) → fd_sem ST (X ↪Y)) → (dtree F (X ↪Y)).
```

Interestingly, while for soundness the hypotheses did not make any assumption on the finiteness of the attribute universe, for the completeness, this assumption was needed. All intermediate lemmas are given in [2] and the main theorem explicitly mentions the fact that all sets of attributes are included in the finite universe U and that the values zero and one are distinct.

5.2 General Dependencies

Constraints described in textbooks (functional, join or inclusion dependencies) are first-order logic sentences of the form

$$\forall x_1 \ldots \forall x_n (\phi(x_1, \ldots, x_n) \Rightarrow \exists z_1 \ldots \exists z_k \psi(x_1, \ldots, x_n, z_1, \ldots, z_k)),$$

where ϕ is a (possibly empty) conjunction of atoms and ψ an atom. In both ϕ and ψ, one finds relation atoms of the form $r(w_1, \ldots, w_l)$ and equality atoms of the form $w = w'$, where each of the $w, w', w_1, \ldots, w_l$ is a variable or a constant. Inclusion dependencies can be expressed by $\forall x_1 \ldots \forall x_n (r_1(x_1, \ldots, x_n) \Rightarrow r_2(x_1, \ldots, x_n))$. According to textbooks, the semantics of such formulas is the natural one. There is a strong relationship between general dependencies and tableaux which provides a convenient notation for expressing and working with dependencies. For example the functional dependency $A \hookrightarrow B$ on relation $r(A, B)$, is represented by the following formula $\forall v, v_1, v_2 \quad r(v, v_1) \wedge r(v, v_2) \Rightarrow v_1 = v_2$

and conjunctive query $\dfrac{\begin{array}{cc} A & B \\ v & v_1 \quad r \\ v & v_2 \quad r \end{array}}{v_1 = v_2}$. When the right part of the implication is a relation predicate, the last line is a summary and such dependencies are referred as "tuple generating" while the other ones are referred as "equality generating". We model this by the following inductive definition of gd, according to whether ϕ is a relation predicate or an equality, we use two constructors TupleGen or EqGen.

```
Notation "s1 ' =^r= ' s2" := (** equivalence of rows **) (Fset.elt_compare Ftrow s1 s2 = Eq).
Inductive gd := TupleGen : setR → trow → gd | EqGen : setR → tvar → tvar → gd.
```

The natural semantics is provided by:

```
Inductive gd_sem : gd → setT → Prop :=
 | TupleGenSem : ∀ (SR : setR) (s : trow) (ST : setT), (∀ (ν : valuation), (∀ x, x ∈ SR → (ν ⟦x⟧_t) ∈ ST) →
     ∃ν_e , (∀ x, x ∈ variables_tableau SR → ν_e ⟦x⟧ = ν ⟦x⟧) ⋀ ν_e ⟦s⟧_t ∈ ST) → gd_sem (TupleGen SR s) ST
 | EqGenSem : ∀ (SR : setR) x1 x2 (ST : setT),
    (∀ (ν : valuation), (∀ x, x ∈ SR → ν ⟦x⟧_t ∈ ST) → ν ⟦x1⟧ = ν ⟦x2⟧) → gd_sem (EqGen SR x1 x2) ST.
```

The only subtle point in this definition is that it is stated for tableaux, but corresponds exactly to the semantics of logical formulas. Due to the particular form of the latter, given a valuation ν assigning values to the x's we extend it by ν_e over the existentially quantified z's.

5.3 The Chase

We present the so-called *chase* a procedure for reasoning about dependencies and used to determine logical implication between sets of dependencies. More precisely, given a set D of dependencies and a dependency d over a given schema, the chase allows to decide whether $D \models d$. The intuition is that the chase starts assuming that the tableau part of d is satisfied and consists in applying all dependencies in D. If the conclusion of d is inferred then we have a proof that

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_1	b_1	c_2	d_3

tg_1

A	B	C	D
a_2	b_3	c_3	d_4
a_3	b_3	c_4	d_5
		$d_4 = d_5$	

eg_2

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_6	c_5	d_7

tg

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7

(i)

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_5	c_5	d_8

(ii)

A	B	C	D
a_4	b_4	c_5	d_6
a_4	b_5	c_6	d_7
a_4	b_5	c_5	d_7

(iii)

Fig. 3. Applying Dependencies

$D \models d$. The main result stated in the literature is that, any instance of the schema, satisfying d' and $chase(d, d')$ (the dependency obtained by applying d' to d), also satisfies d. All the magic resides in the definition of "applying a dependency". Assume that we want to prove that dependencies tg_1 and eg_2 in Figure 3 imply dependency tg, where we omit to tag the rows as a single relation name r is assumed. To do so, we apply them to instance (i) (indeed the tableau part of tg). More precisely, applying tg_1 to (i) consists in finding a mapping ν such that $\{\nu(a_1, b_1, c_1, d_1), \nu(a_1, b_2, c_2, d_2)\} \subseteq (i)$. For instance in this case we can choose, among other mappings, $\nu(a_1) = a_4, \nu(b_1) = b_5, \nu(c_1) = c_6, \nu(d_1) = d_7, \nu(b_2) = b_4, \nu(c_2) = c_5, \nu(d_2) = d_6$. Tuple $\nu(a_1, b_1, c_2, d_3)$ is then added to (i) yielding (ii). There is a subtlety: as d_3 appears only in the summary, d_3 is existentially quantified therefore $\nu(d_3)$ is a fresh variable (d_8). Then applying eg_2 to (ii) makes d_7 and d_8 equal in (iii). Again, as b_6 is existentially quantified in tg, it can be instantiated by b_5 and allows to conclude that since the tuple to be generated in tg occurs in (iii), tg is implied.

We *tried to formalize* what is very informally provided by textbooks with the following inference rules. Let d and d' be respectively $\forall \vec{x},\ \phi(\vec{x}) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z})$ and $\forall \vec{x'},\ \phi'(\vec{x'}) \Rightarrow \exists \vec{z'},\ \psi'(\vec{x'} \cup \vec{z'})$. For applying d' to d we first need to find a mapping ν such that $\nu(\phi'(\vec{x'}))$ seen as a set of atoms is a subset of $\phi(\vec{x})$. Depending on the form of ψ', we get

1. if $\psi' \equiv y'_1 = y'_2$ then let ρ be the renaming: $\{\nu(y'_2) \mapsto \nu(y'_1)\}$ and $chase(d, d')$ is $\forall \vec{x},\ \rho(\phi(\vec{x}) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z}))$.
2. if $\psi' \equiv r'(\vec{y'})$ then $chase(d, d')$ is $\forall \vec{x},\ \phi(\vec{x}) \wedge \nu(r'(\vec{y'})) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z})$.

However the above version is faulty due to variable's capture for $\nu(r'(\vec{y'}))$ by $\forall \vec{x}$ which naturally arose in the second case as shown by the following counter-example. Let d be $\forall y\ z,\ r(y, y, z) \Rightarrow r(y, y, y)$ and d' be $\forall x\ y,\ r(x, x, y) \Rightarrow \exists z,\ r(x, z, x)$. With mapping $\nu = \{x \mapsto y, y \mapsto z\}$, the above definition yields:

$$chase(d, d') \equiv \forall y\ z,\ r(y, y, z) \wedge r(y, z, y) \Rightarrow r(y, y, y).$$

Consider the instance $\mathcal{I} = \{(a, a, b), (a, c, a)\}$. We have $\mathcal{I} \models d'$, and $\mathcal{I} \models chase(d, d')$ since there is no μ such that $\mu(y, y, z) \in \mathcal{I} \wedge \mu(y, z, y) \in \mathcal{I}$. But $\mathcal{I} \not\models d$ as shown by $\mu_1 = \{y \mapsto a, z \mapsto b\}$ since $\mu_1(y, y, z) = (a, a, b) \in \mathcal{I}$ and $\mu_1(y, y, y) = (a, a, a) \notin \mathcal{I}$. This counter-example does not affect the essence of the theorem but emphasizes the fact that humans naturally perform α-conversion in order to avoid capture; therefore when defining the chase in Coq we had to seriously take this into account.

Since variables (in the gd's) are indexed by integers, in order to avoid captures, we generate fresh variables for renaming, starting from the maximum index of all variables in the constraints which is computed thanks to the function max_var_chase. Then, avoid_capture_trow max_n phi' psi' computes a renaming for the variables which are in psi' and not in phi'. The chase may yield three different results: the first one is when there is at least one ν producing a new constraint, the second captures the fact that no such mappings exist, then the third one corresponds to the fact that the current dependency tries to identify two distinct constants. There is one further subtle point to detail. Given a pair of dependencies, there may exist several mapping ν's, thus, in order to avoid the design of a lazy matching function, we chose to apply them at once. The first case applies an equality generating dependency EqGen SR x1 x2. It consists in iterating the replacement of ν x1 by ν x2 for all such ν's. The second case applies a tuple generating dependency TupleGen SR s. In that case we simply add all ν s to current tableau. The only point is to avoid capture for existential variables and also to avoid interference between the different mappings. This has the unfortunate consequence that the chase step given in [2] as well as soundness proofs are intricate. As the chase terminates only for a specific class of dependencies (the one with no existential quantifiers), we defined a kind of "for loop" in order to iterate the application of a set of dependencies over d a fixed number of times. At this point the algorithm stops with a (potentially) new dependency. If this dependency is trivial (*i.e.*, either of the form $\forall \vec{x},\ \phi(\vec{x}) \Rightarrow y = y$ or $\forall \vec{x},\ \phi(\vec{x}) \Rightarrow \exists \vec{z},\ \psi(\vec{x} \cup \vec{z})$ where there exists a substitution σ for z's such that $\psi(\vec{x} \cup \sigma(\vec{z}))$ is an atom of $\phi(\vec{x})$) then the initial set of dependencies implies d. Last the result that the chase procedure is sound is established by

```
Inductive res : Type := Res : gd → res | NoProgress | Fail.
...
Definition var_in_query x SR := match x with Tvar _ ⇒ x ∈ variables_gd SR | Tval _ ⇒ True end.
Lemma chase_is_sound :
 ∀ ST n D d d', chase n d D = Res d' →
   match d with TupleGen _ _ ⇒ True | EqGen SR1 x1 x2 ⇒ var_in_query x1 SR1 ⋀ var_in_query x2 SR1 end
    → (∀ gd, List.In gd D → gd_sem gd ST) → gd_sem d' ST → gd_sem d ST.
```

Doing the proof, the main subtle point was to avoid variables' capture through iteration. Again, it was during this proof step that we discovered that the textbooks were imprecise not to say faulty. The needed functions and technical lemmas are given in [2].

6 Conclusion, Lessons and Perspectives

This article provides a specification of the relational model, a first, *unavoidable*, step towards verifying relational database management systems with the Coq proof assistant. Our specification is the first that covers the named version of the relational model, both algebra and conjunctive queries, logical optimization for both languages, and, finally, dependencies (both functional and general). The whole development consists of 21,000 loc. It makes a clear distinction between specification and implementation, achieved thanks to a parametrization of the

data definition part of the model – attributes, tuples, relations and constraints – by modules whose interface is independent from the concrete implementation (e.g., Letouzey's finite sets). This allowed us to reach a very modular and reusable library. From the data definition point of view, our modeling is very close to the one found in textbooks as well as in real systems and is expressive and versatile enough to allow us to express the main algorithms and to prove the database theorems. In particular, we gave a completely certified version of the algorithm that translates an SPJ query into a conjunctive one, a proof of the main relational structural equivalences, a proof of the homomorphism theorem and based on this proof a certified version of the tableaux minimization algorithm, its extraction in OCaml and finally we modeled and certified Armstrong's system for functional dependencies and the chase procedure for which we also extracted an OCaml algorithm.

We learned several lessons both from the database and Coq sides. There are two different aspects in our work: one concerns modeling, the second is about proving properties and algorithms' correctness. On the side of proofs, the article does not bring very new insights except expliciting technical points such as freshness, unification in the translation, avoiding variables' capture. This is not new for Coq users or even the functional programming community. However it is worth precising that for the database theoreticians and practitioners as well. Such aspects are never mentioned in text books nor appear explicitly in implementations (usually written in C). The real challenge was to model. Our contribution, unlike, [8, 6], is almost complete. We were able to model all these various aspects because our very first choices for attributes, tuples were adequate. Such choices were not trivial nor immediate and neither [8] nor [5] made them hence they never reached the generality we achieved. Obviously once the right choices are done, the whole seems simple.

In a first version of our development, we heavily used dependent types and proofs in types. In particular, they expressed that tuples and queries were well-typed by construction. But, we experienced a lot of problems with type conversion in proofs. In all algorithms given in the article, it is *crucial* to check equality (or congruence). In Coq one can only check equality between two terms which belong to the *same* Type. With dependent types, there are two possibilities: either to use type conversion or John Major equality (fortunately we fall in the decidable case). Both are very cumbersome. Moreover, in order to debug we needed to run the algorithms with well-typed terms (*i.e.*, with hand-written proofs embedded in types). The benefits of our approach are three (i) with it, it is easier and lighter to write algorithms and perform case analysis in proofs (ii) it is closer to main stream programming languages in which real systems are encoded (iii) it precisely allows to locate where well-typedness is needed. Surprisingly, we discovered that types, in the usual sense, were not useful, rather, the notion of *well-sortedness* was indeed crucial. This is an a posteriori justification of the fact that in all theoretical books values range in a unique domain. Specifying the main algorithms and proving the "database theorems" for tableaux and the chase led us to thoroughly make explicit some notions or definitions which

were either unclear or at least very sloppy. For example, freshness or variables' capture are almost completely left aside in textbooks. However, such notions are *central to the correctness* of the results, as shown by our counter-example.

The long term goal of our work is to verify data intensive systems with the Coq proof assistant and the Why3 [4] program verification suite. We shall extend our work in several directions. First for the specification part we shall capture other data models such as JSon, XML etc to mechanize the semantics of languages such as JAQL or Pig. Then, we shall model all the relational normalization theory for logical schema design. Based on our library, another line of research will consist in verifying an SQL compiler and optimizer against our specification. SQL compilers not only transform queries into relational algebra (as far as possible) yielding an AST whose nodes are labeled by relational operators and leaves are base relations, but, they also choose the "best" access method to evaluate the query. To do so they rely on the fact that different algorithms for joins or selections do exist (sort-merge joins, hash-based, nested loops) and on different access paths to actual data (for example indexes). They generate so called query evaluation plans and choose, according to a cost model, the most efficient one. We plan to verify those algorithms using our formalization and Why3. We shall also handle transactions and concurrency control, updates and database triggers as well as security and privacy aspects.

Acknowledgements We are very grateful to Arthur Charguéraud for his helpful comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Benzaken, V., Contejean, E., Dumbrava, S.: A Relational Library `http://datacert.lri.fr/esop/html/Datacert.AdditionalMaterial.html` (2013)
3. Chlipala, A., Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: Hutton, G., Tolmach, A.P. (eds.) ICFP. pp. 79–90. ACM (2009)
4. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013)
5. Gonzalia, C.: Relations in Dependent Type Theory. Ph.D. thesis, Chalmers Göteborg University (2006)
6. Gonzalia, C.: Towards a formalisation of relational database theory in constructive type theory. In: Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS. LNCS, vol. 3051, pp. 137–148. Springer (2003)
7. Letouzey, P.: A library for finite sets
8. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: ACM Int. Conf. POPL (2010)
9. Ramakrishnan, R., Gehrke, J.: Database management systems (3. ed.). McGraw-Hill (2003)
10. The Coq Development Team: The Coq Proof Assistant Reference Manual (2010), http://coq.inria.fr, http://coq.inria.fr
11. Ullman, J.D.: Principles of Database Systems, 2nd Edition. Computer Science Press (1982)