Implementing Tabled Hypothetical Datalog

Fernando Sáenz-Pérez
Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid
Madrid, Spain
fernan@sip.ucm.es

***Abstract*—Hypothetical Datalog is based on an intuitionistic semantics rather than a classical logic semantics, and allows embedded implications in rule bodies. While the usual implication (i.e., the neck of a Horn clause) stands for inferencing facts, an embedded implication plays the role of assuming its premise for deriving its consequence. Although this topic has received considerable attention along time and nowadays is gaining renewed interest, there has not been a tabled implementation of hypothetical Datalog. We present here such a proposal including the formal background and its application to a goal-oriented tabled setting with negation, where non-monotonicity due to negation and implication is handled via stratification and contexts. In addition, we implement it in the deductive system DES, also providing support to duplicates and integrity constraints in the hypothetical framework.**

***Keywords*-Hypothetical Datalog; Tabling; Deductive Databases; DES;**

I. Introduction

Hypothetical queries, also known as "what-if" queries, are a common need in applications as OLAP [1], business intelligence [2], and e-commerce [3]. Whilst such systems and applications inherit from and build upon relational database approaches restricting the use of negation and recursion, earlier works on intuitionistic logic programming integrate such queries in the inference system. In particular, hypothetical Datalog [4] has been a proposal thoroughly studied from semantic and complexity point-of-views, allowing recursion, embedded implications and negation.

Our work follows the interpretation of [4] for which two kind of implications can be identified: The usual implication which is found as the neck of a logic clause, and the (hypothetical) implication which can be found in the body of a logic clause. Indeed, they receive different syntactic devices to be expressed: $\leftarrow$, and $\Leftarrow$, respectively, and are therefore differently interpreted. Whereas in the formula $L \leftarrow R$, the atom R is "executed" for proving L, in the formula $L \Leftarrow R$, the atom R is "assumed" to be true for proving L.

Since negation is also allowed to occur in clause bodies, stratification [5] is imposed as a syntactic restriction to programs including negation in order to avoid multiple models, a natural expectation from database users. Also, combining hypothetical goals and negation can also deal with paradoxes [4], as the introduction of an assumed fact may produce contradictions. We face this non-monotonic behaviour by using contexts as described along the paper.

Tabling provides overcomes to well-known problems of logic programming implementations and enhances efficiency. Systems implementing tabling memorize the deduced instances (answers) to goals (calls) in an answer table and call table, respectively, in order to reuse them and save further computations. Tabling has been applied to several logic programming systems (e.g., [6]) to deductive databases (e.g., [7]), and transaction logic [8]. However, though there have been some works regarding implementations [9], as far as we know there has not been an implementation of hypothetical Datalog based on tabling.

We introduce a formal framework partly based on [4] but with important differences. Here, we allow rules with embedded implications, with the intention to allow the user to assume both facts and rules to be hypothetically added to the current database. As an original motivation was to be able to assume fragments of a database, a first difference lies in that we allow to assume a set of rules and facts, instead of only a rule or fact, for proving a goal. Therefore, variables in assumed rules are encapsulated (i.e., they are not shared out of each assumed rule). And as a natural requirement, safety [5] is required for assumed facts and rules, ensuring finiteness of answers.

As an additional novel feature, we provide support for duplicates in the hypothetical setting. This enables to cope with problem formulations including multiple copies of facts, which in addition can be summarized with aggregates (as counting them). Duplicate sources can be both extensional and intensional and have not been considered for hypothetical Datalog up to now. Also, strong integrity constraints are supported in the hypothetical setting but, in contrast to works as [10], we do not allow to assume a fact or rule violating any integrity constraint, as usual in the relational setting.

We have implemented our approach to hypothetical Datalog in the deductive system DES [11], which can be downloaded from `des.sourceforge.net`, where motivating examples can also be found in the user manual.

This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, UCM-BSCH-GR58/08-910502, and S2009TIC-1465.

II. Formal Background

This section introduces some formal background to describe our approach to hypothetical Datalog, as an extension of function-free Horn logic following [4].

A. Syntax

The syntax of the logic is first order and includes a universe of constant symbols, a set of variables and a set of predicate symbols ($\mathcal{P}$). For concrete symbols, we write variables starting with upper-case letter and the rest of symbols starting with lower-case. Removing function symbols from the logic is a condition for finiteness of answers, a natural requirement of database users. As in Horn-logic, a rule has the form $A \leftarrow \phi$, where A is an atom and ϕ is a conjunction of goals. A goal, for a hypothetical system, can also take the form $\bigwedge R_i \Rightarrow G$, a construction known as an embedded implication, extending [4] by allowing the premise to be a conjunction of rules as an assumption. This means that for solving the conclusion G, rules R_i, together with the current database, will be used to deduce G. The following definition captures the syntax of our language, where $vars(T)$ is the set of variables occurring in T:

Definition 1 (Syntax of Rules):
$R := A \mid A \leftarrow G_1 \wedge \ldots \wedge G_n$
$G := A \mid \neg G \mid R_1 \wedge \ldots \wedge R_n \Rightarrow G$
where R and R_i stand for rules, G and G_i for goals, A for an atom, and $\bigcup vars(R_i) \cap vars(R) = \emptyset$, and the sets $vars(R_i)$ and $vars(G)$ are also disjoint.

Strong constraints are logical formulas of the form $\perp \leftarrow G_1 \wedge \ldots \wedge G_n$, that is, if the premise can be inferred, an inconsistent state is found ($\perp$ is dropped from now on). They are known in the database arena as integrity constraints and are used to specify conditions that database instances must hold. For instance, the constraint $\leftarrow employees(Name, Department) \wedge \neg departments(Department)$ represents a referential integrity constraint: It can not be the case of finding an employee associated to a non-existing department.

Definition 2 (Syntax of Constraints):
$C := \ \leftarrow G_1 \wedge \ldots \wedge G_n$
where C stands for a constraint, and G_i for goals.

Each constraint is given a fresh predicate with as many arguments as different variables there are in the constraint.

Definition 3 (Database): A database is a set of clauses possibly including both constraints and rules.

B. Safety

Safety is a condition for query answers to be ground, avoiding floundering [12]. This issue comes from allowing negation in goals. So, as an additional syntax requirement for our language, the rules R_i in a premise must be safe [5], because eventually they will be part of a database used for inferencing, and therefore soundness must be ensured. Next, definitions for rule and goal safety are given.

Definition 4 (Rule Safety): A rule $R := A \leftarrow G_1 \wedge \ldots \wedge G_n$, $n \geq 0$ is said to be safe (written as the property safe(R)) if:

- $vars(A) \subseteq vars(G)$, and
- For each negative goal G_i of either the form $\neg A'$ or $R_1 \wedge \ldots \wedge R_n \Rightarrow \neg A'$, $vars(A') \subseteq \bigcup vars(G_j)$, where G_j are the positive goals (i.e., with no $\neg$) in R

Note that this definition applies both to program rules and rules in the premises of embedded implications.

Definition 5 (Goal Safety): A goal G is safe if the rule $c \leftarrow G$ is safe, where c is an arbitrary, fresh predicate name.

So, while atoms in goals might be open, an atomic rule must be ground. Also, every call to a negated goal must be ground.

C. Predicate Dependency Graph and Stratification

Introducing negation in literals of body clauses introduces another issue: The possibility to have more than one minimal model [5]. Stratification is a syntactic condition on programs which ensures that only one minimal model can be assigned to a program. Predicates in the program are classified into strata so that negation does not occur through recursion. For building a stratification (i.e., a mapping between predicate symbols and natural numbers), a device called predicate dependency graph (PDG) is convenient. A PDG depicts the positive and negative dependencies between predicates.

Definition 6 (Dependencies): A predicate P *positively* (*negatively*, resp.) depends on Q if P is the predicate symbol of A in a rule (both a program rule and a rule in a premise) $A \leftarrow G_1 \wedge \ldots \wedge G_n$ and Q occurs either in some positive (negative, resp.) atom G_i or in G in an embedded implication $G_j \equiv R_1 \wedge \ldots \wedge R_n \Rightarrow G$.

Note that the implication $\leftarrow$ is the source for dependencies, whereas the embedded implication $\Rightarrow$ is not. However, all the non-atomic rules in the premise of $\Rightarrow$ are involved in adding dependencies. This fact is propagated to the construction of the dependency graph and the stratification for a program.

Definition 7 (Predicate Dependency Graph): A predicate dependency graph for a program Δ (written as $pdg(\Delta)$) is a pair $< N, A >$, where N is the set of predicate symbols in Δ and A is the set of arcs $P \leftarrow Q$ s.t. P positively depends on Q, and $R \overset{\neg}{\leftarrow} S$ such that R negatively depends on S.

Definition 8 (Stratification): A stratification of a program Δ (written as $str(\Delta)$) is a mapping $\mathcal{P} \rightarrow \mathbb{N}$ such that each $P \in \mathcal{P}$ is mapped to $i \in \mathbb{N}$ so that a predicate Q which positively (negatively, resp.) depends on P is mapped to a number $j \geq i$ ($j > i$, resp.) We also use $str(\Delta, P)$ to denote the stratum number corresponding to predicate P.

D. Stratified Inference

Following [4] we define a logical inference system for stratified intuitionistic logic programming, with the following main differences: Allowing duplicates, integrity constraints, premises with multiple rules, and enforcing encapsulation of variables in premises. Stratified inference requires an inference system for each stratum. Inference starts from the lower stratum and its derivations are inputs to the inference for the next stratum above. For a given stratum i, these derivations $\mathcal{E}$ are inference expressions which are constructed by the axioms derived in the stratum below and the rules defining the predicates belonging to stratum i. Input $\mathcal{E}$ is the empty set for the first stratum. In the following, we consider programs Δ which are both safe and stratifiable. Otherwise, inference cannot be applied. We use $pred(A)$ to denote the predicate symbol of atom A.

Duplicates would require working with bags in order to denote the multiple occurrences of the same atom. Instead, we resort to univocally identify each rule in a program and work with expressions tagged with such identifiers.

Definition 9 (Inference Expression): An inference expression for a program Δ is $\Delta \vdash \psi$, where ψ can be either an identified ground atom $id : \phi$, where id is a rule identifier and ϕ a ground atom, or $\bot$. The inference expression is positive iff ϕ is positive and negative iff ϕ is negative, and inconsistent otherwise.

Definition 10 (Inference System): Given a database Δ and a set of input inference expressions $\mathcal{E}$, the inference system associated to the s-th stratum is defined as follows, where $d_s(\mathcal{E})$ is a closure operator that denotes the set of inference expressions derivable in this system:
Axioms:

- $\Delta \vdash id : A$ is an axiom for each (ground) atomic formula $id : A$ in Δ, where $str(\Delta, pred(A)) = s$
- Each expression in $\mathcal{E}$ is an axiom.

Inference Rules:

- For any rule $A \leftarrow \phi_1 \wedge \ldots \wedge \phi_n$ with identifier id in Δ, where $str(\Delta, pred(A)) = s$ and for any ground substitution θ:
$$\frac{\Delta \vdash \phi_i\theta \text{ for each } i}{\Delta \vdash id : A\theta}$$
- For any goal ϕ:
$$\frac{\Delta \cup \{R_1, \ldots, R_n\} \vdash \phi}{\Delta \vdash R_1 \wedge \ldots \wedge R_n \Rightarrow \phi}$$
- For any constraint $\leftarrow \phi_1 \wedge \ldots \wedge \phi_n$ in Δ:
$$\frac{\Delta \vdash \phi_i\theta \text{ for each } i}{\Delta \vdash \bot}$$

Each rule in this inference system is read as: If the formulas above the line can be inferred, then those below the line can also be inferred.

Definition 11 (Inconsistent Set of Axioms): A set $\mathcal{E}$ is an inconsistent set of axioms if the expression $\Delta \vdash \bot$ is in $\mathcal{E}$, and consistent otherwise.

Like all Gentzen-style inference systems, this system is monotonic in the set of axioms, idempotent and inflationary. Let $\mathcal{S}$ denote the set of inference expressions for programs.

Lemma 1. The function $d_s : \mathcal{S} \rightarrow \mathcal{S}$ has the following properties:

- *Monotonicity: If $\mathcal{E} \subseteq \mathcal{F}$ then $d_s(\mathcal{E}) \subseteq d_s(\mathcal{F})$.*
- *Idempotence: $d_s(\mathcal{E}) = d_s(d_s(\mathcal{E}))$.*
- *Inflationaryness: $\mathcal{E} \subseteq d_s(\mathcal{E})$.* □

Negative information is deduced by applying the closed world assumption (CWA) [5] to inference expressions:

Definition 12 (Closed World Assumption of a Set of Inference Expressions): The closed world assumption of the set of inference expressions $\mathcal{E}$ (written as $cwa(\mathcal{E})$) is the union of $\mathcal{E}$ and the negative inference expression for $\Delta \vdash \phi$ such that $\Delta \vdash \phi \notin \mathcal{E}$.

The following definition captures the bottom-up construction of the semantics, stratum by stratum:

Definition 13 (Unified Stratified Semantics):

- $\mathcal{E}^0 = \emptyset$
- $\mathcal{E}^{s+1} = cwa(d_{s+1}(\mathcal{E}^s))$ for $s \geq 0$.

This procedure eventually terminates as the number of strata is finite.

Definition 14 (Consistent Database): A database Δ is consistent (written as the property cons(Δ)) if its unified stratified semantics $\mathcal{E}^{s+1}$ is a consistent set of axioms.

Solving a goal w.r.t. this semantics can be defined as:

Definition 15 (Meaning of a Goal): The meaning of a goal ϕ w.r.t. a set of axioms $\mathcal{E}$ (written as $solve(\phi, \mathcal{E})$) is defined as $solve(\phi, \mathcal{E}) = \{\Delta \vdash id : \psi \in \mathcal{E} \text{ such that } \phi\theta = \psi\}$ where ϕ is a goal, $solve$ returns a bag, and θ is a substitution.

Databases are incrementally built, clause-by-clause, starting from an empty database. Given a database (program rules and integrity constraints) Δ, a consistent database Δ_k is built from $\Delta_0 = \emptyset$ as: $\Delta_{i+1} = \Delta_i \cup \{c_i \in \Delta$ *such that safe*(c_i), *there exist* $str(\Delta_i \cup \{c_i\})$, *and cons*$(\Delta_i \cup \{c_i\})\}$

III. Hypothetical Tabling

Last section has introduced an operational semantics which builds the semantics of the whole database in a purely bottom-up fashion. However, for a system to be practical, it is much better to guide goal solving by queries. Here, we consider a top-down-driven, bottom-up fixpoint computation with tabling as implemented in DES, which follows the ideas found in [13]. In this section we assume databases which are safe and stratifiable.

A. *Tabling*

Tabled resolution for logic programs evaluates queries by memorizing calls and answers to goals. A call table ct stores the goal calls made along resolution as ϕ entries, and answers in an answer table at as $id : \psi$ entries, where id is a clause identifier and ψ is either a positive or negative ground atom.

The inference system defined in Section II (cf. Definition 10) includes the inference rule for hypothetical goals. That inference rule amounts to try to prove a goal in the context of the current database augmented with the premise of the implication. As a literal can be of the form $R_1 \wedge \ldots \wedge R_n \Rightarrow \phi$, where R_i are rules and ϕ a goal, the database Δ for which this hypothetical literal is to be proved must be augmented with $\{R_1, \ldots, R_n\}$. Deductions delivered in proving ϕ are only valid in the context of the augmented database, i.e., in the tabling tree constructed for ϕ. So, such deductions must be tagged in order to be only used in its context.

Filling answer and call tables is due to the memo function which proceeds by tabled SLDNF resolution as follows.

Definition 16 (Hypothetical Memo Function): Given a goal ϕ, a database Δ, a context identifier χ, a call table ct_0^0, and an answer table at_0^0, the memo function $memo(\phi, \Delta, \chi, ct_0^0, at_0^0)$ returns a pair $< ct, at >$ as specified as follows:

- If $\phi\theta \in ct_0^0$ then:
  - $ct = ct_0^0$
  - $at = at_0^0$
- Else:
  - For each program clause $A^j \leftarrow L_1^j \wedge \ldots \wedge L_{nj}^j$ in Δ identified by id_{A^j}, $j \geq 0$, $nj \geq 0$, such that $\phi = A^j\theta_0$.
    - For L_i^j either a positive or a negative literal, if $memo(L_i^j\theta_0 \cdots \theta_{i-1}, \Delta, \chi, ct_{i-1}^j \cup \phi, at_{i-1}^j)$ $=< ct_i^j, at_i^j >$, $id_{L_i^j}^\chi : L_i^j\theta_0 \cdots \theta_i \in cwa(at_i^j)$, then let $at^j = at_{nj}^j \cup id_{A^j}^\chi : A^j\theta_0 \cdots \theta_{nj}$ else $at^j = at_{nj}^j$
    - For L_i^j a hypothetical goal, if $memo(L_i^j\theta_0 \cdots \theta_{i-1}, \Delta \cup \{R_1, \ldots, R_n\}, \chi'$, $ct_{i-1}^j \cup id^\chi : \phi, at_{i-1}^j) =< ct_i^j, at_i^j >$, $id_{L_i^j}^\chi : L_i^j\theta_0 \cdots \theta_i \in cwa(at_i^j)$, where χ' is a context identifier for L_i^j, then let $at^j = at_{nj}^j$ $\cup\, id_{A^j}^\chi : A^j\theta_0 \cdots \theta_{nj}$ else $at^j = at_{nj}^j$
  - $ct = \bigcup ct_{nj}^j$
  - $at = \bigcup at^j$

Here, the closed world assumption of an answer table is defined analogously to the closed world assumption of a set of inference expressions:

Definition 17 (Closed World Assumption of an Answer Table): The closed world assumption of an answer table at (written as $cwa(at)$) in the context of a program is the union of at and $\epsilon^\chi : \neg A$ such that $id^\chi : A \notin at$ for any rule identifier id and context χ, where ϵ is a fixed, arbitrary identifier which does not occur in the program.

Filling the answer and call tables is done by strata by ensuring that the meaning of negated atoms which are required to prove other goals are already in the answer table. So, following the stratification for the program for a given goal ϕ, a goal dependency graph is computed, which is the subgraph of the PDG such that contains all reachable nodes from ϕ. Then, for each node p_i in the subgraph such that there is a negative arc coming out from p_i, an open goal ϕ_i is built with the same arity as p_i. Goals ϕ_i are ordered by $str(\Delta, \phi_i)$, so that lower-strata goals will be computed before upper-strata goals. The goal dependency graph is specified as follows:

Definition 18 (Goal Dependency Graph): A goal dependency graph (GDG) for a program Δ and goal ϕ (written as $gdg(\Delta, \phi)$) is a pair $< N, A >$, where $pdg(\Delta) =< N_\Delta, A_\Delta >$, with $N \subseteq N_\Delta$, $A \subseteq A_\Delta$, such that N are all the reachable nodes from ϕ in $pdg(\Delta)$ by traversing A arcs.

The stratified meaning of a program restricted to a goal is got by filling the tables as specified next:

Definition 19 (Stratified Hypothetical Meaning of a Program restricted to a Goal): Given a program Δ and a goal ϕ_{k+1}

$$< ct_i, at_i >= \bigsqcup_{n \geq 0} memo^n(\phi_i, \Delta, \chi) < ct_{i-1}, at_{i-1} >$$

where $gdg(\Delta, \phi) =< N, A >$, $p_i \in N$, $i \in \{1, \ldots, k\}$, $q \overset{\neg}{\leftarrow} p_i \in A$ for some q, $\phi_i = p_i(X_1, \ldots, X_{arity(p_i)})$, X_j fresh variables, $arity(p_i)$ is the arity of the predicate p_i, and indexes i are ordered such that $str(\Delta, \phi_i) \leq str(\Delta, \phi_{i+1})$.

Here, $\bigsqcup_{n \geq 0}$ represents the least upper bound of the successive applications of the function $memo$ as:

$$memo^1(\phi_i, \Delta, \chi) < ct_{i-1}, at_{i-1} >=< ct_{i-1}^1, at_{i-1}^1 >$$
$$memo^2(\phi_i, \Delta, \chi) < ct_{i-1}^1, at_{i-1}^1 >=< ct_{i-1}^2, at_{i-1}^2 >$$
$$\ldots$$
$$memo^j(\phi_i, \Delta, \chi) < ct_{i-1}^{j-1}, at_{i-1}^{j-1} >=< ct_{i-1}^j, at_{i-1}^j >$$

Then, for $i = k + 1$ we get the stratified meaning of the program restricted to ϕ_{k+1} in the answer table at_{k+1}. So, the meaning of a tabled goal is defined analogously to the meaning of a goal:

Definition 20 (Meaning of a Tabled Goal): The meaning of a tabled goal ϕ w.r.t. an answer table at in the context χ is defined as $tsolve(\phi, \chi, at) = \{\psi$ such that $id^\chi : \psi \in at$, and $\phi\theta = \psi\}$ where $tsolve$ returns a bag, and θ is a substitution.

B. An Example

Let's consider a train database Δ, where $city/1$ and $link/2$ are EDB (extensional database) predicates for representing city names and pairs of connected cities, resp. $travel/2$ is an IDB (intensional database) predicate for representing possible travels between cities as the transitive closure of $link$. IDB predicate $no\_travel/2$ represents pairs of cities such that it is not possible to travel between them. These IDB predicates are specified as the following database Δ:
$travel(X, Y) \leftarrow link(X, Y)$
$travel(X, Y) \leftarrow travel(X, Z) \wedge travel(Z, Y)$
$no\_travel(X, Y) \leftarrow city(X) \wedge city(Y) \wedge \neg travel(X, Y)$

The PDG for Δ is $< \{city,\ link,\ travel,\ no\_travel\}$, $\{travel \leftarrow link,\ travel \leftarrow travel,\ no\_travel \leftarrow city,$ $no\_travel \overset{-}{\leftarrow} travel\} >$. A stratification can be $\{(city, 1),$ $(link, 1),\ (travel, 1),\ (no\_travel, 2)\}$. For solving the goal $no\_travel(X, Y)$, and following definitions 19 and 20: $\phi_1 = travel(X, Y)$ and $\phi_2 = no\_travel(X, Y)$ are the goals for building $< ct_1, at_1 >$ and $< ct_2, at_2 >$, and the meaning of the goal ϕ_2 is $tsolve(\phi_2, \chi, at_2)$, where χ is the initial context. In this case, $pdg(\Delta)$ coincides with $gdg(\Delta, \phi_2)$. Considering also the rule $no\_travel(X, Y) \leftarrow city(a) \wedge city(b) \wedge city(c) \wedge link(a, b) \Rightarrow no\_travel(X, Y)$, the meaning of the former goal is now all the tuples $no\_travel(X, Y)$ such that X and Y can take values in the assumed cities (a, b and c) but the tuple $no\_travel(a, b)$ because of the assumed link. For this extended database, both PDG and GDG have been added with the edge $no\_travel \leftarrow no\_travel$.

IV. Implementing Hypothetical Tabling

This section describes a concrete implementation of the tabling mechanism as found in the deductive database system DES (based on [13]). Although it also supports unsafe rules and recursive rules as duplicate sources, in this description we restrict to safe rules and non-recursive rules as duplicate sources.

The answer table at is implemented with the dynamic predicate `et`/1 (following the nomenclature in [13] for extension tables) and the call table ct with the dynamic predicate `called`/1). Entries in `et` can be either positive (`A`) or negative (`not(A)`), where `A` is a ground atom. If a positive goal `G` is called, it is added to the call table and, if it succeeds, the (ground) fact $G\theta$ is added as an answer to `et`, where θ is a success substitution. A negative entry `not(A)` is added to `et` if a (ground) call to `A` cannot be proven. An entry is added to any of these tables if it does not occur already in the table. For supporting duplicates, each entry in `et` is tagged with an identifier of its source [14].

Along tabled resolution, a tabling tree is constructed analogously to [6]: On the one hand, the first time a goal `G` is called and there is no a more general goal subsuming `G` in ct, a new entry is added to this table. Then, first, results already present in at for `G` (from either a previous query or a previous fixpoint iteration, see `solve_star` in [14], following ET^* [13]) are returned upon backtracking. And, second, program rules are used to derive new results. On the other hand, if the goal (or a more general goal) has been already called, then simply return the results in at upon backtracking. Each time a goal `G` is called, resolution reuses answers already in the answer table, if any.

The function $memo$ is implemented by following the ET algorithm in [13] with the predicate `memo(+G,+D,-Id)`, where its arguments are, respectively, the input goal, duplicate elimination flag, and output goal identifer (see [14] for details). In contrast to Definition 16, the call and answer tables are implemented as dynamic predicates instead of predicate arguments. Also, instead of traversing all the program rules for each call, each program rule is traversed by backtracking. Then, there is at most only one answer per call and goal, with the answer substitution due to either: 1) matching the goal `G` with an entry in at (performed by the predicate `et_lookup`, or 2) solving the goal with the predicate `solve_goal`. This last predicate selects a matching program rule with `G` via backtracking and solves each literal in its body as calls to the the predicate `solve`. If the argument of `solve` is an atom, a straight call to `memo` is done. If the argument of `solve_goal` is a negated goal, the it calls to `solve`, succeeding if this call fails and vice versa. For the concrete implementation, `solve_goal` also computes built-ins as aggregates, and `solve` computes conjunctions and metapredicates.

The context parameter is added to the predicates `memo`, `called`, `et_lookup`, `solve_goal`, and `solve`, and the dynamic predicates `et` and `ct`. For the last two, the context identifier tags each entry as corresponding to the computation of a particular database: Either the loaded database (following Subsection **??**) or an augmented database due to embedded implications. So, `et_lookup` and `called` look for entries corresponding to the context which is being computed. In addition, the dynamic predicate `datalog`, which holds program rules, is also added with this context identifier. This makes possible to retrieve program rules for a given context, omitting rules of subsequent contexts.

As solving an implication amounts to add to the current database the rules in the premise, iterative applications of the function $memo$ might lead to adding the same hypothetical rules for the same context. To avoid this, we tag each context with a dynamic predicate (`hyp_program_asserted`) so that the corresponding premise is added only once. Then, the first call to solving an implication adds the premise to the current database, tags the context, and solves the goal (consequent) by stratified solving. A subsequent call does not add the premise, and stratified solving is only required if the current call is not subsumed by a previous one (as more tuples might be delivered from lower strata). If the call is subsumed by a previous call, then only a call to `solve` is needed. An implication is processed by the predicate `solve` with a straight call to `solve_implication`, which is depicted next:

```
solve_implication(L => R,CId,Rule,Ids) :-
  dlrule_id(Rule,RId),
  assert_hyp_program(RId,L,R,_G,CId,NCId),
  solve_stratified(R,NCId), !,
  solve(R,NCId,Rule,Ids).
solve_implication(_L => R,CId,Rule,Ids) :-
  dlrule_id(Rule,RId),
  is_hyp_program_asserted(RId,G,CId,NCId),
  (my_subsumes(G,R)
  ;
   (G==R -> true ;
    assertz(
      hyp_program_asserted(RId,R,CId,NCId))
          ),
   solve_stratified(R,NCId)
  ), !,
  solve(R,NCId,Rule,Ids).
```

Here, `Rule` is the rule which `L => R` belongs to, and `Ids` the duplicate identifier [14]. The predicate `dlrule_id` returns in its second argument the rule identifier for its first input argument. This rule identifier is prepended to the current context identifier (the list `CId`) to build the new context identifier (`NCId`). The predicate `assert_hyp_program` succeeds if the current call is the first one (the new context has not been tagged already), tagging the new context. The predicate `is_hyp_program_asserted` looks by backtracking for the unifiable previous calls to the current goal. If the current call is more general than a previous one, then this call is added to the context tag. Whereas only the entries in the call and answer tables for a given context are considered in solving a goal (cf. `et_lookup` and `called` in predicate `memo`), all the assumed rules up to the current context are taken into account.

Asserting a rule of the premise can either succeed or not, depending on whether it fulfills strong constraints. So, asserting a new rule follows the same route than asserting a regular rule, i.e., checking its consistency w.r.t. such constraints. Those rules that does not fulfill some constraint are rejected and the user notified, but computation progresses.

V. Conclusions and Future Work

We have proposed a novel implementation of an intuitionistic semantics based on tabling. This includes both duplicates and strong constraints as new features over existing proposals, also allowing to assume a set of rules. We have described the formal framework and its implementation for the first time, dealing with non-monotonicity due to negation and implication via stratification and contexts. This framework can be augmented by including a declarative semantics, and soundness and completeness results. One of the motivations behind our proposal is to allow assumptions in SQL, as already done in DES, which supported SQL hypothetical queries. Now, it can be extended to define SQL hypothetical views by translating SQL views to Datalog predicates. Also, solving an embedded implication as presented requires to recompute from scratch the given goal for all the involved strata. However, some computations in previous contexts already stored in the answer table can be reused in subsequent contexts. Identifying and reusing such entries is also subject of further work.

References

[1] G. Zhou, H. Chen, and Y. Zhang, “Hypothetical queries on multidimensional dataset,” in *Proc. of BIFE*, S. Wang, L. Yu, F. Wen, S. He, Y. Fang, and K. K. Lai, Eds. IEEE, 2009, pp. 539–543.

[2] M. Golfarelli and S. Rizzi, “What-if simulation modeling in business intelligence,” *IJDWM*, vol. 5, no. 4, pp. 24–43, 2009.

[3] Y. Zhang, H. Chen, H. Sheng, and Z. Wu, “Applying hypothetical queries to e-commerce systems to support reservation and personal preferences,” in *Proc. of IDEAS ’07*. IEEE, 2007, pp. 46–53.

[4] A. J. Bonner and L. T. McCarty, “Adding negation-as-failure to intuitionistic logic programming,” in *Proc. of the North American Conference on Logic Programming*, E. L. Lusk and R. A. Overbeek, Eds. The MIT Press, 1990, pp. 681–703.

[5] J. D. Ullman, *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press, 1988.

[6] T. Swift and D. S. Warren, “XSB: Extending Prolog with Tabled Logic Programming,” *TPLP*, vol. 12, no. 1-2, pp. 157–187, 2012.

[7] Y. Shen, L. Yuan, and J. You, “SLT-Resolution for the Well-Founded Semantics,” *JAR*, vol. 28, pp. 53–97, 2002.

[8] P. Fodor and M. Kifer, “Tabling for transaction logic,” in *Proc. of PPDP*. New York, NY, USA: ACM, 2010, pp. 199–208.

[9] L. Vieille, P. Bayer, V. Küchenhoff, A. Lefebvre, and R. Manthey, “The EKS-V1 System,” in *LPAR*, ser. LNCS, A. Voronkov, Ed., vol. 624. Springer, 1992, pp. 504–506.

[10] H. Christiansen and T. Andreasen, “A Practical Approach to Hypothetical Database Queries,” in *Transactions and Change in Logic Databases*, ser. LNCS, B. Freitag, H. Decker, M. Kifer, and A. Voronkov, Eds., vol. 1472. Springer, 1998, pp. 340–355.

[11] F. Sáenz-Pérez, “DES: A Deductive Database System,” *ENTCS*, vol. 271, pp. 63–78, March 2011.

[12] H. Decker, “The Range Form of Databases and Queries or: How to Avoid Floundering,” in *5. Ãsterreichische Artificial-Intelligence-Tagung*, ser. Informatik-Fachberichte, J. Retti and K. Leidlmair, Eds. Springer Berlin Heidelberg, 1989, vol. 208, pp. 114–123.

[13] S. W. Dietrich, “Extension tables: Memo relations in logic programming,” in *IEEE Symp. on Logic Programming*, 1987, pp. 264–272.

[14] F. Sáenz-Pérez, “Tabling with support for relational features in a deductive database,” *Electronic Communications of the EASST*, vol. 55, pp. 1 – 16, May 2013.