Implementing Tabled Hypothetical Datalog

Fernando Sáenz-Pérez*

Grupo de programación declarativa (GPD),
Dept. Ingeniería del Software e Inteligencia Artificial,
Universidad Complutense de Madrid, Spain
fernan@sip.ucm.es

Abstract. Hypothetical Datalog is an extension of Horn-clause logic programming, based on an intuitionistic semantics rather than a classical logic semantics. It allows implications in rule bodies, which are known as embedded implications. 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. This allows to pose hypothetical queries and manage “what-if” scenarios, where hypothetical facts which are not part of the database can be used throughout the inference. This topic has received considerable attention along time and nowadays is gaining renewed interest in different fields. However, as far as we know, there has not been a tabled implementation of hypothetical Datalog allowing both embedded implications and negation, where non-monotonicity due to negation and implication is handled via stratification and contexts. In addition, we develop it in the context of the deductive system DES, also providing support to duplicates and integrity constraints in the hypothetical framework. We present here such an implementation in the DES deductive system, allowing both hypothetical queries and clauses in a goal-oriented tabled setting, following the memoization techniques which find their roots on dynamic programming. Non-monotonicity due to negation and implication is handled via stratification and contexts.

Keywords: Hypothetical Datalog, Tabling, Deductive Databases, DES

1 Introduction

Hypothetical queries are a common need in several scenarios, related mainly with business intelligence applications and the like. They are also known as "what-if" queries and help managers to take decisions on scenarios which are somewhat changed with respect to a current state. Such queries are used, for instance, for deciding which resources must be added, changed or removed to optimize some criterion (i.e., a cost function, a notion well related to optimization technologies).

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, UCM-BSCH-GR58/08-910502, and S2009TIC-1465.
Current applications include OLAP environments [1, 33], business intelligence [12], and e-commerce [32, 31]. Even, major vendors of relational databases include approaches to hypothetical queries, as for instance the model clause in Oracle SQL data warehousing [16].

Whilst such systems and applications inherit from and build upon database approaches, earlier works on logic programming fully integrate hypothetical queries in the inference system. These approaches [13–15, 10] fit into intuitionistic logic programming, an extension of logic programming including both embedded implications and negation. Hypothetical Datalog [3, 4] has been a proposal thoroughly studied from semantic and complexity point-of-views.

We focus in this work on such approaches targeted to a Datalog deductive system, and in particular the interpretation of [2] 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 \( A \leftarrow B \), the atom \( B \) is "executed" for proving \( A \), in the formula \( A \leftarrow B \), the atom \( A \) is "assumed" to be true for proving \( A \).

As usual, since negation is also allowed to occur in clause bodies, stratification [28] is imposed as a syntactic restriction to programs in order to avoid multiple models. This restriction avoids programs containing rules as \( A \leftarrow B \) and \( B \leftarrow \neg A \) altogether, because more than a minimal model exists (i.e., \{A\} and \{B\}). Although other approaches omit this restriction [11, 29], it can be argued that database users understand this restriction as natural because answers meet intended semantics and only a model is returned, instead of a disjunction of possible models.

Though there have been some works regarding implementations [30], as far as we know there has not been an implementation of hypothetical Datalog based on tabling. Tabling has been a successful technique applied to different fields (several current logic programming systems, such as B-Prolog [34], Ciao [6], Mercury [25], XSB [26], and Yap Prolog [18], to name just a few) and in particular to deductive databases (e.g., [23, 17, 22, 19]). Tabling faces some well-known problems of logic programming implementations: unsoundness, repeated computations, and termination, providing some overcomes, and it has been useful in particular for efficient systems (both general-purpose languages and deductive systems). Systems implementing tabling memoize the deduced instances (answers) to goals (calls) in an answer table and call table, respectively. When a goal is to be solved, it is solved by using the contents of the answer table if it is subsumed by an entry in the call table. If there is not such an entry, the goal is added to the call table, and its answers deduced by using its program rules are also added to the answer table.

The current work therefore presents a novel tabling implementation of an approach to hypothetical Datalog, partly based on [2] but with important differences. Here, we allow rules with embedded implications and stratified negation, with the intention to allow the user to assume both facts and rules to be hypo-
Implementing Tabled Hypothetical Datalog

...theretically added to the current database. A first difference lies in that we allow to assume a set of rules and facts, instead of only a rule or fact. The original motivation was therefore to be able to add fragments of a database (both extensionally and intensionally) in order to solve such hypothetical queries. Therefore, variables in assumed rules are encapsulated (i.e., they are not shared out of each assumed rule). And as a natural requirement, safety [28] must be ensured for assumed facts and rules, that is, if we augment a database with a given fact or rule, we must ensure finiteness of answers.

As an additional novel feature, we develop our approach providing support to 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 [5] and [9], we do not allow to assume a fact or rule violating any integrity constraint. This follows the same criterion already present for the non-assumed part of the database, as usual in the relational setting. We have implemented such an approach to hypothetical Datalog in the current deductive system DES [21] with a first attempt in version 3.2 without dealing with negation, and the current proposal as implemented in a forthcoming release.

Organization of this paper proceeds as follows: Section 2 introduces concrete syntax and presents some motivating examples, including uses of negation, recursion, integrity constraints and duplicates in the hypothetical setting. Next, some formal background is presented in Section 3. The setting to implement this background is described in Section 4 as part of the deductive system DES. Section 5 shows how to modify the existing tabling implementation to support hypothetical inferencing. Finally, Section 6 concludes and lists some future work.

2 Getting Started

This section first introduces the concrete syntax we have chosen for the system implementing our proposal and then shows different examples to highlight the new possibilities derived for assumptions, negation, recursion, duplicates, and integrity constraints.

2.1 Concrete Syntax

The syntax of a hypothetical query is as follows:

\[ \text{Rule}_1 \lor \ldots \lor \text{Rule}_N \Rightarrow \text{Goal} \]

which means that, assuming that the current database is augmented with the rules \( \text{Rule}_1, \ldots, \text{Rule}_N \) (which follow Prolog standard syntax), then \( \text{Goal} \) is computed with respect to the current database which is augmented with these rules, which must be safe [28]. Such query is also understood as a literal in the context of a rule, so that any rule can contain hypothetical goals, as in \( a :- b \Rightarrow c \).
turn, any Rule \( i \) can contain hypothetical goals. Variables in Rule \( i \) are encapsulated (i.e., they are neither shared with other rules nor with the goal, even when they might have the same name). Moreover, a hypothetical literal does neither share variables with other literals nor the head of the rule in which it occurs.

### 2.2 A University Example

Borrowing an example from [3], we consider an extended and adapted rule-based system for describing university policy: student(S) means that S is a student, course(C) that C is a course, take(S,C) that student S takes course C, and grad(S) that S is eligible for graduation. The extensional database can contain facts as:

| student(adam) | course(eng) | take(adam,eng) |
| student(bob) | course(his) | take(pete,his) |
| student(pete) | course(lp) | take(pete,eng) |
| student(scott) | course(his) | take(scott,his) |
| student(tony) | course(lp) | take(tony,his) |

The intensional database can contain rules as:

\[
\text{grad}(S) :- \text{take}(S,\text{his}), \text{take}(S,\text{eng}).
\]

A regular query for students that would be eligible to graduate is:

DES> grad(S)
{ grad(pete) }
Info: 1 tuple computed.

A first hypothetical query for this database asks "If Tony took eng, would he be eligible to graduate?", which can be queried with:

DES> take(tony,eng) => grad(tony)
Info: Processing:
answer :-
    take(tony,eng)=>grad(tony).
{ answer }
Info: 1 tuple computed.

Here, the query has been automatically rewritten as a temporary view with name answer, i.e., a view which is added to the database and eventually removed. This allows non atomic goals to be solved, as it is the case for an implication. The outcome of the query is the result of the goal answer, which can be proved as assuming that premise allows to deduce the consequent.

Also, more than one assumption can be simultaneously stated, as in: "If Tony took eng, and Adam took his, what are the students that are eligible to graduate?"
Implementing Tabled Hypothetical Datalog

\[
\text{DES}> \text{take(tony, eng)} \lor \text{take(adam, his)} \Rightarrow \text{grad}(S)
\]

Info: Processing:
\[
\text{answer}(S) :-
\text{take(tony, eng)} \lor \text{take(adam, his)} \Rightarrow \text{grad}(S).
\]
\{
\text{answer(adam)},
\text{answer(pete)},
\text{answer(tony)}
\}
Info: 3 tuples computed.

Another query is "Which are the students which would be eligible to graduate if his and lp were enough to get it?":

\[
\text{DES}> (\text{grad}(S) :- \text{take}(S, \text{his}), \text{take}(S, \text{lp})) \Rightarrow \text{grad}(S)
\]

Info: Processing:
\[
\text{answer}(S) :-
(\text{grad}(S) :- \text{take}(S, \text{his}), \text{take}(S, \text{lp})) \Rightarrow \text{grad}(S).
\]
\{
\text{answer(pete)},
\text{answer(scott)}
\}
Info: 2 tuples computed.

Note that, although \( S \) occurs in both the premise and the conclusion of \( \Rightarrow \), they are not actually shared, and they simply act as different variables.

Considering also information about course prerequisites as:

\[
\text{pre(eng, lp)}.
\text{pre(hist, eng)}.
\text{pre(Pre, Post)} :-
\text{pre(Pre, X)},
\text{pre(X, Post)}.
\]

one might wonder whether adding a new prerequisite implies a cycle (so that students cannot fulfil prerequisites at all for the courses in a cycle):

\[
\text{DES}> \text{pre(lp, hist)} \Rightarrow \text{pre(X, X)}
\]

Info: Processing:
\[
\text{answer}(X) :-
\text{pre(lp, hist)} \Rightarrow \text{pre(X, X)}.
\]
\{
\text{answer(eng)},
\text{answer(hist)},
\text{answer(lp)}
\}
Info: 3 tuples computed.

The answer includes those nodes in the graph that are in a cycle.
2.3 Hypothetical Queries and Integrity Constraints

Assumptions can be used in combination with any of the features of DES; in particular, integrity constraints. Following the previous example, you can even express it with the aid of integrity constraints. Avoiding cycles can be forced by:

\[ \text{DES} > \text{:-pre}(X,X) \]

Then, if you want to list prerequisites assuming \text{pre}(lp,hist) as before:

\[ \text{DES} > \text{pre}(lp,hist)\Rightarrow\text{pre}(X,Y) \]

\text{Info: Processing:}
\[
\text{answer}(X,Y) :- \text{pre}(lp,hist)\Rightarrow\text{pre}(X,Y).
\]

\text{Error: Integrity constraint violation.}
\[
\text{ic}(X) :- \text{pre}(X,X).
\]

\text{Offending values in database: \{ic(lp),ic(eng),ic(hist)\}}

\text{Info: The following rule cannot be assumed:}
\[
\text{pre}(lp,hist).
\]

\{ 
\text{answer(eng,lp)}, 
\text{answer(hist,eng)}, 
\text{answer(hist,lp)} 
\}

\text{Info: 3 tuples computed.}

So, the system informs that there is an inconsistency when trying to assert such offending fact (\text{pre}(lp,hist)), which makes prerequisites to form a cycle (as shown in the offending value list \{ic(lp),ic(eng),ic(hist)\}). The system informs about the rules that cannot be assumed but continues its processing. This is also useful to know the result for the admissible assumptions. Note that, in general, offending facts can be a subset of the meaning of an assumed rule in the context of the current database. To illustrate this, let’s consider the following program for tossing a coin:

\% Tails win:
\::= win, heads.

\text{win} ::= \text{heads} ; \text{tails}.

Predicate \text{win} states that one wins if either heads or tails are got, and the constraint states that you have to get tails to win. Here, the semicolon ";" denotes disjunction as in Prolog syntax. Then, the following hypothetical goal states whether assuming heads or tails leads to win.

\[ \text{DES} > \text{heads} \lor \text{tails} \Rightarrow \text{win} \]

\text{Info: Processing:}
\[
\text{answer} :- \text{heads}\lor\text{tails}\Rightarrow\text{win}.
\]

\text{Error: Integrity constraint violation.}
\[
\text{ic} :-
\]
Implementing Tabled Hypothetical Datalog

As informed, heads cannot be assumed in order to win.

2.4 Hypothetical Queries and Duplicates

Duplicates can also be used along computations involving assumptions. Let’s consider a variation of the classical Nim game, known as the subtraction game. Here, there is only one heap from which a player can take one or two tokens in his turn. A player wins if there is only one token in other player’s turn (misère game). This can be formulated with the next program:

```
win_nim :-
    take => one_left.
win_nim :-
    take\take => one_left.
win_nim :-
    take => enough, win_nim.
win_nim :-
    take\take => enough, win_nim.

one_left :-
    total(N),
    count(take,C),
    N-C=1.

enough :-
    total(N),
    count(take,C),
    C>0.
```

The predicate `win_nim` states that one wins if it can take one or two tokens at a time and there is already one left for the other player. Otherwise, if there are enough tokens (after taking one or two) to continue playing, then let’s see if one can win. Each occurrence of `take` in the left hand side of `=>` is an assumed fact that can be counted if duplicates are enabled (otherwise, the counting will be 0 – if there is no one – or 1 – if there is one or more, as duplicates are discarded). So, the predicate `one_left` determines whether there is exactly one token left, and `enough` determines if there is one token left at least. The predicate `total` states the total number of tokens which are available for a game. For instance, if we had 4 tokens and was my turn, I cannot ensure to win because the other player can take only one token and, then, in my next turn, should I take either one or two, I’ll lose.
2.5 Hypothetical Queries and Negation

Implication can also be used in conjunction with negation. Let’s consider the following example, which states flight links and flight travels, where \( \text{flight}(X,Y) \) states the link from \( X \) to \( Y \), and \( \text{flight_travel}(X,Y) \) represents possible flight travels where the involved airports are not closed:

\[
\text{flight_travel}(X,Y) \leftarrow \\
\text{flight}(X,Y), \\
\neg \text{closed}(X), \\
\neg \text{closed}(Y).
\]

\[
\text{flight_travel}(X,Y) \leftarrow \\
\text{flight_travel}(X,Z), \\
\text{flight_travel}(Z,Y).
\]

\[
\begin{align*}
\text{flight}(a,b). \\
\text{flight}(b,c). \\
\text{flight}(c,d).
\end{align*}
\]

A regular query for consulting possible travels is:

\[
\text{DES}\> \text{flight_travel}(X,Y) \\
\{ \\
\text{flight_travel}(a,b), \\
\text{flight_travel}(a,c), \\
\text{flight_travel}(a,d), \\
\text{flight_travel}(b,c), \\
\text{flight_travel}(b,d), \\
\text{flight_travel}(c,d)
\}
\]

Info: 6 tuples computed.

Assuming that airport \( b \) is closed, we ask for the possible travels with this assumption:

\[
\text{DES}\> \text{closed}(b) \Rightarrow \text{flight_travel}(X,Y)
\]

Info: Processing:
answer(X,Y) :-
   closed(b)=>flight_travel(X,Y).
{
   answer(c,d)
}
Info: 1 tuple computed.

where negated calls to closed/1 occur in the first rule of flight_travel/2.

We can also ask for the opposite: "Which are the flight travels which are not possible for that assumption?":

DES> flight_travel(X,Y),(closed(b)=>not(flight_travel(X,Y)))
Info: Processing:
   answer(X,Y) :-
      flight_travel(X,Y),
      closed(b)=>not(flight_travel(X,Y)).
{
   answer(a,b),
   answer(a,c),
   answer(a,d),
   answer(b,c),
   answer(b,d)
}
Info: 5 tuples computed.

Note that, first, we ask for all the possible flights with the first goal flight_travel(X,Y) (which indeed has the very same meaning as the first regular query of this subsection) and, then, we restrict to those flights which are not possible under the assumption. The first goal is needed for the query to be safe, i.e., recall that Datalog with negation is not constructive (variables in the negated goal are not instantiated unless their values are already provided by a positive goal), and answers must be ground.

3 Formal Background

This section introduces some formal background to describe the approach to hypothetical Datalog we are considering, as an extension of function-free Horn logic following [2]. We present the syntax of the language, safety conditions, the notion of stratifiable program, and an operational semantics.

3.1 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 (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 ← φ, where A is an atom and φ is a conjunction of goals. But, in addition, since we consider a hypothetical system, a goal can also take
the form $G \leftarrow R$, a construction known as an embedded implication, where the premise $R$ represents an assumption and takes the form of a rule. Moreover, we extend [2] by allowing the premise to be a disjunction of rules $\bigvee R_i$ as an assumption. This means that for solving the conclusion $G$, rules $R_i$ will be used to deduce $G$ together with the current database. As an embedded implication behaves different from a regular implication [2], it receives a different syntax symbol: $\Rightarrow$. 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).**

\[
R := A | A \leftarrow G_1 \land \ldots \land G_n \\
G := A | \neg G | R_1 \lor \ldots \lor R_n \Rightarrow G
\]

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 $\vars(R_i)$ are also disjoint.

Examples of negative goals are $\neg A$, $\neg(B \Rightarrow A)$, and $B \Rightarrow \neg A$, but $\neg B \Rightarrow A$ is not allowed.

Strong constraints are logical formulas of the form $\bot \leftarrow G_1 \land \ldots \land G_n$, that is, if the premise can be inferred, an inconsistent state if found. They are known in the database arena as integrity constraints and are used to specify conditions that database instances must hold. For instance, let’s consider the constraint $\leftarrow \text{employees}(\text{Name}, \text{Department}) \land \neg \text{departments}(\text{Department})$ in the context of a database for which employees maps a employee name with its department, and departments includes the possible departments. Such constraint represents a referential integrity constraint: It can not be the case of finding a employee associated to a non-existing department.

**Definition 2 (Syntax of Constraints).**

\[
C := \leftarrow G_1 \land \ldots \land G_n
\]

where $C$ stands for a constraint, and $G_i$ for goals.

Each constraints is given a name and can be represented by a predicate with that name with as many arguments as different variables there are in the constraint. Note that constraints can also include embedded implications.

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

### 3.2 Safety

Safety is a condition for query answers to be ground, avoiding floundering [7] and therefore unsoundness. This issue comes from allowing negation to the premise of a rule implication. So, as an additional syntax requirement for our language, the rules $R_i$ in a premise must be safe [28], 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). For a rule \( R := A \leftarrow G_1 \land \ldots \land G_n, n \geq 0 \) to be safe (written as the property \( \text{safe}(R) \)):

- \( \text{vars}(A) \subseteq \text{vars}(G) \), and
- For each negative goal \( G_i \) of either the form \( \neg A \) or \( R_1 \lor \ldots \lor R_n \Rightarrow \neg A \), \( \text{vars}(A) \subseteq \bigcup \text{vars}(G_j) \), where \( G_j \) are the positive goals (i.e., they do not contain \( \neg \)) in \( R \).

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

In particular, this means that while atoms in goals might be open, an atomic rule must be ground. Also, a call to a negated goal must be ground.

3.3 Predicate Dependency Graph and Stratification

Introducing negation in literals of body clauses introduces another issue: the possibility to have more than one minimal model [28]. 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, a situation illustrated already in the introduction. For building a stratification (i.e., a mapping between predicate symbols and natural numbers), a device called predicate dependency graph (PDG) is usually 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 \( A \leftarrow G_1 \land \ldots \land G_n \) and \( Q \) occurs either in some positive (negative, resp.) atom \( G_i \) or in \( G \) in an embedded implication \( G_j \equiv R_1 \lor \ldots \lor R_n \Rightarrow G \).

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 \( \Delta \) (written as \( \text{pdg}(\Delta) \)) is a pair \( < N, A > \), where \( N \) is the set of predicate symbols in \( \Delta \) and \( A \) is the set of arcs \( P \leftarrow Q \) such that \( P \) positively depends on \( Q \), and \( R \leftarrow S \) such that \( R \) negatively depends on \( S \).

Definition 8 (Stratification). A stratification of a program \( \Delta \) (written as \( \text{str}(\Delta) \)) is a mapping \( P \to \mathbb{N} \) 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 \( j > i \) (\( j \geq i \), resp.). We also use \( \text{str}(\Delta, P) \) to denote the stratum number corresponding to predicate \( P \).
3.4 Stratified Inference

Following [2] we define a logical inference system for stratified intuitionistic logic programming. The main differences we introduce with respect to that work are: duplicates, integrity constraints, the allowed premises in embedded implications, and 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 \( A \) 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 \( A \) is the empty set for the first stratum. In the following, we consider programs \( \Delta \) which are both safe and stratifiable. Otherwise, inference cannot be applied. We use \( \text{pred}(A) \) to denote the predicate symbol of atom \( A \).

Duplicates would require working with bags (multisets) 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 \( \Delta \) is \( \Delta \vdash \psi \), where \( \psi \) can be either an identified ground atom \( \text{id} : \phi \), where \( \text{id} \) is a rule identifier, or \( \perp \). The inference expression is positive iff \( \phi \) is positive and negative iff \( \phi \) is negative, and inconsistent otherwise.

**Definition 10 (Inference System).** Given a database \( \Delta \) and a set of input inference expressions \( A \), the inference system associated to the \( s \)-th stratum is defined as follows, where \( d_s(A) \) is a closure operator that denotes the set of inference expressions derivable in this system:

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

**Inference Rules:**
- For any rule \( A \leftarrow \phi_1 \land \ldots \land \phi_n \) with identifier \( \text{id} \) in \( \Delta \), where \( \text{str}(\Delta, \text{pred}(A)) = s \) and for any ground substitution \( \theta \):
  \[
  \frac{
  \Delta \vdash \phi_i \theta \quad \text{for each } i
  }{
  \Delta \vdash \text{id} : A\theta
  }
  \]
- For any goal \( \phi \):
  \[
  \frac{
  \Delta \cup \{R_1, \ldots, R_n\} \vdash \phi
  }{
  \Delta \vdash R_1 \lor \ldots \lor R_n \Rightarrow \phi
  }
  \]
For any constraint \( \leftarrow \phi_1 \land \ldots \land \phi_n \) in \( \Delta \):

\[
\Delta \vdash \phi_i \theta \quad \text{for each } i
\]

\[
\Delta \vdash \bot
\]

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

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

**Lemma 1.** The function \( d_s : \mathcal{E} \rightarrow \mathcal{E} \) has the following properties:

- **Monotonicity:** If \( \mathcal{A} \subseteq \mathcal{B} \) then \( d_s(\mathcal{A}) \subseteq d_s(\mathcal{B}) \).
- **Idempotence:** \( d_s(\mathcal{A}) = d_s(d_s(\mathcal{A})) \).
- **Inflationaryness:** \( \mathcal{A} \subseteq d_s(\mathcal{A}) \).

Negative information is deduced by applying the closed world assumption (CWA) [28] to inference expressions:

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

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

**Definition 13 (Unified Stratified Semantics).**

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

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

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

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

**Definition 15 (Meaning of a Goal).** The meaning of a goal \( \phi \) w.r.t. a set of axioms \( \mathcal{A} \) (written as \( \text{solve}(\phi, \mathcal{A}) \)) is defined as:

\[
\text{solve}(\phi, \mathcal{A}) = \{ \Delta \vdash \psi : \psi \in \mathcal{A} \text{ such that } \phi \theta = \psi \}
\]

where \( \phi \) is a goal, solve returns a bag, and \( \theta \) is a substitution.
3.5 Building Consistent Databases

Databases are incrementally built, clause-by-clause, starting from an empty database. Given a database (program rules and integrity constraints) $\Delta$, a consistent database $\Delta_k$ is built from $\Delta_0 = \emptyset$ as:

$$\Delta_{i+1} = \Delta_i \cup \{c_i \in \Delta \text{ s.t. safe}(c_i), \text{ there exist } \text{str}(\Delta_i \cup \{c_i\}), \text{ and cons}(\Delta_i \cup \{c_i\})\}$$

4 Tabling-based Query Solving

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 [21], which follows the ideas found in [24, 8, 27]. In this section we assume a database which is a safe, stratifiable and consistent with respect to a given set of integrity constraints. Whilst this section introduces tabling, next section extends this to include hypothetical rules and queries.

4.1 Tabling

Tabled resolution for logic programs evaluate queries by memoizing calls and answers to goals. A call table $ct$ stores the goal calls made along resolution as $\phi$ entries, and answers in an answer table $at$ as id : $\psi$ entries, where id is a clause identifier and $\psi$ is either a positive or negative ground atom. The answer to a query $\phi$ can be specified as $\text{solve}(\phi) = \{\psi \in at \text{ such that } \psi = \phi \theta \text{ for some } \theta\}$.

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

**Definition 16 (Memo Function).** Given a goal $\phi$, a database $\Delta$, a call table $ct_0$, and an answer table $at_0$, the memo function $\text{memo}(\phi, \Delta, ct_0, at_0)$ returns a pair $<ct, at>$ as specified as follows:

- If $\phi \theta \in ct_0$ then:
  - $ct = ct_0$
  - $at = at_0$

- Else:
  - For each program clause $A_j \leftarrow L^j_1 \land \ldots \land L^j_{nj}$ in $\Delta$ identified by id$_{A_j}$,
    - $j \geq 0$, $nj \geq 0$, s.t. $\phi = A_j \theta_0$, with $L^j_i$ either a positive or negative literal, if $\text{memo}(L^j_i \theta_0 \cdots \theta_{i-1}, \Delta, ct_{i-1} \cup \phi, at_{i-1}) = <ct^j_i, at^j_i>$, then let $at^j = at^j_{nj} \cup \text{id}_{A_j} : A_j \theta_0 \cdots \theta_{nj}$ else $at^j = at^j_{nj}$
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 \( \text{at} \) (written as \( \text{cwa} (\text{at}) \)) in the context of a program is the union of \( \text{at} \) and \( \epsilon \) such that \( id : \neg A \) such that \( id : A \notin \text{at} \) for any identifier \( id \), where \( \epsilon \) 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 deduce the meaning of other goals are already in the answer table. So, following the stratification for the program for a given goal \( \phi \), a goal dependency graph is computed, which is the subgraph of the PDG such that contains all reachable nodes from \( \phi \). Then, for each node \( p_i \) in the subgraph such that there is a negative arc coming out from \( p_i \), an open goal \( \phi_i \) is built with the same arity as \( p_i \). Goals \( \phi_i \) are ordered by \( \text{str} (\Delta, \phi_i) \), 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 for a program \( \Delta \) and goal \( \phi \) (written as \( \text{gdg} (\Delta, \phi) \)) is a pair \( \langle N, A \rangle \), where \( \text{pdg} (\Delta, \phi) = \langle N, A \rangle \), \( p_i \in N, i \in \{1, \ldots, k\}, q \xleftarrow{\ i} p_i \in A \) for some \( q, \phi_i = p_i (X_1, \ldots, X_{\text{arity}(p_i)}), X_j \) fresh variables, \( \text{arity}(p_i) \) is the arity of the predicate \( p_i \), and indexes \( i \) are ordered such that \( \text{str} (\Delta, \phi_i) \leq \text{str} (\Delta, \phi_{i+1}) \).

Then, filling the tables is specified as follows:

**Definition 19 (Stratified Meaning of a Program restricted to a Goal).** Given a program \( \Delta \) and a goal \( \phi_{k+1} \)

\[
< ct_i, at_i > = \bigcup_{n \geq 0} \text{memo}^n (\phi_i, \Delta) < ct_{i-1}, at_{i-1} >
\]

where \( \text{pdg} (\Delta, \phi) = \langle N, A \rangle \), \( p_i \in N_i \), \( i \in \{1, \ldots, k\} \), \( q \xleftarrow{\ i} p_i \in A \) for some \( q, \phi_i = p_i (X_1, \ldots, X_{\text{arity}(p_i)}), X_j \) fresh variables, \( \text{arity}(p_i) \) is the arity of the predicate \( p_i \), and indexes \( i \) are ordered such that \( \text{str} (\Delta, \phi_i) \leq \text{str} (\Delta, \phi_{i+1}) \).

Here, \( \bigcup_{n \geq 0} \) represents the least upper bound of the successive applications of the function \( \text{memo} \) as:

\[
\text{memo}^1 (\phi_i, \Delta) < ct_{i-1}, at_{i-1} > = < ct_{i-1}^1, at_{i-1}^1 >
\]

\[
\text{memo}^2 (\phi_i, \Delta) < ct_{i-1}^1, at_{i-1}^1 > = < ct_{i-1}^2, at_{i-1}^2 >
\]

\[
\ldots
\]

\[
\text{memo}^i (\phi_i, \Delta) < ct_{i-1}^{i-1}, at_{i-1}^{i-1} > = < ct_{i-1}^i, at_{i-1}^i >
\]

Then, for \( i = k + 1 \) we get the stratified meaning of the program restricted to \( \phi_{k+1} \) in the answer table \( \text{at}_{k+1} \). 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 \( \phi \) w.r.t. an answer table \( at \) is defined as \( tsolve(\phi, at) = \{ \psi \text{ such that } id : \psi \in at, \text{ and } \phi \theta = \psi \} \) where \( tsolve \) returns a bag, and \( \theta \) is a substitution.

4.2 An Example

Let’s consider a train database \( \Delta \), 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 follows:

\[
\begin{align*}
\text{travel}(X,Y) & \leftarrow \text{link}(X,Y) \\
\text{travel}(X,Y) & \leftarrow \text{travel}(X,Z) \land \text{travel}(Z,Y) \\
\text{no\_travel}(X,Y) & \leftarrow \text{city}(X) \land \text{city}(Y) \land \neg \text{travel}(X,Y)
\end{align*}
\]

The PDG for this program is \( < \{ city, link, travel, no\_travel \}, \{ \text{travel} \leftarrow \text{link}, \text{travel} \leftarrow \text{travel}, \text{no\_travel} \leftarrow \text{city}, \text{no\_travel} \leftarrow \text{travel} \} > \). A stratification can be \( \{ (\text{city}, 1), (\text{link}, 1), (\text{travel}, 1), (\text{no\_travel}, 2) \} \). For solving the goal \( no\_travel(X,Y) \), and following definitions 19 and 20: \( \phi_1 = \text{travel}(X,Y) \) and \( \phi_2 = \text{no\_travel}(X,Y) \) are the goals for building \( < ct_1, at_1 > \) and \( < ct_2, at_2 > \), and the meaning of the goal is \( tsolve(\text{no\_travel}(X,Y), at_2) \). In this case, \( pdg(\Delta) \) coincides with \( gdg(\Delta, \phi_2) \).

4.3 Implementing Tabling

This subsection describes a concrete implementation of the tabling mechanism as found in the deductive database system DES (based on [8]). 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 [8] for an extension table) and the call table \( ct \) with the dynamic predicate \( called/1 \). Entries in \( et \) can be either positive (\( A \)) or negative (\( \text{not}(A) \)), 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 \( \theta \) is a success substitution. A negative entry \( \text{not}(A) \) is added to \( et \) if a (ground) call to \( A \) cannot be proven\(^1\). An entry is added to any of these tables if such an entry does not occur already in that table. In order to account for duplicates, each entry in \( et \) is tagged with an identifier to determine its source [20].

During tabled resolution, a tabling tree is constructed analogously to [26]: 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 \texttt{solve\_star} below) are returned upon backtracking. And, second,

\(^1\) All calls to negated atoms are ground as we require safe rules [28].
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 \textit{memo} is implemented following the \textit{ET} algorithm in [8] with the predicate \texttt{memo(+G,+D,-Id)}, where its arguments are, respectively, the input goal, duplicate elimination flag, and output goal identifier [20]. 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 \texttt{et_lookup}, or 2) solving the goal with the predicate \texttt{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 \texttt{solve}. If the argument of \texttt{solve} is an atom, a straight call to \texttt{memo} is done. If the argument of \texttt{solve_goal} is a negated goal, the it calls to \texttt{solve}, succeeding if this call fails and vice versa. For the concrete implementation, \texttt{solve_goal} also computes built-ins as aggregates, and \texttt{solve} computes conjunctions and metapredicates.

\% Already called. Call table with an entry for the current call
\texttt{memo(G,D,Id) :-
  build(G,Q), \% Build in Q the same call with fresh variables
called(Q), \% Look for a unifiable call in CT for the current call
subsumes(Q,G), \% Test whether CT call subsumes the current call
\%
  \texttt{et_lookup(G,D,Id).} \% If so, use the results in answer table (ET)
}\% New call. Call table without an entry for the current call
\texttt{memo(G,D,Id) :-
  assertz(called(G)), \% Assert the current call to CT
  ( (\texttt{et_lookup(G,D,Id)})) \% First call returns all previous answers in ET
  ;
  (\texttt{solve_goal(G,Id)}, \% Solve the current call using applicable rules
  build(G,Q), \% Build in Q the same call with fresh variables
  no_subsumed_by_et(Q,D,(G,Id)), \% Test whether there is no entry in ET for Q
  et_assert(G,Id), \% If so, assert the current result in ET
  et_changed)). \% Flag the change

Successive applications of the function \texttt{memo} (as presented after Definition 20) until a fixpoint is reached [20] is implemented with the predicate \texttt{solve_star/2} and it is shown next (following \textit{ET*} [8]):

\texttt{solve_star(Q) :-
  repeat,
  (remove_calls, \% Clear CT
  et_not_changed, \% Flag ET as not changed
  solve(Q), \% Solve the call to Q using memoization at stratum St
  fail \% Request all alternatives
  ;
  no_change, \% If no more alternatives, start a new iteration
  !, fail). \% Otherwise, fail and exit
Filling the tables following the stratified approach in Definition 19 is implemented with the predicate `solve_stratified/1` as [20]:

```prolog
solve_stratified(Query) :-
  sub_pdg(Query, (Nodes, Arcs)),
  (neg_dependencies(Arcs) ->
   solve_star(Query, 1) % Solve the query if no negative dependencies
  ;
  strata(St), % Solve queries by stratum order
  sort_by_strata(St, Arcs, Preds),
  build_queries(Preds, Query, Queries),
  solve_star_list(Queries)).
```

5 Hypothetical Tabling

The inference system defined in Section 3 (cf. Definition 10) includes the inference rule for hypothetical goals, which must be included in the memo function (cf. Definition 16). 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 \( R_1 \lor \ldots \lor R_n \Rightarrow \phi \), where \( R_i \) are rules and \( \phi \) a goal, the database \( \Delta \) for which this hypothetical literal is to be proved must be augmented with \( \{ R_1, \ldots, R_n \} \). Deductions delivered in proving \( \phi \) are only valid in the context of the augmented database, i.e., in the tabling tree constructed for \( \phi \). So, such deductions must be tagged in order to be only used in its context. To this end, a context identifier is added to the answer and call tables as well as a new parameter to the extended memo function.

**Definition 21 (Hypothetical Memo Function).** Given a goal \( \phi \), a database \( \Delta \), a context identifier \( \chi \), a call table \( ct_0 \) and an answer table \( at_0 \), the memo function \( \text{memo}(\phi, \Delta, \chi, ct_0, at_0) \) returns a pair \( ct, at \) as specified as follows:

- If \( \phi \theta \in ct_0 \) then:
  - \( ct = ct_0 \)
  - \( at = at_0 \)
- Else:
  - For each program clause \( A_j \leftarrow L_{j_1}^1 \land \ldots \land L_{j_n}^1 \) in \( \Delta \) identified by \( id_{A_j} \), \( j \geq 0, n_j \geq 0, s.t. \phi = A_j \theta_0 \).
    * For \( L_i^j \) either a positive or a negative literal, if \( \text{memo}(L_i^j, \theta_0, \ldots, \theta_{i-1}, \Delta, \chi, ct_1^j \cup \phi, at_1^j) = \langle ct_1^j, at_1^j \rangle, \text{id}_{L_i^j} : L_i^j \theta_0 \ldots \theta_{i-1} \in \text{cwa}(at_1^j) \), then let \( at^j = at_1^j \cup \text{id}_{L_i^j} : A_j \theta_0 \ldots \theta_{n_j} \text{ else } at^j = at_1^j \).
    * For \( L_i^j \) a hypothetical goal, if \( \text{memo}(L_i^j, \theta_0, \ldots, \theta_{i-1}, \Delta \cup \{ R_1, \ldots, R_n \}, \chi', ct_1^j \cup \text{id}_{\chi} : \phi, at_1^j) = \langle ct_1^j, at_1^j \rangle, \text{id}_{L_i^j} : L_i^j \theta_0 \ldots \theta_i \in \text{cwa}(at_1^j) \), where \( \chi' \) is a context identifier for \( L_i^j \), then let \( at^j = at_1^j \cup \text{id}_{\chi'} : A_j \theta_0 \ldots \theta_{n_j} \text{ else } at^j = at_1^j \).
  - \( ct = \bigcup ct_1^j \cup \text{id}_{\chi} \)
  - \( at = \bigcup at_1^j \cup \text{id}_{\chi} \).
The closed world assumption of an answer table (Definition 12) is modified accordingly:

**Definition 22 (Hypothetical Closed World Assumption of an Answer Table).** The closed world assumption of an answer table at \( \text{cwa}(at) \) in the context of a program is the union of \( at \) and \( \epsilon^x : \neg A \) such that \( id^x : A \notin at \) for any rule identifier \( id \) and context \( \chi \), where \( \epsilon \) is a fixed, arbitrary identifier which does not occur in the program.

The stratified meaning of a program restricted to a goal is simply extended by adding the extra context parameter (cf. Definition 19):

**Definition 23 (Stratified Hypothetical Meaning of a Program restricted to a Goal).** Given a program \( \Delta \) and a goal \( \phi_{k+1} \)

\[
< ct_i, at_i > = \bigcup_{n \geq 0} \text{memo}^n(\phi_i, \chi_i) < ct_{i-1}, at_{i-1} >
\]

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

Also, the meaning of a tabled goal in a context is defined analogously to the tabled meaning of a goal (Definition 20):

**Definition 24 (Hypothetical Meaning of a Tabled Goal).** The meaning of a tabled goal \( \phi \) w.r.t. an answer table \( at \) in the context \( \chi \) is defined by \( \text{tsolve}(\phi, \chi, at) = \{ \psi \mid \text{id}^{x} : \psi \in at, \ \text{and} \ \phi \theta = \psi \} \) where \( \text{tsolve} \) returns a bag, and \( \theta \) is a substitution.

### 5.1 Implementing Hypothetical Tabling

The new context parameter is added to the predicates \( \text{memo}, \text{called}, \text{et\_lookup}, \text{solve\_goal}, \) and \( \text{solve} \), and the dynamic predicates \( \text{et} \) and \( \text{ct} \). 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 3.5) or an augmented database due to embedded implications. So, \( \text{et\_lookup} \) and \( \text{called} \) look for entries corresponding to the context which is being computed. In addition, the dynamic predicate \( \text{datalog} \), 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 \( \text{memo} \) might lead to adding the same hypothetical rules for the same context. To avoid this, we tag each context with a dynamic predicate \( \text{hyp\_program\_asserted} \) 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:

```prolog
solve_implication(L => R,CId,Rule,Ids) :-
    dlrule_id(Rule,RId),
    assert_hyp_program(RId,L,R,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 [20]. 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. This allows to check non-valid rules as illustrated in Subsection 2.3.

### 5.2 Reusing Answers from Previous Contexts

The introduced implementation for solving an embedded implication requires to recompute from scratch the given goal for all the involved strata. While this is a conservative approach, we can take advantage of former computations in previous contexts to avoid some recomputations, i.e., reusing entries in the answer table.
for all the former contexts. For the database restricted to the goal (following the
goal dependency graph – Definition 18) consisting only of a single stratum this
reusing is safe as only additions to the answer table are expected. So, retrievals
from the answer table can be done from the first context up to the current one.
However, when negation is involved in this restricted database, some already
deduced information in a former context might be not true anymore. Consider,
for instance, the following program with rules identified with numbers:

1: p :- not(q).
2: r :- q => p.

The goal p succeeds in the context [], but fails in the context [2]. (A similar
situation occurs in the example of Subsection 2.5.) A straightforward implementa-
tion for facing this issue is simply to avoid the reusing for strata greater than
1, which can be done by adding a new parameter to the predicates stating the
current stratum. Another, more refined implementation is by identifying those
predicates which do not depend on assumed information, either directly or indi-
rectly, and avoiding the reusing of their deduced information, committing only
to the current context.

6 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.
However, this approach is somewhat different from works as [2] because since
it is intended to assume program rules, their variables are neither shared with
the literal in which they occur, nor with the rule in which the literal occurs.
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. This is subject of future work, as optimizations in Subsection 5.2.

References

1. Andrey Balmin, Thanos Papadimitriou, and Yannis Papakonstantinou. Hypo-
thetical queries in an olap environment. In Proceedings of the 26th International
Conference on Very Large Data Bases, VLDB ’00, pages 220–231, San Francisco,
2. A. J. Bonner and L. T. McCarty. Adding negation-as-failure to intuitionistic logic
programming. In Ewing L. Lusk and Ross A. Overbeek, editors, Proc. of the North
3. Anthony J. Bonner. Hypothetical datalog: Complexity and expressibility. Theo-


