Tabling with Sound Answer Subsumption

Alexander Vandenbroucke
KU Leuven, Belgium
(*e-mail:* alexander.vandenbroucke@kuleuven.be)

Maciej Piróg
KU Leuven, Belgium
(*e-mail:* maciej.pirog@kuleuven.be)

Benoit Desouter
Ghent University, Belgium
(*e-mail:* benoit.desouter@ugent.be)

Tom Schrijvers
KU Leuven, Belgium
(*e-mail:* tom.schrijvers@kuleuven.be)

submitted 29 April 2016; revised 8 July 2016; accepted 22 July 2016

Abstract

Tabling is a powerful resolution mechanism for logic programs that captures their least fixed point semantics more faithfully than plain Prolog. In many tabling applications, we are not interested in the set of all answers to a goal, but only require an aggregation of those answers. Several works have studied efficient techniques, such as lattice-based answer subsumption and mode-directed tabling, to do so for various forms of aggregation.

While much attention has been paid to expressivity and efficient implementation of the different approaches, soundness has not been considered. This paper shows that the different implementations indeed fail to produce least fixed points for some programs. As a remedy, we provide a formal framework that generalises the existing approaches and we establish a soundness criterion that explains for which programs the approach is sound.

KEYWORDS: tabling, answer subsumption, lattice, partial order, mode-directed tabling, denotational semantics, Prolog

1 Introduction

Tabling considerably improves the declarativity and expressiveness of the Prolog language. It removes the sensitivity of SLD resolution to rule and goal ordering, allowing a larger class of programs to terminate. As an added bonus, the memoisation of the tabling mechanism may significantly improve run time performance in exchange for increased memory usage. Tabling has been implemented in a few well-known Prolog systems, such as XSB (Swift and Warren 2010; Swift and Warren 2012),Yap (Santos Costa et al. 2012), Ciao (Chico de Guzmán et al. 2008) and B-Prolog (Zhou 2012), and has been successfully applied in various domains.

Much research effort has been devoted to improving the performance of tabling for

various specialised use-cases (Swift 1999; Ramakrishna et al. 1997; Zhou and Dovier 2011). This paper is concerned with one such fairly broad class of use-cases: we are not directly interested in all the answers to a tabled-predicate query, but instead wish to aggregate these answers somehow. The following shortest-path example illustrates this use-case.

```
query(X,Y,MinDist) :- findall(Dist,p(X,Y,Dist),List), min_list(List,MinDist).
:- table p/3.
p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- p(X,Z,D1), p(Z,Y,D2), D is D1 + D2.
e(a,b). e(b,c). e(a,c).
```

The query `?- query(a,c,D).` computes the distance `D` of the shortest path from `a` to `c` by first computing the set of distances $\{1, 2\}$ of all paths and then selecting the smallest value from this set. Unfortunately, when the graph is cyclic, the set of distances is infinite and the query never returns, even though the infinite set has a well-defined minimal value.

Various tabling extensions (know collectively as answer subsumption: mode-directed tabling (Guo and Gupta 2004; Guo and Gupta 2008; Zhou et al. 2010; Santos and Rocha 2013), partial order answer subsumption and lattice answer subsumption (Swift and Warren 2012)), have come up with ways to integrate the aggregation into the tabled resolution. This way answers are incrementally aggregated and the tabling may converge more quickly to the desired results. For instance, the shortest-path program can be written with mode-directed tabling as:

```
:- table p(+,+,min).
p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- p(X,Z,D1), p(Z,Y,D2), D is D1 + D2.
e(a,b). e(b,c). e(a,c).
```

Here the query `?- p(a,c,D).` yields only the shortest distance. It does so by greedily throwing away non-optimal intermediate results and in this way only considers finitely many paths, even if the graph is cyclic. In summary, this approach makes tabling (sometimes infinitely) more efficient for our aggregation use-case.

Unfortunately, none of the existing implementations that we are aware of is generally sound. Consider the following pure logic program.

```
p(0). p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

The query `?- p(X).` has a finite set of solutions, {`p(0),p(1),p(2),p(3)`}, the largest of which is `p(3)`. However, XSB, Yap and B-Prolog all yield different (invalid) solutions when answer subsumption is used to obtain the maximal value. Both XSB and B-Prolog yield `X = 2`, with a maximum lattice and `max` table mode respectively. Yap (also with `max` table mode) yields `X = 0; X = 1; X = 2`, every solution except the right one.[1]

Clearly, these results are unsound. This example is not the only erroneous one; we can easily construct more erroneous scenarios with other supported forms of aggregation. Hence, we must conclude that answer subsumption is in general not a semantics-preserving optimisation. Yet, as far as we know, the existing literature does not offer any guidance on when the feature can be relied upon. In fact, to our knowledge, its semantics have not been formally discussed before.

[1]The batch scheduling used by Yap returns any answer as soon as it is found.

This paper fills the semantic gap of answer subsumption with the help of lattice theory. We show how the existing implementations fit into this semantic framework and derive a sufficient condition for semantics preservation that allows answer subsumption to be safely used.

2 Background: Tabling Semantics

Because the operational semantics of tabling is rather complex and different systems vary in subtle ways, we make a simplifying assumption and assume that tabling systems implement Lloyd's least fixed-point semantics (Lloyd 1984) for definite logic programs, that is, that tabling is a sound program optimisation with no impact on the denotation of a program. This semantics conveniently abstracts from low-level aspects such as clause and goal ordering and the specific clause scheduling algorithm used by the Prolog engine.

2.1 Least Fixed-Point Semantics

First, we need the notion of a *Herbrand base*: the set of all possible (ground) atoms that occur in a logic program. More formally, let Σ be an alphabet, and P be a logic program over Σ, then the Herbrand base H_P is the set of all ground atoms over Σ. For example, the Herbrand base of the shortest path program (without `query/3`,`+/2`) is:

$$H_P = \{\mathtt{e}(X,Y) \mid X,Y \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}\} \cup \{\mathtt{p}(X,Y,D) \mid X,Y \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}, D \in \mathbb{N}\}$$

A *(Herbrand) interpretation* is a set $I \subseteq H_P$. Intuitively, it contains atoms in the Herbrand base that are true: $\forall a \in H_P : I \models a \iff a \in I$.

Finally, define the operator $T_P : \mathcal{P}(H_P) \to \mathcal{P}(H_P)$ such that, given an interpretation I, the value $T_P(I)$ is the interpretation that immediately follows from I by any of the program rules:

$$T_P(I) = \{B_0 \in H_P \mid B_0 \leftarrow B_1, \ldots, B_n \in \mathsf{ground}(P) \wedge \{B_1, \ldots, B_n\} \subseteq I\} \tag{1}$$

This operator is called the *immediate consequence operator*. Its least fixed-point with respect to subset-inclusion ($\subseteq$), denoted $\mathsf{lfp}(T_P)$, defines the semantics of the program P, and is also known as the *least Herbrand model*. It is the interpretation that contains those and only those atoms that follow from the program and that are not self-supported.

Example 1 Consider the following program P:

```
p(a). p(b). q(c).
q(X) :- p(X).
```

Its Herbrand base is $\{\mathtt{p(a)}, \mathtt{p(b)}, \mathtt{p(c)}, \mathtt{q(a)}, \mathtt{q(b)}, \mathtt{q(c)}\}$. Its fixed-point semantics is:

$$\mathsf{lfp}(T_P) = \{\mathtt{p(a)}, \mathtt{p(b)}, \mathtt{q(a)}, \mathtt{q(b)}, \mathtt{q(c)}\}$$

Observe that this is exactly the set of atoms that follow from the program.

2.2 Existence and Computability of the Least Herbrand Model

The least fixed-point semantics is not necessarily well-defined: it is not immediate that the least fixed-point actually exists. Moreover, if it exists, it may not actually be constructively computable.

Fortunately, there is no reason for concern: by appeal to a well-known theorem from lattice theory, we can easily establish the well-definedness. A *complete lattice* is a partially ordered set (poset) $\langle L, \leq_L \rangle$ such that every $X \subseteq L$ has a least upper bound $\bigvee X$, i.e.:

$$\forall z \in L : \bigvee X \leq_L z \iff \forall x \in X : x \leq_L z$$

We do indeed have a lattice structure at hand: the power set of the Herbrand base $\langle \mathcal{P}(H_P), \subseteq \rangle$ is a complete lattice. In fact, any power set is a complete lattice. Moreover, it is quite easy to see that if P is a definite logic program (i.e., contains no negations), then T_P is monotone with respect to this lattice. It follows that $\mathsf{lfp}(T_P)$ exists, ensuring that the semantics is well-defined for every definite program P, by the following theorem:

Theorem 2.1 (*Knaster–Tarski*)
Let $\langle L, \leq_L \rangle$ be a complete lattice, and let $f : L \to L$ be a monotone function. Then, f has a least fixed point, denoted $\mathsf{lfp}(f)$.

Moreover, the T_P operator is ω-continuous, which means that for all ascending chains $l_1 \subseteq l_2 \subseteq \ldots$ with $l_1, l_2, \ldots \subseteq H_P$, it is the case that $\bigcup_{i=1}^{\infty} T_P(l_i) = T_P(\bigcup_{i=0}^{\infty} l_i)$. Then, Kleene's fixed-point theorem gives us a constructive way of obtaining $\mathsf{lfp}(T_P)$:

$$\mathsf{lfp}(T_P) = \bigcup \{T_P(\emptyset), T_P^2(\emptyset), \ldots\}$$

The least fixed-point can therefore be obtained in a bottom-up fashion by iterating T_P from the empty set onward. Operationally, tabling usually interleaves a top-down goal-directed strategy with bottom-up iteration. The bottom-up strategy always computes the entire least Herbrand model, even when only a small portion of it may be required to prove a particular query. The top-down part of tabling avoids computing irrelevant atoms as much as possible, making inference feasible.

2.3 Stratification

Unfortunately, the T_P-operator is not monotone for programs containing more advanced constructs, such as negation. Therefore, Lloyd's semantics as described above is not suitable for capturing the semantics of such programs. In the case of negation, this problem is solved by partitioning the clauses of a program into an ordered set of *strata* based on their interdependence. This procedure is called *stratification* (Apt et al. 1988). Then, the semantics for each stratum is computed based on the semantics of the lower strata, with no relation to the higher strata. To make this more concrete, suppose a ground program P admits a stratification $P_1, \ldots, P_n$, with the P_i non-empty and pairwise disjoint, then:

$$\begin{aligned} &P = P_1 \cup \cdots \cup P_n, && \\ &Q_1 = P_1,\ Q_{i+1} = P_{i+1} \cup M_i, && \text{for all } i = 0, \ldots, n-1 \\ &M_i = \mathsf{lfp}(T_{Q_i}) && \text{for all } i = 1, \ldots n \end{aligned}$$

where M_i should be understood as a set of facts. If a program admits a stratification where all negated calls are to predicates defined in lower strata, the obvious extension T_P^{neg} of the T_P operator to include negation is guaranteed to be monotone. The semantics of P is then given by $\bigcup_{i=1}^{n} M_i$.

3 Answer Subsumption Approaches

In this section, we propose a denotational semantics for tabling with answer subsumption. Broadly speaking, we modify the semantics for stratified programs as described in the previous section in two respects. First, our semantics includes new answers that may emerge from the program-defined rules of subsumption, which are not necessarily logical consequences of the same program without answer subsumption. We obtain this by extending the T_P operator. Secondly, we perform the actual subsumption, that is, we remove the subsumed answers. Stratification, as discussed in Section 3.4, is used to control the order in which these two steps are invoked.

In a bit more detail, the semantics of a stratum is given by the extended immediate consequence operator, which we call $\widehat{T}_P$, and a function $\eta : H_P \to L$ that aggregates the answers using a lattice L. A consequence of this specification is that an aggregation naturally ignores all operational aspects of the program P. That is to say, given two structurally distinct programs P_1 and P_2 whose least fixed-point semantics coincide, that is, $\mathsf{lfp}(\widehat{T}_{P_1}) = \mathsf{lfp}(\widehat{T}_{P_2})$, it follows that $\bigvee \eta(\mathsf{lfp}(\widehat{T}_{P_1})) = \bigvee \eta(\mathsf{lfp}(\widehat{T}_{P_2}))$, i.e. their aggregates coincide as well.

Obviously, the existing systems do not implement answer subsumption as a single post-processing function. Instead, they execute it repeatedly during the bottom-up phase of the computation, which sometimes makes them deviate from the intended semantics, as exemplified in the introduction. We formalise and deal with this in Section 4.

For now, we assume that the program P has only one stratum. Towards the end of the section, we show how to assemble the semantics of programs with any number of strata.

3.1 Mode-Directed Tabling

Mode-directed tabling is a convenient aggregation approach supported by ALS-Prolog, B-Prolog and Yap where the arguments of a tabled predicate are annotated with one of a range of aggregation *modes*. Yap provides the largest range of possible modes: `index`, `first`, `last`, `min`, `max` and `all`. The answers are grouped by distinct values for the `index` arguments; the remaining arguments are aggregated according to their mode – the mode names should be self-explanatory.

Based on our assumptions so far, we can immediately disqualify the existing implementations of the three modes `first`, `last` and `sum`. The reason is that the semantics of the existing implementations is inherently sensitive to the program structure. Consider the two programs below:

```
% P1
:- table p(first).
p(1). p(2).
```

```
% P2
:- table p(first).
p(2). p(1).
```

Clearly $\mathsf{lfp}(T_{P1}) = \{\mathtt{p(1)}, \mathtt{p(2)}\} = \mathsf{lfp}(T_{P2})$, however $P1$ yields `p(1)` as an answer for `?- p(X).` while $P2$ yields `p(2)`. The opposite happens with the `last` mode. The next programs illustrate the problem of the `sum` mode:

```
% P3
:- table p(sum).
p(1).
```

```
% P4
:- table p(sum).
p(1). p(1).
```

Again the least fixed-point semantics of both programs coincides: $\mathsf{lfp}(T_{P3}) = \{\mathtt{p(1)}\} = \mathsf{lfp}(T_{P4})$. However, they produce the following results in Yap:

```
% P3
?- p(X).
X = 1.
```

```
% P4
?- p(X).
X = 1 ; X = 1.
?- p(X).
X = 2.
```

Yap produces the result `p(1)` twice the first time the query is called. Any subsequent query is answered with `p(2)`. In other words, not only are the results of $P3$ and $P4$ not consistent, the results for $P4$ are not internally consistent either.

In the rest of this paper we disregard these three modes. As their implementations are so obviously sensitive to the program structure, we do not see a good way to reconcile them with our semantics-oriented post-processing specification for aggregation. In fact, in our opinion these modes are best avoided in high-level logic programs.

3.2 Lattice-Based Approaches

The remaining three modes, `min`, `max` and `all`, share one notable property: they are all based on a join-semilattice structure defined on (subsets of) U_P, the set of all ground terms over the alphabet of P. A *join-semilattice* is a poset $\langle S, \leq_S \rangle$ such that every finite subset $X \subseteq S$ has a least upper bound in S, which we denote, as in the case of complete lattices, $\bigvee X$. For example, the set of natural numbers with standard order $\langle \mathbb{N}, \leq \rangle$ is a join-semilattice (with $\bigvee X = \mathsf{max}\, X$), but it is not a complete lattice. Different modes define the following join-semilattices:

- `min` defines the join-semilattice $\langle U_P, \leq \rangle$ where $\leq$ is the lexicographical ordering on terms (`=</2`). The least upper bound is the minimum.
- `max` defines the join-semilattice $\langle U_P, \geq \rangle$ where $\geq$ is the inverse of $\leq$. The least upper bound is the maximum.
- `all` defines the join-semilattice $\langle \mathcal{P}^{\mathsf{fin}}(U_P), \subseteq \rangle$ where $\mathcal{P}^{\mathsf{fin}}(U_P)$ is the set of all finite subsets of U_P. Existing implementations represent sets as lists of terms, which are themselves terms.

The two additional aggregation approaches, offered by XSB, are also based on join-semilattices:

- XSB generalises the above modes to user-defined join-semilattices with the `lattice(`$\bigvee$`/3)` mode that is parameterised in a binary join operator. For instance, we can define the `min` mode as `lattice(min/3)`.
- XSB also provides a second user-definable mode `po(`$\preceq$`/2)` in terms of a partial order $\preceq$ on U_P. This partial order induces a join-semilattice $\langle \mathcal{P}^{\mathsf{fin}}(U_P), \sqsubseteq \rangle$ where $X \sqsubseteq Y \equiv \forall x \in X : \exists y \in Y : x \preceq y$.

Therefore, in what follows, we only have to deal with lattices that are essentially subsets of U_P, considerably simplifying the formulae.

As we can always reorder arguments and combine multiple join-semilattices into their product join-semilattice, we assume, without loss of generality, that only the final argument of a predicate is an output tabling mode. That is, all ground atoms have the shape $Q(\boldsymbol{X}, x)$ where Q is the name of some predicate, $\boldsymbol{X}$ is a vector of input arguments $X_1, X_2, \ldots, X_n$ and x is the value of the output parameter. We make the simplifying assumption that all predicates are tabled. If a predicate has only input arguments (like tabling without answer subsumption), a (constant) dummy output can always be added.

Mode-directed tabling groups atoms for a predicate Q by distinct values for the *input* arguments $\boldsymbol{X}$ and aggregates the values of the *output* argument x into a single term. Therefore, we model a table of aggregated answers by a function $table : I_P \times U_P^n \to U_P^\perp$ (where I_P is the set of predicate names in P) that maps a pair of a predicate name and inputs to a single aggregated output. The set of aggregate answers $U_P^\perp = U_P \cup \{\perp\}$ is the set of all terms, on which a special element $\perp$ is grafted, to indicate the lack of an answer. We extend the chosen order on terms $\preccurlyeq$ such that $\perp$ is an (adjoined) bottom element, that is, $\forall x \in U_P^\perp : \perp \preccurlyeq x$. For legibility, we will also sometimes refer to $I_P \times U_P^n \to U_P^\perp$ by $L(\preccurlyeq)$. The lattice structure $\langle U_P^\perp, \preccurlyeq\rangle$ induces a join-semilattice structure $\langle L(\preccurlyeq), \preceq\rangle$ on the set of tables, where $\preceq$ is the pointwise order:

$$f \preceq g \iff \forall (p, \boldsymbol{X}) \in I_P \times U_P^n : f(p, \boldsymbol{X}) \preccurlyeq g(p, \boldsymbol{X})$$

This lattice structure allows us to aggregate over multiple tables, by aggregating the answers pointwise:

$$(\bigvee F)(p, \boldsymbol{X}) = \bigvee_{f \in F} f(p, \boldsymbol{X})$$

By storing each individual element of $\mathsf{lfp}(T_P)$ into a table and then aggregating over tables we obtain the semantics for mode directed tabling:

Let $\eta : H_P \to (I_P \times U_P^n \to U_P^\perp)$ and $\rho : (I_P \times U_P^n \to U_P^\perp) \to \mathcal{P}(H_P)$ be defined as:

$$\eta(p(\boldsymbol{X}, x))(q, \boldsymbol{Y}) = \begin{cases} x & \text{if } p = q \wedge \boldsymbol{X} = \boldsymbol{Y} \\ \perp & \text{otherwise} \end{cases}$$

$$\rho(f) = \{p(\boldsymbol{X}, f(p, \boldsymbol{X})) \mid f(p, \boldsymbol{X}) \neq \perp\}$$

Thus, the function η turns an atom into a singleton table, and ρ maps a table to the set of its true atoms.

To compute the set of all true atoms of a program P, we need to consider the consequence of joining two elements of a semi-lattice in addition to regular logical consequences. This is because, for arbitrary lattices, the result of a join can be distinct from any of its arguments, and thus produce new facts. We define a new immediate consequence operator $\widehat{T}_P$, which extends the regular T_P operator with answers obtained by joins. Formally, we define $\widehat{T}_P$ as follows, where $\mathcal{P}^{\text{fin}}(A)$ denotes the set of all finite subsets of a set A:

$$\widehat{T}_P(X) = \bigcup \{\rho(\bigvee Y) \mid Y \in \mathcal{P}^{\text{fin}}(\eta(T_P(X)))\} \tag{2}$$

One can show that $\widehat{T}_P$ is continuous, hence monotone. In fact, for linear orders (such as `min` and `max`), $\widehat{T}_P$ behaves exactly like T_P. Again, we consider the least fixed-point of $\widehat{T}_P$ to be the set of all the answers that can be obtained by the logical rules and the 'lattice rules'.

The next step is to discard the subsumed answers by applying the join operator on the set of answers. Thus, the set of all true atoms of the program P using mode-directed tabling is given by:

$$\rho\left(\bigvee_{x \in \mathsf{lfp}(\widehat{T}_P)} \eta(x)\right) \tag{3}$$

Obviously, when $\mathsf{lfp}(\widehat{T}_P)$ is infinite, the least upper bound above does not necessarily exist. That is why to give a full denotational semantics of answer subsumption in the

next subsection, we model tables in a more abstract way as complete lattices. Now, to provide some intuition, we give an example in which the least upper bound exists.

Example 2 Consider the example from the introduction, rewritten using XSB's lattice answer subsumption for the sake of variety:

```
:- table p(lattice(_,_,min/3)).
:- table e/3.
p(X,Y,1) :- e(X,Y,nt).
p(X,Y,D) :- p(X,Z,D1), p(Z,Y,D2), D is D1 + D2
e(a,b,nt). e(b,c,nt). e(a,c,nt).
min(X,Y,Z) :- Z is min(X,Y).
```

Note that we have additionally tabled `e/3` and added a dummy output parameter (`nt` stands for not tabled), as described above. The don't cares (_) in the tabling directive indicate that they are not part of the lattice. In Yap's terminology: they use the `index` tabling mode. The least fixed-point semantics of this program, that is $\mathsf{lfp}(\widehat{T}_P)$, is given by the following set:

$$\{\mathtt{e(a,b,nt)}, \mathtt{e(b,c,nt)}, \mathtt{e(a,c,nt)}, \mathtt{p(a,b,1)}, \mathtt{p(b,c,1)}, \mathtt{p(a,c,1)}, \mathtt{p(a,c,2)}\}$$

The complete lattice on the final argument of `p` is $\langle \mathbb{N}, \geq \rangle$, the reversed standard order. The least upper bound in this lattice is the usual *infimum* on natural numbers.

Interpreted by this lattice, the semantics is

$$\begin{aligned}
&\rho\left(\bigvee_{x \in \mathsf{lfp}(\widehat{T}_P)} \eta(x)\right) \\
&= \rho(\bigvee\{\eta(\mathtt{e(a,b,nt)}), \eta(\mathtt{e(b,c,nt)}), \eta(\mathtt{e(a,c,nt)}), \\
&\qquad \eta(\mathtt{p(a,b,1)}), \eta(\mathtt{p(b,c,1)}), \eta(\mathtt{p(a,c,1)}), \eta(\mathtt{p(a,c,2)})\}) \\
&= \rho(t) \text{ where } t(q,x,y) = \begin{cases} \mathtt{nt} & \text{if } q = \mathtt{e} \\ 1 & \text{if } (x = \mathtt{a} \wedge y = \mathtt{b}) \vee (x = \mathtt{b} \wedge y = \mathtt{c}) \\ \min\{1,2\} & \text{if } x = \mathtt{a} \wedge y = \mathtt{c} \\ \bot & \text{otherwise} \end{cases} \\
&= \{\mathtt{e(a,b,nt)}, \mathtt{e(b,c,nt)}, \mathtt{e(a,c,nt)}, \mathtt{p(a,b,1)}, \mathtt{p(b,c,1)}, \mathtt{p(a,c,1)}\}
\end{aligned}$$

Only the atoms representing the shortest paths are retained, as expected.

Example 3 This example illustrates why we need to extend the T_P operator to include the results of the lattice operations, that is, why we need the $\widehat{T}_P$ operator. Consider the lattice $\{a, b, c, d\}$, with $a, b \leq c$ and $c \leq d$, which we use in the following program:

```
lub(a,b,c). lub(a,c,c). lub(a,d,d).
lub(b,a,c). lub(b,c,c). lub(b,d,d).
lub(c,d,d).
lub(X,X,X).

:- table p(lattice(lub/3)).

p(a).
p(b).
p(d) :- p(c).
```

The regular immediate consequence gives us $\mathsf{lfp}(T_P) = \{\mathtt{p(a)}, \mathtt{p(b)}\}$, which means that

$\rho(\bigvee \eta(\mathsf{lfp}(T_P))) = \{\mathtt{p(c)}\}$. The atom $\mathtt{p(c)}$ does not follow from the logical inference, but from the lattice's join operator. It is included in the overall answer thanks to the post-processing step, but its logical consequences are not. With the $\widehat{T}_P$ operator we have $\mathsf{lfp}(\widehat{T}_P) = \{\mathtt{p(a)}, \mathtt{p(b)}, \mathtt{p(c)}, \mathtt{p(d)}\}$, and so $\rho(\bigvee \eta(\mathsf{lfp}(\widehat{T}_P))) = \{\mathtt{p(d)}\}$, which is the intended semantics.

3.3 Answer Subsumption for Arbitrary Lattices

Even though at any point of computation each table is finite, it is potentially infinite when a program produces infinitely many answers. Thus, to give a denotational semantics for answer subsumption, a join-semilattice on terms is not enough, as we need least upper bounds of infinite sets, i.e. a complete lattice structure. For example, the most natural candidate for the types of values in the case of the `all` mode is $\langle \mathcal{P}(U_P), \subseteq \rangle$, the complete lattice of all subsets of U_P, which cannot be modelled by (finite) terms. In general, every semilattice can be extended to a complete lattice via MacNeille (1937) completion.

Thus, we do not impose any order on the set $U_P^{\perp}$, and the type of the table becomes $(I_P \times U_P^n \to L)$ for a complete lattice L. For each predicate $p \in I_P$, we also need two bottom-preserving *abstraction* and *representation* functions: $\lceil\text{-}\rceil_p : U_P^{\perp} \to L$ and $\lfloor\text{-}\rfloor_p : L \to U_P^{\perp}$ respectively. We require $\lceil\text{-}\rceil_p$ to be a retraction of $\lfloor\text{-}\rfloor_p$, that is, $\lceil \lfloor x \rfloor_p \rceil_p = x$. Since we want the two functions to preserve bottoms, the least element of L denotes 'no value'. With this, we give new definitions of η and ρ, appropriately adding abstraction and representation, where $\perp^L$ is the least element of L:

$$\eta(p(\boldsymbol{X}, x))(q, \boldsymbol{Y}) = \begin{cases} \lceil x \rceil_p & \text{if } p = q \wedge \boldsymbol{X} = \boldsymbol{Y} \\ \perp & \text{otherwise} \end{cases}$$

$$\rho(f) = \{p(\boldsymbol{X}, \lfloor f(p, \boldsymbol{X}) \rfloor_p) \mid f(p, \boldsymbol{X}) \neq \perp^L\}$$

To give the semantics, we define the $\widehat{T}_P$ operator exactly as in (2) but using the new definitions of η and ρ. It is easy to see that it is monotone, so it always has a least fixed point. The semantics of the entire program is given again as in (3).

3.4 Lattice Semantics for Stratified Programs

For general programs, we use stratification to distinguish between predicates that imply and are implied by tabled values. We define the *depends on* relation $\ltimes$ as follows: for any two predicates $\mathtt{p}$ and $\mathtt{q}$, it is the case that $\mathtt{p} \ltimes \mathtt{q}$ if and only if there exists a clause `p(...):-...,q(...),...`. We say that $\mathtt{p}$ and $\mathtt{q}$ are in the same stratum if $\mathtt{p} \ltimes^+ \mathtt{q}$ and $\mathtt{q} \ltimes^+ \mathtt{p}$, where $\ltimes^+$ is the reflexive and transitive closure of $\ltimes$. Put differently, a stratum is a strongly connected component of the dependency graph defined by $\ltimes$. The relation $\ltimes$ induces a partial ordering on the set of all strata S, that is, for $X, Y \in S$, it is the case that $X \leq Y$ if and only if there exists $\mathtt{p} \in X$ and $\mathtt{q} \in Y$ such that $\mathtt{p} \ltimes^+ \mathtt{q}$.

A stratum forms a logical unit to which the least fixed point semantics and aggregation are applied in turn: For each stratum $X \in S$, we can define its semantics $M_X \subseteq \mathcal{P}(H_P)$ as follows: $M_X = \rho(\bigvee \eta(\mathsf{lfp}(\widehat{T}_Q)))$, where $Q = P_X \cup \bigcup_{Y<X} M_Y$ and P_X is the set $\mathsf{ground}(P)$ restricted to the predicates in the stratum X, while $\bigcup_{Y<X} M_Y$ should be understood as a set of facts. Informally, this means that to give a semantics for a stratum, we first

compute the semantics of the strata below, use the results as a set of facts added to the part of the program in the current stratum, compute the fixed point, and finally perform the aggregation step using the join operator. There are always finitely many strata, so M_X is well-defined. The semantics of the program P is then the aggregation of the sum of the interpretations of all the strata, that is, $\bigcup_{X \in S} M_X$.

Stratification ensures that the answers for a predicate are always aggregated before they are used by another predicate, unless there is a cyclic dependency between them. for example, consider the following variation on the shortest path program:

```
:- table p(index,index,min).
e(1,2). e(2,3). e(1,3).
p(X,Y,1) :- e(X,Y)
p(X,Y,D) :- p(X,Z,D1),p(Z,Y,D2), D is D1 + D2.
s(X,Y,D) :- p(X,Y,D).
```

Without stratification, the semantics is given by $\rho(\bigvee \eta(\mathsf{lfp}(\widehat{T}_P))$ which contains `s(1,3,2)`, as a consequence of `p(1,3,2)` before aggregation. However, if we stratify the program as discussed above, the rules for `p` end up in a lower stratum than `s`. Therefore, the results for `p` will be aggregated by `min`, before any consequence is derived from it. Since `p(1,3,2)` is subsumed, `s(1,3,2)` is no longer derived.

When two predicates are interdependent (and therefore in the same stratum), but only one of them is tabled, the answers for the untabled predicate are not subsumed. Stratification then gives a result that appears inconsistent:

```
:- table even(min).

even(0).
even(X) :- odd(Y), Y is X - 1.
odd(X) :- even(Y), Y is X - 1.
```

Our semantics interprets this program as the set

$$\rho(\bigvee \eta(\mathsf{lfp}(\widehat{T}_P))) = \{\mathtt{even}(0), \mathtt{odd}(1), \mathtt{odd}(3), \mathtt{odd}(5), \mathtt{odd}(7), \ldots\}.$$

While $\{\mathtt{even}(0), \mathtt{odd}(1)\}$ seems equally reasonable. Because we treat subsumption as a post-processing step *per stratum*, which means that inter-dependent predicates are resolved as if no answers were subsumed. Subsumption only affects predicates in the strata above. For instance, assume we add the following (non-tabled) predicate to the program:

```
also_odd(X) :-  even(Y), Y is X - 1.
```

It is in a different stratum than `even` and `odd`, so its semantics depends on the semantics of `even` after the subsumption step. This means that the semantics together with the `also_odd` predicate is given as:

$$\{\mathtt{even}(0), \mathtt{also_odd}(1), \mathtt{odd}(1), \mathtt{odd}(3), \mathtt{odd}(5), \mathtt{odd}(7), \ldots\},$$

Here `also_odd` behaves like the alternative suggested for `odd` above. Importantly, programs like the one above do not satisfy our correctness condition for the greedy strategy given in Section 4.

4 Generalised Answer Subsumption Semantics

The previous section specifies the semantics of tabling with lattice-based answer subsumption in terms of a post-processing aggregation. However, the existing implemen-

tations do not actually first compute the least Herbrand model. Instead, they greedily aggregate intermediate results during SLD-resolution. This makes it feasible to, for instance, compute the shortest path in a cyclic graph in a finite amount of time, as well as generally improving efficiency. However, as the examples in the introduction acutely demonstrate, this greedy strategy is not always valid. This section characterises the greedy strategy as a form of generalised semantics for logic programs and considers its correctness with respect to our postulated specification.

4.1 Generalised Semantics

Again, we assume that we work in a single stratum. We can capture alternate greedy strategies as *generalised semantics* of P in terms of structures $\langle L, \eta^L, T_P^L \rangle$, where:

1. L is a complete lattice,
2. $\eta^L : H_P \to L$ is a function that 'embeds' terms in the lattice,
3. $T_P^L : L \to L$ (a *generalised* immediate consequence operator) is monotone,
4. L is generated by $\eta^L(H_P)$, which means that for every $x \in L$, there exists an $X \subseteq \eta^L(H_P)$ such that $x = \bigvee X$.

Note that in general we do not need a counterpart of the ρ function: in the post-processing semantics given in Section 3, we need ρ to define the $\widehat{T}_P$ operator and move between strata. Here, we use this semantics to capture correctness within a single stratum, and the immediate consequence operator is not defined, but it is given. This allows us to generalise the whole table to a lattice, which simplifies the generalised semantics.

The generalised semantics of the program P is given by $\mathsf{lfp}(T_P^L)$, which exists due to the Knaster–Tarski theorem. It is easy to verify that the regular least fixed-point semantics is a valid instance of this generalised semantics. Also, note that the generalised semantics does not depend on η^L or the fact that L is generated by η^L. We need these facts in a moment to establish a correctness criterion.

One obvious instantiation of this semantics is with the $\widehat{T}_P$ operator defined in Section 3.3, where $\iota : X \to \mathcal{P}(X)$ is the singleton function:

$$\langle \mathcal{P}(H_P), \iota, \widehat{T}_P \rangle$$

We say that an instance of the generalised semantics is a correct implementation strategy only if yields the same result as the post-processing semantics defined in Section 3. More formally, a generalised semantics $\langle L, \eta^L, T_P^L \rangle$ is *correct* if and only if

$$\mathsf{lfp}(T_P^L) = [\eta^L](\mathsf{lfp}(\widehat{T}_P)), \tag{4}$$

where, for convenience, we define, for any lattice L, set S, and function $f : S \to L$, the function $[f] : \mathcal{P}(S) \to L$ as follows:

$$[f](Y) = \bigvee_{x \in X} f(x)$$

Intuitively, it means that evaluating the whole program with no answer subsumption and then obtaining the final result using L's join operation on the answers is the same as computing every step with the lattice L (which is usually more efficient). The following theorem gives sufficient conditions for the equation (4) to hold.

Theorem 4.1 (*Fixed-Point Fusion (Backhouse 2000)*)
Let $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ be posets. Let $f : X \to X$ and $g : Y \to Y$ be two functions with least fixed points. Let $h : X \to Y$ be a function. Assume the following:

(a) It is the case that $h \circ f = g \circ h$.

(b) The function h has an upper Galois adjoint, that is, there exists a function $j : Y \to X$ such that $h(x) \leq_Y y \iff x \leq_X j(y)$ for all $x \in X$ and $y \in Y$.

Then, $\mathsf{lfp}(g) = h(\mathsf{lfp}(f))$.

The equation (4) is clearly an instance of this theorem's conclusion, with $f = \widehat{T}_P$, $g = T_P^L$ and $h = [\eta^L]$. Yet, the theorem only applies if we can show that (a) $[\eta^L] \circ \widehat{T}_P = T_P^L \circ [\eta^L]$, and that (b) $[\eta^L]$ has an upper Galois adjoint. Fortunately, (b) readily follows from the fact that $[\eta^L]$ is continuous, and that every continuous function has an upper Galois adjoint (Backhouse 2000).

Intuitively the remaining condition (a) means that $[\eta^L]$ should preserve immediate consequences. In other words, subsumption of a conventional immediate consequence should be the generalised immediate consequence of $[\eta^L]$.

4.2 Greedy Strategy

For better performance, practical implementations of tabling with answer subsumption use a greedy strategy, which means that they remove subsumed answers as soon as possible, and not as a single post-processing step. We can try to express it as the following instance of the generalised semantics, in terms of the functions defined in Section 3.3:

$$\langle I_P \times U_P^n \to L, \eta, [\eta] \circ T_P \circ \rho \rangle$$

The function ρ extracts the set of true ground atoms from a table, which is a subset of the Herbrand base H_P, so we can indeed express the 'greedy' immediate consequence operator in terms of the standard immediate consequence operator T_P, subsumption $[\eta]$ and extraction ρ. In fact this definition is quite convenient, because tabled Prolog systems readily provide an efficient implementation of T_P.

The question remains if the function $[\eta] \circ T_P \circ \rho$ has a fixed point. Luckily, given a tuple $\langle L, \eta^L, T_P^L \rangle$, where L is generated by $\eta^L(H_P)$, the condition (a) is enough for T_P^L to be monotone (this is where the assumption that $\eta^L(H_P)$ generates L comes in handy):

Theorem 4.2
Let $\langle L, \eta^L, T_P^L \rangle$ be as above. If $[\eta^L] \circ \widehat{T}_P = T_P^L \circ [\eta^L]$ (the condition (a)), then T_P^L is monotone.

Additionally, it is the case that $[\eta] \circ T_P = [\eta] \circ \widehat{T}_P$. Thus, to show correctness of a program under the greedy semantics, we need only show the following:

$$[\eta] \circ T_P = [\eta] \circ T_P \circ \rho \circ [\eta] \tag{5}$$

Example 4 Reconsider the invalid program from the introduction. We show that the condition 5 does not hold for this program under semantics $\langle L(\preccurlyeq), \preceq, \eta, T_P^{L(\preccurlyeq)} \rangle$ where $H_P = \{\mathtt{p(0)}, \mathtt{p(1)}, \mathtt{p(2)}, \mathtt{p(3)}\}$ and $(\preccurlyeq) = (\leq)$ is the partial order on terms. By means of

the following counter example: A simple calculation shows that $([\eta]\circ T_P)\big(\{\texttt{p(0)},\texttt{p(1)}\}\big) \neq ([\eta]\circ T_P\circ\rho\circ[\eta])\big(\{\texttt{p(0)},\texttt{p(1)}\}\big)\big)$.

This example also explains the odd behaviour of XSB, Yap and B-Prolog: `p(0)` is subsumed by `p(1)`, therefore the body of the third rule in the program cannot be satisfied and `p(3)` (the correct answer) is never produced.

Interestingly, when we change the last rule to `p(3) :- p(0)`, the result of the query `?-p(X).` changes once more in all systems, although logically both rules should behave identically. Furthermore, different implementations handle this rule differently. For instance, XSB reasonably disallows calls where `lattice`-mode arguments are not free, and the latter rule therefore produces an error message. While Yap instead produces `X=3`, because its batched-mode evaluation immediately derives `p(1)` from the fact `p(1)`.

Example 5 Now, consider the shortest path program from Example 2 under semantics $\langle L(\preccurlyeq), \preceq, \eta, T_P^{L(\preccurlyeq)}\rangle$ and $(\preccurlyeq) = (\geq)$. We prove the correctness condition by proving that $lhs \preceq rhs$ and $lhs \succeq rhs$. Then by anti-symmetry of $\preceq$, the correctness condition holds.
($\succeq$-direction) Because $[\eta]\circ T_P$ is monotone, and for this case $\rho\circ[\eta]$ is deflative, i.e. $\rho([\eta](X)) \subseteq X$ for all $X \subseteq H_P$, we are done with this direction.
($\preceq$-direction) A `p(x,y,d)`-atom is either introduced by an edge `e(x,y,nt)` or by two other atoms `p(x,z,d1)` and `p(z,y,d2)`. In the former case $d = 1$, in the latter case $d = d_1 + d_2$. It is easy to see that if an atom `p(x,y,1)` is introduced by T_P, it is also introduced by $T_P\circ\rho\circ[\eta]$. In the latter case, d_1 and d_2 are at least as large as their corresponding entries in the table produced by $[\eta]$. Hence, d must be at least as large as any corresponding distance in the set produced by $T_P\circ\rho\circ[\eta]$. Hence, the infimum of ds produced by T_P must be at least as large as any corresponding distance. Finally, this means that the entry in the table produced by $[\eta]\circ T_P$ must be at least as large as the entry in the table produced by $[\eta]\circ T_P\circ\rho\circ[\eta]$.

5 Related Work

As far as we know, "output" subsumption for tabling was first proposed by Van Hentenryck et al. (1993) in the context of abstract interpretation.

Tabling Modes for Dynamic Programming In dynamic programming, an optimal solution to a problem is defined in terms of the optimal solutions of smaller sub-problems. This intuition is in fact captured by the correctness condition (Equation 5): it states that the solution found by examining all sub-solutions, is equal to examining only the optimal solutions. This is good news, because it implies correctness for dynamic programming.

We have already discussed the tabling modes of Yap (Santos and Rocha 2013) and XSB (Swift and Warren 2012) at length in Section 3.1. XSB's lattice based answer subsumption is more suitable for implementing techniques that require more general lattices than simple minimum and maximum, such as abstract interpretation.

Guo and Gupta (2004; 2008) implemented 5 tabling modes in ALS-Prolog with the aim of simplifying and accelerating dynamic programming. These modes are `+` (indexed), `-`(only the first answer is retained), `min` (minimum), `max` (maximum) and `last` (the last answer is retained). The correspondence to Yap's tabling modes is obvious, but there is

no equivalent for Yap's `all` and `sum` modes. Answers are grouped by distinct values for the `+` arguments and the remaining arguments are aggregated based on their mode.

Zhou et al. (2010) added tabling to B-Prolog with the same purpose in mind. Consequently, the tabling modes they support are identical, except they do not implement a `last` mode. Instead, they support cardinality constraints, which limit the answers that are stored in the table to the first N optimal answers for some positive integer N. Also supported is an `nt` (not-tabled) tabling mode, which is used to pass around global constants efficiently. From the perspective of the tabling system `nt` arguments do not exist, and thus are never stored in the table.

Haskell Vandenbroucke et al. (2016) have added lattice answer subsumption to their tabling implementation in Haskell. It is based on the effect handlers approach.

Abstract interpretation Our approach bears a strong resemblance to abstract interpretation (Cousot and Cousot 1992; Abramsky and Hankin 1987). Unlike answer subsumption, abstract interpretation admits approximate solutions, implying a weaker correctness condition where equality is replaced by an order relation.

Matroids and Greedoids Other set theoretic structures (besides lattices) such as *matroids* (Oxley 1992) and *greedoids* (Korte et al. 1991), have been developed to analyse greedy algorithms and show their optimality. As answer subsumption is essentially a greedy strategy, we plan to re-examine answer subsumption in this new context in the future.

6 Conclusion and Future Work

In many instances of tabling only the optimal answers to a query are relevant. To improve performance over a naive generate-and-aggregate approach, various forms of answer subsumption that greedily combine these answers have been developed in the literature. However, their semantics has never been described formally. An operational understanding is always an option in this case, and although often useful, it is a far cry from the declarative ideal that tabling promises.

We define a high-level semantics for answer subsumption based on lattice theory. Then we generalise it to establish a correctness condition indicating when it is safe to use (greedy) answer subsumption. We show several examples where the existing implementations of answer subsumption fail that condition and derive an erroneous result.

This condition is sufficient, but not necessary: there may still exist programs that do not satisfy the condition, for which the greedy strategy nevertheless delivers correct results. Since we have not run across any non-contrived examples of such programs, we contend that this apparent lack of necessity is an artefact of our rather coarse semantics, which we intend to refine in future work.

The verification of correctness does constitute a non-trivial effort. Hence, manually proving the correctness condition for realistically sized programs could be unfeasible in practice. Ideally we would have an automated analysis that warns the programmer if it fails to establish the correctness condition. This is future work.

Acknowledgements We would like to thank Bart Demoen for enlightening discussions during the preparation of this paper. This research was partially funded by the Flemish Fund for Scientific Research (FWO).

References

Abramsky, S. and Hankin, C. 1987. *Abstract Interpretation of declarative languages.* Vol. 1. Ellis Horwood, Chapter An introduction to abstract interpretation, 63–102.

Apt, K. R., Blair, H. A., and Walker, A. 1988. *Towards a Theory of Declarative Knowledge.* Morgan Kaufmann.

Backhouse, R. C. 2000. Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, International Summer School and Workshop, Oxford, UK, April 10-14, 2000, Revised Lectures*, R. C. Backhouse, R. L. Crole, and J. Gibbons, Eds. LNCS, vol. 2297. Springer, 89–148.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., Silva, C., and Rocha, R. 2008. An improved continuation call-based implementation of tabling. In *Practical Aspects of Declarative Languages, 10th International Symposium.* LNCS, vol. 4902. Springer, 197–213.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *The Journal of Logic Programming 13,* 23, 103 – 179.

Guo, H.-F. and Gupta, G. 2004. Simplifying dynamic programming via tabling. In *Practical Aspects of Declarative Languages.* LNCS, vol. 3057. Springer, 163–177.

Guo, H.-F. and Gupta, G. 2008. Simplifying dynamic programming via mode-directed tabling. *Software: Practice and Experience 38,* 1, 75–94.

Korte, B., Lovász, L., and Schrader, R. 1991. Greedoids, algorithms and combinatorics, vol. 4.

Lloyd, J. W. 1984. *Foundations of Logic Programming.* Springer-Verlag, New York.

MacNeille, H. M. 1937. Partially ordered sets. *Transactions of the American Mathematical Society*, 416–460.

Oxley, J. G. 1992. *Matroid theory.* Oxford University Press.

Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Swift, T., and Warren, D. S. 1997. *Computer Aided Verification: 9th International Conference, Haifa, Israel, June 22–25, 1997 Proceedings.* Springer, 143–154.

Santos, J. and Rocha, R. 2013. On the efficient implementation of mode-directed tabling. In *Practical Aspects of Declarative Languages.* LNCS, vol. 7752. Springer, 141–156.

Santos Costa, V., Rocha, R., and Damas, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming 12,* 1-2, 5–34.

Swift, T. 1999. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 201–240.

Swift, T. and Warren, D. 2010. Tabling with answer subsumption: Implementation, applications performance. 300–312.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming 12,* 1-2 (Jan.), 157–187.

Van Hentenryck, P., Degimbe, O., Charlier, B. L., and Michel, L. 1993. Abstract interpretation of Prolog based on OLDT resolution. Tech. rep., Providence, RI, USA.

Vandenbroucke, A., Schrijvers, T., and Piessens, F. 2016. Fixing non-determinism. In *Proceedings of the 27th symposium on Implementation and Application of Functional Languages 2015.*

Zhou, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 189–218.

Zhou, N. F. and Dovier, A. 2011. A tabled prolog program for solving sokoban. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence.* 896–897.

Zhou, N.-F., Kameya, Y., and Sato, T. 2010. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *22nd International Conference on Tools with Artificial Intelligence (ICTAI), 2010.* Vol. 2. 213–218.