TABDUAL: A TABLED ABDUCTION SYSTEM FOR LOGIC PROGRAMS

ARI SAPTAWIJAYA[*]
LUÍS MONIZ PEREIRA
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
Departamento de Informática, Universidade Nova de Lisboa, Portugal
`ar.saptawijaya@campus.fct.unl.pt, lmp@fct.unl.pt`

Abstract

Abduction has been on the back burner in logic programming, as abduction can be too difficult to implement, and costly to perform, in particular if abductive solutions are not tabled. On the other hand, current Prolog systems, with their tabling mechanisms, are mature enough to facilitate the introduction of tabling abductive solutions (tabled abduction) into them.

Our contributions are as follows. First, we conceptualize a tabled abduction technique for abductive normal logic programs, permitting abductive solutions to be reused, from one abductive context to another. The approach is underpinned by the theory of ABDUAL and relies on a transformation into tabled logic programs. It particularly makes use of the dual transformation of ABDUAL that enables efficiently handling the problem of abduction under negative goals, by introducing dual positive counterparts for them. Second, we realize this tabled abduction technique in TABDUAL, a system implemented in XSB Prolog. The implementation poses several challenges to concretely realize the abstract theory of ABDUAL, e.g., by taking care of all varieties of loops (positive loops and loops over negation) in normal logic programs, now complicated by tabled abduction. Other challenges are pertinent to optimizations, by benefitting from XSB features, e.g., constructing dual rules by need only. Third, we evaluate TABDUAL with respect to various standpoints. The evaluations employ cases from declarative debugging, and also touch upon tabling *nogoods* of subproblems in the context of abduction.

The techniques introduced in TABDUAL intends to sensitize a general audience of users, and of implementers of various LP systems, to the potential benefits of tabled abduction, where a number of its techniques are also adaptable and importable into LP systems that afford tabling mechanisms, other than XSB Prolog.

We thank the referees for their constructive comments and suggestions. Ari Saptawijaya acknowledges the support of FCT/MEC Portugal, grant SFRH/BD/72795/2010. We thank Terrance Swift and David Warren for their expert advice in dealing with implementation issues in XSB Prolog.

[*]Affiliated also with the Faculty of Computer Science at Universitas Indonesia, Depok, Indonesia.

1 Introduction

Abduction has been well studied in the field of computational logic, and logic programming (LP) in particular, for a few decades by now [4, 11, 14, 17, 20, 22, 44]. Abduction in LP offers a formalism to declaratively express problems in a variety of areas, e.g. in diagnosis, planning, scheduling, reasoning of rational agents, decision making, knowledge assimilation, natural language understanding, security protocols verification, and systems biology [1, 6, 16, 18, 23–25, 31, 34]. On the other hand, many Prolog systems have become mature and practical, and thus it makes sense to facilitate the use of abduction into such systems, be it two-valued abduction (as adopted in this work) or three-valued, e.g. [9].

In abduction, finding some best explanations (i.e. adequate abductive solutions) to the observed evidence, or finding assumptions that can justify a goal, can be very costly. It is often the case that abductive solutions found within one context are also relevant in a different context, and can be reused with little cost. In LP, absent of abduction, goal solution reuse is commonly addressed by employing a tabling mechanism [46]. Therefore, tabling appears to be conceptually suitable for abduction, so as to reuse abductive solutions. In practice, abductive solutions reuse is not immediately amenable to tabling, because such solutions go together with an abductive context.

In [32], we preliminarily explore the idea of how to benefit from tabling mechanisms in order to reuse priorly obtained abductive solutions, from one abductive context to another. This technique of tabling abductive solutions (*tabled abduction*) is underpinned by ABDUAL [4], a theory for computing abduction over Well-Founded Semantics. The technique presented in [32] consists of a program transformation from abductive normal logic programs into tabled logic programs, and it specifically employs the dual transformation of ABDUAL. The dual transformation allows to more efficiently handle the problem of abduction under negative goals, by introducing their positive dual counterparts.

In this paper, we formalize the tabled abduction transformation in [32], while also simplifying it, by abstracting away from implementation details, such as dealing with loops (i.e. positive loops and loops over negation) in abductive normal logic programs, non-grounded programs, etc. In other words, the present transformation focuses on an innovative re-uptake of prior abductive solution entries in tabled predicates as well as the dual transformation of ABDUAL, on which it relies.

Based on the formalization, we develop a tabled abduction system TABDUAL. The implementation of TABDUAL poses several challenges [36], both in concretely realizing the abstract theory of ABDUAL and in benefitting from features of XSB Prolog [48], in which TABDUAL is implemented, for optimizations:

1. In the theory of ABDUAL, the dual transformation does not concern itself with programs having variables. Without violating the groundness assumption in the the-

ory of ABDUAL, the TABDUAL implementation takes care of such programs. More precisely, TABDUAL helps ground (dualized) negative subgoals and deals with non-ground negative goals. Note that, in dealing with non-ground negative goals, we look just for abductive solutions of such non-ground negative goals, and not for constraints on free variables of its calling arguments, i.e., no constructive negation is applied.

2. The tabling mechanism in XSB Prolog supports Well-Founded Semantics [51], and allows dealing with loops in the program to ensure the termination of looping queries. The implementation of TABDUAL employs XSB's tabling as much as possible to deal with loops. Nevertheless, tabled abduction introduces a complication concerning some varieties of loops. This complication is resolved by some pragmatic approaches using available tabling constructs in XSB, such as tabled negation.

3. TABDUAL allows modular mixes of abductive and non-abductive program parts, and one can benefit from the latter part by enacting a simpler translation of predicates in the program being comprised just of facts. This simpler treatment distinguishes the transformation between rules in general and predicates defined extensionally by facts alone. It particularly helps avoid superfluous transformation of facts, which would hinder the use of large factual data.

4. We address the issue of potentially heavy transformation load due to producing all dual rules in advance, regardless of their need. Such a heavy dual transformation makes it a bottleneck of the whole abduction process. A natural solution is instead to perform the dual transformation *by-need*, i.e. dual rules for a predicate are only created as their need is felt during abduction. We detail two approaches to realizing the dual transformation by-need: (a) by creating and tabling all dual rules for a predicate on the first invocation of its negation; and (b) by lazily generating and storing its dual rules in a trie (instead of tabling them), as new alternatives are required. The former approach leads to an *eagerly by-need* dual rules tabling (under local table scheduling strategy), whereas the latter permits a *lazily by-need* dual rules construction (in lieu of batched table scheduling).

5. TABDUAL provides a system predicate that permits accessing ongoing abductive solutions. This is a useful feature and extends TABDUAL's flexibility, as it allows manipulating abductive solutions dynamically, e.g. preferring or filtering ongoing abductive solutions, e.g. checking them explicitly against nogoods at predefined program points.

TABDUAL has been evaluated with various objectives, where several TABDUAL variants (of the same underlying implementation) are examined, by separately factoring out TABDUAL's features relevant to each evaluation objective [38]. First, we evaluate the benefit of tabling abductive solutions, where we employ an example from declarative debugging to debug missing solutions of logic programs, via a process now characterized as abduction [39], instead of as belief revision [29, 30]. Second, we use the other case of declarative debugging, that of debugging incorrect solutions, to evaluate the relative worth of the dual transformation by-need. Third, we touch upon tabling so-called *nogoods* of subproblems in the context of abduction (i.e. abductive solution candidates that violate constraints), and show that tabling abductive solutions can be appropriate for tabling nogoods of subproblems. Fourth and finally, we also evaluate TABDUAL in dealing with programs having loops, where we compare its results with those obtained from an implementation of ABDUAL [5].

TABDUAL is an ongoing work, which primarily intends to sensitize a general audience of users, and of implementers of various LP systems, to the potential benefits of tabled abduction. Though TABDUAL is implemented in XSB Prolog, a number of its techniques are adaptable and importable into other LP systems that afford required tabling mechanisms. They add and aid to the considerations involved in the research of the still ongoing developments of tabling mechanisms in diverse LP systems, and serve to inspire these systems in terms of solutions, options and experimentation results of incorporating tabled abduction.

The rest of the paper is organized as follows. Section 2 provides basic definitions in LP and how abduction is accomplished in LP. Tabled abduction and the formalization of the TABDUAL transformation are presented in Section 3. We justify soundness and completeness of TABDUAL, based on the theory of ABDUAL, in Section 4, and discuss its complexity in Section 5. The aforementioned implementation aspects of TABDUAL are detailed in Section 6. Section 7 exhibits and analyses the evaluation results of TABDUAL. We conclude in Section 8, by discussing related work and further developments, including their potential joint use with other non-monotonic LP features, having their own tabling requirements and attending benefits.

2 Preliminaries

A *logic rule* has the form $H \leftarrow B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$, where $n \geq m \geq 0$ and H, B_i with $1 \leq i \leq n$ are atoms; H and $B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$ are called the head and the body of the rule, resp. We use 'not' to denote default negation. The atom B_i and its default negation $not\ B_i$ are named positive and negative *literals*, resp. When $n = 0$, we say the rule is a *fact*, simply written as H. The atoms *true* and *false* are, by definition, respectively true and false in every interpretation. A rule in the form of a denial, i.e. with empty head, or equivalently with *false* as head, is an *integrity constraint* (IC). A *logic program* is a set of logic rules, where non-ground rules (i.e. rules containing variables) stand for all their ground instances. We focus on *normal logic programs*, i.e. those whose heads of rules are positive literals or empty. As usual, p/n denotes predicate p with arity n.

Abduction (inference to the best explanation – a common designation in the philosophy of science [21, 26]), is a reasoning method, whereby one chooses those hypotheses that would, if true, best explain the observed evidence (satisfy some query), while meeting any attending ICs. In LP, abductive hypotheses (*abducibles*) are named positive or negative literals of the program, which have no rules, and whose truth value is not initially assumed. Abducibles may have arguments, but for simplicity they must be ground when abduced. An *abductive normal logic program* is a normal logic program that allows abducibles appearing in the body of rules. Note that abducible '$not\ a$' does not refer to the default negation of abducible a, as abducibles do not appear in the head of a rule, but instead to the explicitly assumed hypothetical negation of a. The truth value of abducibles may be independently assumed *true* or *false*, via either their positive or negated form, as the case may be, to produce an abductive solution to a query, i.e. a consistent set of assumed hypotheses that support it. An *abductive solution* to a query is a consistent set of abducible instances that, when substituted by their assigned truth value everywhere in the program P, affords us with a model of P (for the specific semantics used on P), which satisfies both the query and the ICs – a so-called *abductive model*.

Abduction in LP can naturally be accomplished by a top-down query-oriented procedure to find an (abductive) solution to a query (by-need, i.e. as abducibles are encountered), where the abducibles in the solution are leaves in its procedural query-rooted call-graph, i.e. the graph recursively engendered by the procedure calls from literals in bodies of rules to heads of rules, and thence to the literals in the rule's body. This top-down computation is possible only when the underlying semantics is relevant, i.e. avoids having to computing a whole model (to guarantee its existence) in order to find an answer to a query: it suffices to use only the rules relevant to the query – those in its procedural call-graph – to find its truth value. The Well-Founded Semantics (WFS) [51] enjoys the relevance property, and thus it allows abduction to be performed by need. This is induced by the top-down query-oriented procedure, solely for finding the relevant abducibles and their truth value, whereas the values of abducibles not mentioned in the abductive solution are indifferent to the query. Tabled abduction and its prototype TABDUAL is underpinned by the theory of ABDUAL [4] that computes abduction over WFS. Note that though WFS is three-valued, the abduction mechanism in TABDUAL enforces, by design, two-valued abductive solutions; that is, needed abducibles are assumed either true or false, so as not to contribute with undefinedness towards the query.

3 Tabled abduction in TABDUAL

We start by giving the motivation for the need of tabled abduction, and subsequently show how tabled abduction is conceptualized and realized in the TABDUAL transformation.

3.1 Motivation

Example 1. *Consider an abductive logic program P_0, with a and b abducibles:*

$$q \leftarrow a. \qquad s \leftarrow b, q. \qquad t \leftarrow s, q.$$

Suppose three queries: q, s, and t, are individually launched, in that order. The first query, q, is satisfied simply by taking $[a]$ as the abductive solution for q, and tabling it. Executing the second query, s, amounts to satisfying the two subgoals in its body, i.e. abducing b followed by invoking q. Since q has previously been invoked, we can benefit from reusing its solution, instead of recomputing, given that the solution was tabled. That is, query s can be solved by extending the current ongoing abductive context $[b]$ of subgoal q with the already tabled abductive solution $[a]$ of q, yielding $[a, b]$. The final query t can be solved similarly. Invoking the first subgoal s results in the priorly registered abductive solution $[a, b]$, which becomes the current abductive context of the second subgoal q. Since $[a, b]$ subsumes the previously obtained (and tabled) abductive solution $[a]$ of q, we can then safely take $[a, b]$ as the abductive solution to query t. This example shows how $[a]$, as the abductive solution of the first query q, can be reused from one abductive context of q (i.e. $[b]$ in the second query, s) to its other context (i.e. $[a, b]$ in the third query, t). In practice the body of rule q may contain a huge number of subgoals, causing potentially expensive recomputation of its abductive solutions and thus such unnecessary recomputation should be avoided.

Tabled abduction in TABDUAL consists of two stages: program transformation and abduction. The program transformation produces tabled logic programs from abductive normal logic programs. Abduction is then enacted on the transformed program. Example 1 indicates two key ingredients of the transformation:

1. *abductive context*, which relays the ongoing abductive solution from one subgoal to subsequent subgoals, as well as from the head to the body of a rule, via *input* and *output* contexts, where abducibles can be envisaged as the terminals of parsing,
2. *tabled predicates*, which table the abductive solutions for predicates defined in the input program, such that they can be reused from one abductive context to another.

3.2 TABDUAL transformation

The TABDUAL transformation is underpinned by the theory of ABDUAL [4], but additionally employs the aforementioned idea of tabling and of reusing abductive solutions.

The whole TABDUAL transformation (Section 3.2.4) consists of several parts, viz., the transformations for tabling abductive solutions, for producing dualized negation, and for inserting abducibles into an abductive context. Their specifications are formalized in Sections 3.2.1, 3.2.2, and 3.2.3, respectively. Finally, queries should also be transformed, as detailed in Section 3.2.5.

3.2.1 Tabling abductive solutions

We show in Example 2, how the idea described in Example 1 can be realized by the program transformation. It illustrates how every rule in P_0 is transformed, by introducing a corresponding tabled predicate with one extra argument for the abductive solution entry, such that it can facilitate solution reuse from one abductive context to another.

Example 2. *We show first how the rule $t \leftarrow s, q$ in P_0 is transformed into two rules:*
$t_{ab}(E_2) \leftarrow s([\,], E_1), q(E_1, E_2).$ $\qquad$ $t(I, O) \leftarrow t_{ab}(E), produce\_context(O, I, E).$
Predicate $t_{ab}(E)$ is the tabled predicate which is introduced to table one abductive solution for t in its argument E. Its definition, in the rule on the left, follows from the original definition of t. Two extra arguments, that serve as input and output contexts, are added to the subgoals s and q in the rule's body. The left rule expresses that the tabled abductive solution E_2 of t_{ab} is obtained by relaying the ongoing abductive solution stored in context E_1 from subgoal s to subgoal q in the body, given the empty input abductive context of s (because there is no abducible by itself in the body of the original rule of t). The rule on the right shows how the tabled abductive solution in E of t_{ab} can be reused for a given (input) abductive context of t. This rule expresses that the output abductive solution O of t is obtained from the solution entry E of t_{ab} and the given input context I of t, via the TABDUAL *system predicate $produce\_context(O, I, E)$. This system predicate concerns itself with: whether E is already contained in I and, if not, whether there are any abducibles from E, consistent with I, that can be added to produce O. If E is inconsistent with I then the specific entry E cannot be reused with I, $produce\_context/3$ fails and another entry E is sought. In other words, $produce\_context/3$ should guarantee that it produces a consistent output context O from I and E that encompasses both.*

The other two rules in P_1 are transformed following the same idea. The rule $s \leftarrow b, q$ is transformed into:

$s_{ab}(E) \leftarrow q([b], E).$ $\qquad$ $s(I, O) \leftarrow s_{ab}(E), produce\_context(O, I, E).$

where $s_{ab}(E)$ is the predicate that tables, in E, the abductive solution of s. Notice how b, the abducible appearing in the body of the original rule of s, becomes the input abductive context of q. The same transformation is obtained, even if b comes after q in the body of the rule s.

Finally, the rule $q \leftarrow a$ is transformed into:

$q_{ab}([a]).$ $\qquad$ $q(I, O) \leftarrow q_{ab}(E), produce\_context(O, I, E).$

where the original rule of q, which is defined solely by the abducible a, is simply transformed into the tabled fact $q_{ab}/1$.

Example 3. *Consider the following program that contains rules of non-nullary predicate $q/1$ with variables ($a/1$ abducible):*

$$q(0). \qquad q(s(X)) \leftarrow a(X), q(X).$$

The transformation results in rules as follows:

$$q_{ab}(0, [\,]). \qquad\qquad q_{ab}(s(X), E) \leftarrow q(X, [a(X)], E).$$
$$q(X, I, O) \leftarrow q_{ab}(X, E), produce\_context(O, I, E).$$

Notice that the single argument of $q/1$ is kept in the tabled predicate q_{ab} (as its first argument), and one extra argument is added (as its second argument) for tabling its abductive solution entry. The transformed rules $q_{ab}/2$ and $q/3$ are defined following the same idea described in Example 2.

The transformation for tabling abductive solutions is formalized in Definition 1.

Consider an abductive normal logic program P, where every integrity constraint in P with empty head is rewritten as a rule with *false* as its head, i.e. as a denial. In the sequel, we write $\bar{t}$ to denote $[t_1, \ldots, t_n]$, $n \geq 0$, and for predicate p/n, we write $p(\bar{t})$ to denote $p(t_1, \ldots, t_n)$,[1] and we write H_r and $\mathcal{B}_r$ to denote the head and the body of rule $r \in P$, respectively. Mark that abducibles do not have rules.

Definition 1 (Transformation for tabling abductive solutions). *Let $\mathcal{A}_r \subseteq \mathcal{B}_r$ be the set of abducibles (either positive or negative) in $r \in P$, and r' be the rule, such that $H_{r'} = H_r$ and $\mathcal{B}_{r'} = \mathcal{B}_r \setminus \mathcal{A}_r$.*

1. *For every rule $r \in P$ with r' the rule $l(\bar{t}) \leftarrow L_1, \ldots, L_m$, we define $\tau'(r)$:*

   $$l_{ab}(\bar{t}, E_m) \leftarrow \alpha(L_1), \ldots, \alpha(L_m).$$

   where α is defined as:

   $$\alpha(L_i) = \begin{cases} l_i(\bar{t}_i, E_{i-1}, E_i) & \text{, if } L_i = l_i(\bar{t}_i) \\ not\_l_i(\bar{t}_i, E_{i-1}, E_i) & \text{, if } L_i = not\ l_i(\bar{t}_i) \end{cases}$$

   with $1 \leq i \leq m$, E_i are fresh rule variables,[2] *and $E_0 = \mathcal{A}_r$.*

2. *For every predicate p/n defined in P, we define $\tau^+(p)$:*

   $$p(\bar{X}, I, O) \leftarrow p_{ab}(\bar{X}, E),\ produce\_context(O, I, E).$$

   where $produce\_context/3$ is a TABDUAL *system predicate.*

Example 4. *Consider the following program P, where rules are named with r_i and $a/1$ is an abducible.*

$$\begin{array}{ll} r_1: & u(0, \_). \\ r_2: & u(s(X), Y) \leftarrow a(X), v(X, Y, Z), not\ w(Z). \\ r_3: & v(X, X, s(X)). \end{array}$$

[1] In particular, we write $\bar{X}$ to denote $[X_1, \ldots, X_n]$, $p(\bar{X})$ to denote $p(X_1, \ldots, X_n)$, and $p(\bar{X}, Y, Z)$ to denote $p(X_1, \ldots, X_n, Y, Z)$, where all variables are distinct.

[2] Variables E_i serve as abductive contexts.

We have $\mathcal{A}_{r_i}$ and r'_i, for $1 \leq i \leq 3$, as follows:[3]

- $\mathcal{A}_{r_1} = [\,]$ *and* $r'_1 :\ u(0, \_)$.
- $\mathcal{A}_{r_2} = [a(X)]$ *and* $r'_2 :\ u(s(X), Y) \leftarrow v(X, Y, Z), not\ w(Z)$.
- $\mathcal{A}_{r_3} = [\,]$ *and* $r'_3 :\ v(X, X, s(X))$.

The transformation of Definition 1 results in:

$$\begin{array}{lllll}
\tau'(r_1): & u_{ab}(0, \_, [\,]). & & \\
\tau'(r_2): & u_{ab}(s(X), Y, E_2) & \leftarrow & v(X, Y, Z, [a(X)], E_1),\ not\_w(Z, E_1, E_2). \\
\tau'(r_3): & v_{ab}(X, X, s(X), [\,]). & & \\
\tau^+(u): & u(X_1, X_2, I, O) & \leftarrow & u_{ab}(X_1, X_2, E),\ produce\_context(O, I, E). \\
\tau^+(v): & v(X_1, X_2, X_3, I, O) & \leftarrow & v_{ab}(X_1, X_2, X_3, E),\ produce\_context(O, I, E).
\end{array}$$

Notice that both arguments of $u/2$ are kept in the tabled predicate u_{ab} (as its first two arguments), and one extra argument is added (as its third argument) for tabling its abductive solution entry. Similar reasoning also applies to $v/3$. We do not have $\tau^+(w)$, because there is no rule of $w/1$ in the program, i.e. $w/1$ is not defined in P.

3.2.2 Abduction under negative goals

For abducing under negative goals, the program transformation employs the *dual transformation* from ABDUAL [4]. It makes negative goals 'positive' literals, thus permitting to avoid the computation of all abductive solutions of the positive goal argument, and then having to negate their disjunction. The dual transformation enables us to obtain one abductive solution at a time, just as when we treat abduction under positive goals. The dual transformation defines for each atom A and its set of rules R in a normal program P, a set of dual rules whose head not_A is true if and only if A is false by R in the employed semantics of P. Note that, instead of having a negative goal $not\ A$ as the rules' head, we use its corresponding 'positive' literal, not_A. Example 5 illustrates the main idea of how the dual transformation is employed in TABDUAL.

Example 5. *Consider program P_2, where a is an abducible:*

$$p \leftarrow a. \qquad p \leftarrow q, not\ r. \qquad r.$$

- *With regard to p, the transformation will create a set of dual rules for p which falsify p with respect to its two rules, i.e. by falsifying both the first rule* and *the second rule, expressed below by predicate p^{*1} and p^{*2}, respectively:*

[3]We use Prolog list notation to represents sets.

$$not\_p(T_0, T_2) \leftarrow p^{*1}(T_0, T_1),\ p^{*2}(T_1, T_2).$$

In the TABDUAL *transformation, this single rule is known as the first layer of the dual transformation. Note the addition of the input and output abductive context arguments, T_0 and T_2, in the head, and similarly in each subgoal of the rule's body, where intermediate context T_1 relays the ongoing abductive solution from p^{*1} to p^{*2}.*

The second layer contains the definitions of p^{*1} and p^{*2}, where p^{*1} and p^{*2} are defined by falsifying the body of p's first rule and second rule, respectively.

- *In case of p^{*1}: the first rule of p is falsified only by abducing the negation of a. Therefore, we have:*

  $$p^{*1}(I, O) \leftarrow not\_a(I, O).$$

  Notice that the negation of a, i.e. $not\ a$, is abduced by invoking the subgoal $not\_a(I, O)$. This subgoal is defined via the transformation of abducibles, as discussed below.

- *In case of p^{*2}: the second rule of p is falsified by alternatively failing one subgoal in its body at a time, i.e. by negating q or, instead, by negating $not\ r$.*

  $$p^{*2}(I, O) \leftarrow not\_q(I, O). \qquad p^{*2}(I, O) \leftarrow r(I, O).$$

• *With regard to q, the dual transformation produces the fact*

$$not\_q(I, I).$$

as its dual, because there is no rule for q in P_2. Since it is a fact, the content of the context I is simply relayed from the input to the output context, i.e. having no body, the output context does not depend on the context of any other goals, but depends only on its corresponding input context.

• *With regard to r, since it is a fact, its dual contains*

$$not\_r(T_0, T_1) \leftarrow r^{*1}(T_0, T_1).$$

but with no definition of $r^{*1}/2$. It may equivalently be defined as:

$$not\_r(\_, \_) \leftarrow fail$$

Example 5 shows that the dual rules of nullary predicates are simply defined by falsifying the bodies of their corresponding positive rules. But a goal of non-nullary predicates may also fail (or equivalently, its negation succeeds), when its arguments disagree with the arguments of its rules. For instance, if we have just a fact $q(1)$, then goal $q(0)$ will fail (or equivalently, goal $not\ q(0)$ succeeds). That is, besides falsifying the body of a rule, a dual of a non-nullary predicate can additionally be defined by disunifying its arguments and the arguments of its corresponding positive rule, as in Example 6.

Example 6. *Consider program P_5:*

$$q(0). \qquad q(s(X)) \leftarrow a(X).$$

where $a/1$ is an abducible. Let us examine the dual transformation of non-nullary predicate $q/1$.

$$\begin{array}{lllll} 1. & not\_q(X,T_0,T_2) & \leftarrow & q^{*1}(X,T_0,T_1), q^{*2}(X,T_1,T_2). \\ 2. & q^{*1}(X,I,I) & \leftarrow & X \ \backslash{=}\, 0. \\ 3. & q^{*2}(X,I,I) & \leftarrow & X \ \backslash{=}\, s(\_). \\ 4. & q^{*2}(s(X),I,O) & \leftarrow & not\_a(X,I,O). \end{array}$$

Line 1 shows the first layer of the dual rules for predicate $q/1$, which is defined as usual, i.e. $q/1$ is falsified by falsifying both its first and second rules. Lines 2-4 show the second layer of the dual rules for non-nullary predicates:

- In case of q^{*1}, the first rule of $q/1$, which is fact $q(0)$, is falsified by disunifying q^{*1}'s argument X with 0 (line 2). Note that, this is the only way to falsify $q(0)$, since it has no body.

- In case of q^{*2}, the second rule of $q/1$ is falsified by disunifying q^{*2}'s argument X with the term $s(\_)$ (line 3), or alternatively, by instead keeping the head unification and falsifying its body, i.e. by abducing the negation of $a/1$ (line 4).

Definition 2 provides the specification of the transformation to construct dualized negation in TABDUAL.

Definition 2 (Transformation of dualized negation). *1. For every predicate p/n, $n \geq 0$, defined in P:*

$$\begin{array}{rcl} p(\bar{t}_1) & \leftarrow & L_{11}, \ \ldots, \ L_{1n_1}. \\ & \vdots & \\ p(\bar{t}_m) & \leftarrow & L_{m1}, \ \ldots, \ L_{mn_m}. \end{array}$$

with $n_i \geq 0$, $1 \leq i \leq m$:

(a) The first layer of the dual transformation is defined by $\tau^{-}(p)$:

$$not\_p(\bar{X},T_0,T_m) \leftarrow p^{*1}(\bar{X},T_0,T_1), \ \ldots, \ p^{*m}(\bar{X},T_{m-1},T_m).$$

with T_i, $0 \leq i \leq m$, are fresh rule variables.[4]

[4]Variables T_i serve as abductive contexts.

(b) *The second layer of the dual transformation is defined by:* $\tau^*(p) = \bigcup_{i=1}^{m} \tau^{*i}(p)$, *and* $\tau^{*i}(p)$ *is the smallest set that contains the following rules:*

$$\begin{array}{rcl} p^{*i}(\bar{X}, I, I) & \leftarrow & \bar{X} \neq \bar{t}_i. \\ p^{*i}(\bar{t}_i, I, O) & \leftarrow & \sigma(L_{i1}, I, O). \\ & \vdots & \\ p^{*i}(\bar{t}_i, I, O) & \leftarrow & \sigma(L_{in_i}, I, O). \end{array}$$

where σ *is defined as follows:*

$$\sigma(L_{ij}, I, O) = \begin{cases} l_{ij}(\bar{t}_{ij}, I, O) & , \textit{if } L_{ij} = \textit{not } l_{ij}(\bar{t}_{ij}) \\ \textit{not\_}l_{ij}(\bar{t}_{ij}, I, O) & , \textit{if } L_{ij} = l_{ij}(\bar{t}_{ij}) \end{cases}$$

Notice that, in case of $p/0$ *(i.e.* $n = 0$*), rule* $p^{*i}(\bar{X}, I, I) \leftarrow \bar{X} \neq \bar{t}_i$ *is omitted, since both* $\bar{X}$ *and* $\bar{t}_i$ *are* [ ].[5]

2. *For every predicate* r/n, $n \geq 0$, *in* P, *that has no definition, we define* $\tau^-(r)$*:*[6]

$$\textit{not\_}r(\bar{X}, I, I).$$

Example 7. *Recall program* P *in Example 4. The transformation of Definition 2 results in:*

$$\begin{array}{rl} \tau^-(u): & \textit{not\_}u(X_1, X_2, T_0, T_2) \leftarrow u^{*1}(X_1, X_2, T_0, T_1),\ u^{*2}(X_1, X_2, T_1, T_2). \\ \tau^-(v): & \textit{not\_}v(X_1, X_2, X_3, T_0, T_1) \leftarrow v^{*1}(X_1, X_2, X_3, T_0, T_1). \\ \tau^-(w): & \textit{not\_}w(X, I, I). \\ \tau^-(\textit{false}): & \textit{not\_false}(X, I, I). \\ \tau^*(u): & u^{*1}(X_1, X_2, I, I) \leftarrow [X_1, X_2] \neq [0, \_]. \\ & u^{*2}(X_1, X_2, I, I) \leftarrow [X_1, X_2] \neq [s(X), Y]. \\ & u^{*2}(s(X), Y, I, O) \leftarrow \textit{not\_}a(X, I, O). \\ & u^{*2}(s(X), Y, I, O) \leftarrow \textit{not\_}v(X, Y, Z, I, O). \\ & u^{*2}(s(X), Y, I, O) \leftarrow w(Z, I, O). \\ \tau^*(v): & v^{*1}(X_1, X_2, X_3, I, I) \leftarrow [X_1, X_2, X_3] \neq [X, X, s(X)]. \end{array}$$

3.2.3 Transforming abducibles

In Example 5, $p^{*1}(I, O)$ is defined by abducing $not\ a$, achieved by invoking subgoal $\textit{not\_}a(I, O)$. Abduction in TABDUAL is realized by transforming each abducible atom

[5]This means, when $p/0$ is defined as a fact in P, we have $\textit{not\_p}(T_0, T_1) \leftarrow p^{*1}(T_0, T_1)$ in the first layer, but there is no rule of $p^{*1}/2$ in the second layer. Equivalently, it may be defined as $\textit{not\_p}(\_, \_) \leftarrow \textit{fail}$. (cf. the dual rule of predicate $r/0$ in Example 5).

[6]In particular, if there is no integrity constraint in P, we have $\tau^-(\textit{false}) : \textit{not\_false}(I, I)$.

(and its negation) into a rule, which updates the abductive context with the abducible atom (or its negation, respectively). Say, abducible a of Example 5 translates to:

$$a(I, O) \leftarrow insert\_abducible(a, I, O).$$

where $insert\_abducible(A, I, O)$ is a TABDUAL system predicate that inserts abducible A into input context I, resulting in output context O. It keeps the consistency of the context, failing if inserting A results in an inconsistent one. Abducible $not\ a$ is transformed similarly, where $not\ a$ is renamed into not_a in the head:

$$not\_a(I, O) \leftarrow insert\_abducible(not\ a, I, O).$$

The specification for the transformation of abducibles is given in Definition 3.

Definition 3 (Transformation of abducibles)**.** *Let $\mathcal{A}_P$ be the set of abducible atoms in P. For every $a(\bar{X}) \in \mathcal{A}_P$, we define $\tau^\circ(a(\bar{X}))$ as the smallest set that contains the rules:*

$$\begin{array}{lcl} a(\bar{X}, I, O) & \leftarrow & insert\_abducible(a(\bar{X}), I, O). \\ not\_a(\bar{X}, I, O) & \leftarrow & insert\_abducible(not\ a(\bar{X}), I, O). \end{array}$$

where $insert\_abducible/3$ is a TABDUAL *system predicate. Mark that, in the body of the second rule, 'not a' is used instead of 'not_a'.*

Example 8. *Recall program P in Example 4. We have $\mathcal{A}_P = \{a(X)\}$. The transformation of Definition 3 results in:*

$$\begin{array}{llcl} \tau^\circ(a(X)): & a(X, I, O) & \leftarrow & insert\_abducible(a(X), I, O). \\ & not\_a(X, I, O) & \leftarrow & insert\_abducible(not\ a(X), I, O). \end{array}$$

3.2.4 TABDUAL program transformation

Finally, the specification of the TABDUAL program transformation is given in Definition 4.

Definition 4 (TABDUAL program transformation)**.** *Let P be an abductive normal logic program, $\mathcal{P}_P$ be the set of predicates in P, and $\mathcal{A}_P$ be the set of abducible atoms in P. Taking:*

- $\tau'(P) = \{\tau'(r) \mid r \in P\}$
- $\tau^+(P) = \{\tau^+(p) \mid p \in \mathcal{P}_P \text{ and p is defined}\}$
- $\tau^-(P) = \{\tau^-(p) \mid p \in \mathcal{P}_P\}$
- $\tau^*(P) = \{\tau^*(p) \mid p \in \mathcal{P}_P \text{ and p is defined}\}$
- $\tau^\circ(P) = \{\tau^\circ(a) \mid a \in \mathcal{A}_P\}$

The TABDUAL *transformation τ transforms P into $\tau(P)$, where $\tau(P)$ is defined as:*

$$\tau(P) = \tau'(P) \cup \tau^+(P) \cup \tau^-(P) \cup \tau^*(P) \cup \tau^\circ(P)$$

Example 9. *The set of rules obtained in Example 4, 7, and 8 forms $\tau(P)$ of program P.*

3.2.5 Transforming queries

A query to a program, consequently, should be transformed:

- A positive goal G is simply augmented with the two extra arguments for the input and output abductive contexts.
- A negative goal $not\ G$ is made 'positive', not_G, and added the two extra input and output context arguments.

Moreover, a query should additionally ensure that all ICs are satisfied. When there is no IC defined in a program, then, following the dual transformation, fact

$$not\_false(I, I).$$

is added. Otherwise, ICs, which are rules with *false* in their heads, are transformed just like any other rules; the transformed rules with the heads $false(E)$ and $false(I, O)$ may be omitted. Finally, a query should always be conjoined with $not\_false/2$ to ensure that all integrity constraints are satisfied.

Example 10. *Query*

$$?\text{-}\ \ not\ p.$$

first transforms into $not\_p(I, O)$. *Then, to satisfy all ICs, it is conjoined with* $not\_\mathrm{false}/2$, *resulting in top goal:*

$$?\text{-}\ \ not\_p([\,], T),\ not\_false(T, O).$$

where O is an abductive solution to the query, given initially an empty input context. Note, how the abductive solution for not_p *is further constrained by passing it to the subsequent subgoal* not_false *for confirmation, via the intermediate context* T.

Definition 5 provides the specification of the query transformation.

Definition 5 (Transformation of queries)**.** *Let P be an abductive normal logic program and Q_P be a query to P as follows:*

$$?\text{-}\ \ G_1,\ \ldots,\ G_m.$$

TABDUAL transforms query Q_P into $\Delta(Q_P)$:

$$?\text{-}\ \ \delta(G_1),\ \ldots,\ \delta(G_m), not\_false(T_m, O).$$

where δ is defined as:

$$\delta(G_i) = \begin{cases} g_i(\bar{t}_i, T_{i-1}, T_i) & \text{, if } G_i = g_i(\bar{t}_i) \\ not\_g_i(\bar{t}_i, T_{i-1}, T_i) & \text{, if } G_i = not\ g_i(\bar{t}_i) \end{cases}$$

T_0 is a given initial abductive context (or $[\,]$ by default), $1 \le i \le m$, T_i, O are fresh rule variables.[7]

[7]Notice that O is the output abductive context, which returns the abductive solution(s) of the query.

Example 11. *Recall program P in Example 4. Query:*

$$?\text{-}\ \ u(0, s(0)),\ not\ u(s(0), 0).$$

is transformed by Definition 5 into:

$$?\text{-}\ \ u(0, s(0), [\,], T_1),\ not\_u(s(0), 0, T_1, T_2),\ not\_false(T_2, O).$$

4 Soundness and completeness

TABDUAL essentially exploits a LP engineering aspect of ABDUAL [4], by providing a self-sufficient program transform to table and reuse abductive solutions from one abductive context to another. As the TABDUAL transformation is underpinned by the theory of ABDUAL, its soundness and completeness stem from that of ABDUAL, notably from Theorem 3.2 of [4]. This theorem is adapted below in the context of TABDUAL, whose proof details can be found therein. The definitions of abductive framework $\langle P, \mathcal{A}, I\rangle$ and abductive solution $\langle P, \mathcal{A}, \mathcal{B}, I\rangle$ are referred to Definitions 2.6 and 2.8 of [4].

Theorem 1. *Let $\langle P, \mathcal{A}, I\rangle$ be an abductive framework and $\Delta(Q_P)$ is the transform of query Q_P w.r.t. program P, following Definition 5.*

- *Soundness: If O is a solution to $\Delta(Q_P)$, then $\langle P, \mathcal{A}, O, I\rangle$ is an abductive solution for Q_P.*
- *Completeness: If $\langle P, \mathcal{A}, O, I\rangle$ is an abductive solution for Q_P, then O is a solution to $\Delta(Q_P)$.*

Note that the ABDUAL evaluation to query Q stated in Theorem 3.2 [4] is obtained by applying ABDUAL operations (cf. Definition 3.9 of [4]). With the exception of operations ABDUCTION, CO-UNFOUNDED SET REMOVAL, and ANSWER CLAUSE RESOLUTION, other operations are covered and justified by XSB's SLG operations [46]. Indeed, these SLG operations underlie tabling mechanisms employed in TABDUAL, which is implemented in XSB. Other points worthy of note to relate the theory of ABDUAL and the LP engineering aspects of TABDUAL:

- The input and output abductive contexts are operational representation of abductive context Set in an *abductive subgoal* $\langle L, Set\rangle$, cf. Definition 2.6 of [4].
- The system predicate $insert\_abducible/3$ in Definition 3 implements the ABDUAL operation ABDUCTION. Predicate $produce\_context/3$ in Definition 1 does a similar job, i.e., taking care of the union of abductive contexts that appears in ABDUAL operations ANSWER CLAUSE RESOLUTION and SIMPLIFICATION.

- In [4], two forms of dual program are introduced, i.e. folded and unfolded dual program. Their need is more theoretical, to show the soundness and completeness of the dual transformation and ABDUAL. The folded form specifically deals with infinite ground programs: it avoids infinite dual rule bodies, by swapping that infinite body possibility with a folded recurrent call, to the first body literal, followed by a folded call to the remaining body literals, and so on, possibly incurring in an infinite number of rules instead. On the other hand, the unfolded form differs from the folded one only insofar as no folding rules are defined, but is equivalent to the folded form.

  Though the theory of ABDUAL underpins TABDUAL, we need not be concerned with the folded dual form in TABDUAL, as it deals only with real finite non-ground programs, whose rules stand for all their ground instances. Indeed, the dual transformation in TABDUAL of Definition 2 is another way of expressing the unfolded form. Observe that p^{*i} of Definition 2 corresponds to default conjugate $conj_D(L_{i,j_i})$ in the unfolded form, where the computation of $conj_D(L_{i,j_i})$ for each j_i is realized by the second layer of the transformation in Definition 2.

- The TABDUAL implementation caters to programs with variables and non-ground queries (cf. Example 6, and later discussed in Section 6.2). Its non-groundness does not violate the groundness assumption in the theory of ABDUAL, since one can move the head unifications of a rule to equalities in its the body, before applying the TABDUAL transformation. Recall Example 6, the two rules of $q/1$ can be rewritten as:

  $$q(X) \leftarrow X = 0. \qquad\qquad q(X) \leftarrow X = s(X'),\ a(X').$$

  Using the above rewritten rules, it now becomes obvious how q^{*1} and q^{*2} in Example 6 are derived by the dual transformation. Mark that, because the dual transformation needs only fail one subgoal in the body at a time, the second definition of q^{*2}, i.e. $q^{*2}(s(X), I, O) \leftarrow not\_a(X, I, O)$ is obtained by assuming the equality $X = s(X')$ in the body, but alternatively failing $a(X')$:[8]

  $$q^{*2}(X, I, O) \leftarrow X = s(X'), not\_a(X', I, O).$$

  which is equivalent to rule 4 in Example 6, treating back the equality $X = s(X')$ in the body as head unification modulo variable renaming. To sum up, applying the above rewriting (before the TABDUAL transformation) to rules with variables allows to avoid defining a specific dual transformation for particularly dealing with such rules. Inasmuch as head unifications of a rule are moved to equalities in its body, one

[8] As shown in Section 6.2.1, the implementation includes all positive literals that precede the negated literal in a dual rule. Indeed, this is just another instance of requiring one failed literal in the body and allowing to assume other (preceding) positive literals to succeed.

can think of the ground instances of all the rules, and stick to the dual transformation of ABDUAL with its groundness assumption.

- When dealing with loops, TABDUAL relies as much as possible on XSB's tabling mechanism. This is specifically true for direct positive loops (Section 6.4.1). For positive loops in dualized negation (Section 6.4.2) and negative loops over negation (Section 6.4.3) additional treatments in the implementation are required, to deal with these loops correctly, now in the presence of the TABDUAL transformation. For instance, the CWA list introduced in dealing with positive loops in dualized negation (Section 6.4.2) implements the co-unfounded set of literals and supports its corresponding operation CO-UNFOUNDED SET REMOVAL in ABDUAL. Indeed, the same technique is also implemented in the ABDUAL meta-interpreter [5]. The other treatment, viz., for negative loops over negation (Section 6.4.3), benefits from XSB's tabled negation to explicitly enact DELAYING operation in ABDUAL, now within the TABDUAL transformation. All these techniques and their rationale are detailed in Section 6.4.

In summary, constructs introduced in the TABDUAL transformation as well as the implementation technical details, such as how to deal with non-groundness and programs having loops, are just a concrete realization of the more abstract theory of ABDUAL and its operations. Other implementation aspects, e.g., the dual transformation by-need (Section 6.5), are extra optimizations pertinent to XSB features, like tabling and tries.

5 Complexity

In terms of complexity, the size of the program produced by the TABDUAL transformation is linear in the size of the input program, as shown in Theorem 2, which is similar to that of ABDUAL using the folded dual form (cf. Lemma A.5 in [4]).

Definition 6. *Let P be a finite logic program and $\mathcal{B}_r$ be the body of rule $r \in P$.*

- *$preds(P)$ denotes the number of predicates in P.*
- *$heads(P)$ denotes the number of predicates defined (i.e. with rules) in P.*
- *$rules(P)$ denotes the number of rules in P.*
- *$size(P|_p)$ denotes the size of rules in P whose head is the predicate p.*
- *$size(P)$ denotes the size of P and is defined as*

$$size(P) = \Sigma_{i=1}^{rules(P)} \left(1 + |\mathcal{B}_{r_i}|\right)$$

where $|\mathcal{B}_{r_i}|$ denotes the number of body literals in r_i.[9]

The following theorem shows that the size of the program produced by the TABDUAL transformation is linear in the size of the original program.

Theorem 2. *Let P be an abductive normal logic program and $\mathcal{A}_P$ be the set of abducible atoms in P. Then $size(\tau(P)) < 9.size(P) + 4.|\mathcal{A}_\mathcal{P}|$.*

The proof is contained in Appendix A.

The problem of query evaluation to abductive frameworks is NP-complete, even for those frameworks in which entailment is based on the WFS [14]. In [4], it is shown that the complexity of an ABDUAL query evaluation is proportional to the maximal number of abducibles in any abductive subgoals, and to the number of abducible atoms in the program. In particular, if the set of abducible atoms and ICs are both empty, then the cost of query evaluation is polynomial. The complexity of TABDUAL query evaluation should naturally be based on that of ABDUAL. One may observe that the table size, used in tabling abductive solutions, would be proportional to the number of distinct (positive) subgoals in the procedural call-graph, i.e. each first call of the subgoals in a given query will table, as solution entries, the abductive solutions of the called subgoal. Besides tabling, a number of implementation aspects discussed in Section 6 may help improve performance in practice.

6 Implementation aspects of TABDUAL

Next, we discuss several aspects pertaining to the implementation of the TABDUAL transformation. The implementation aspects of TABDUAL introduced in this section aims at realizing the abstract theory of ABDUAL as well as benefitting from XSB's features for TABDUAL optimizations to foster its more practical use.

6.1 Abductive and non-abductive program parts

We start by specifying TABDUAL's input programs and its basic constructs.

Example 12. *An example of input programs of* TABDUAL*:*

$$\begin{array}{lll}
abds([a/1]). & & \\
s(X) & \leftarrow & prolog(atom(X)), a(X). \\
s(X) & \leftarrow & prolog(nat(X)), a(X). \\
 & & \\
beginProlog. & & \\
\quad nat(0). & & nat(s(X)) \leftarrow nat(X). \\
endProlog. & &
\end{array}$$

[9]That is, the size of a rule r is defined as the total number of (head and body) literals in r.

The input program of TABDUAL, as shown in Example 12, may consist of two parts: abductive and non-abductive parts. Abducibles need to be declared, in the abductive part, using predicate $abds/1$, whose sole argument is the list of abducibles and their arities. The non-abductive part is distinguished from the abductive part by the $beginProlog$ and $endProlog$ identifiers. Any program between these identifiers will not be transformed, i.e. it is treated as a usual Prolog program. Access to the program in the non-abductive part is established using the TABDUAL system predicate $prolog/1$. That is, $prolog/1$ is a meta-predicate that calls a user-defined predicate specified in the non-abductive part, e.g., in $prolog(nat(X))$ of Example 12. This meta-predicate's argument may also be a Prolog built-in predicate, e.g., in $prolog(atom(X))$. In essence, it executes the goal in its only argument not subject to TABDUAL's transformation.

6.2 Dealing with non-ground programs

This section touches upon abduction in programs with variables. The implementation deals with two issues, viz., grounding the dualized negative subgoals in the dual transformation and dealing with non-ground negative goals.

6.2.1 Grounding dualized negated subgoals

Example 13. *Consider program P_6, with $a/1$ abducible:*

$$q(1). \qquad r(X) \leftarrow a(X). \qquad \leftarrow q(X), r(X).$$

The TABDUAL *transformation results in (notice that the last rule in P_6 is an IC):*

$$\begin{array}{rlcl}
1. & q_{ab}(1,[\,]). & & \\
2. & q(X,I,O) & \leftarrow & q_{ab}(X,E), produce\_context(O,I,E). \\
3. & not\_q(X,I,O) & \leftarrow & q^{*1}(X,I,O). \\
4. & q^{*1}(X,I,I) & \leftarrow & X \ \backslash= 1. \\
 & & & \\
5. & r_{ab}(X,[a(X)]). & & \\
6. & r(X,I,O) & \leftarrow & r_{ab}(X,E), produce\_context(O,I,E). \\
7. & not\_r(X,I,O) & \leftarrow & r^{*1}(X,I,O). \\
8. & r^{*1}(X,I,I) & \leftarrow & X \ \backslash= \_. \\
9. & r^{*1}(X,I,O) & \leftarrow & not\_a(X,I,O). \\
 & & & \\
10. & \text{not\_false}(I,O) & \leftarrow & \text{false}^{*1}(I,O). \\
11. & \text{false}^{*1}(I,O) & \leftarrow & not\_q(X,I,O). \\
12. & \text{false}^{*1}(I,O) & \leftarrow & not\_r(X,I,O).
\end{array}$$

Consider query $q(1)$, which is transformed into:

$$?\text{-}\ \ q(1, [\,], T), not\_false(T, O).$$

Invoking the first subgoal, $q(1, [\ ], T)$, results in $T = [\ ]$. Invoking subsequently the second subgoal, $not\_false([\ ], O)$, results in the abductive solution of the given query: $O = [not\ a(X)]$, obtained via rules 10, 12, 7, and 9. Note that rule 11, an alternative to *false*^{*1}, fails due to uninstantiated X in its subgoal $not\_q(X, I, O)$, which leads to failing rules 3 and 4. For the same reason, rule 8, an alternative to r^{*1}, also fails.

Instead of having $[not\ a(1)]$ as the abductive solution to the query $q(1)$, we have the incorrect non-ground abductive solution $[not\ a(X)]$. It does not meet our requirement, in Section 2, that abducibles must be ground on the occasion of their abduction. The problem can be remedied by instantiating X, in rule 12, thereby eventually grounding the abducible $not\ a(X)$ when it is abduced, i.e. the argument X of subgoal $not\_a/3$, in rule 9, becomes instantiated.

In the implementation, grounding a dualized negated subgoal is achieved as follows: in addition to placing a negated literal, say not_p, in the body of the second layer dual rule, all positive literals that precede literal p, in the body of the corresponding original positive rule, are also kept in the body of the dual rule. For rule 12, introducing the positive subgoal $q(X)$, originating from the positive rule, before the negated subgoal $not\_r(X, I, O)$ in the body of rule 12, helps instantiate X in this case. Rule 12 now becomes (all other rules remain the same):

$$12.\quad \mathit{false}^{*1}(I, O) \quad \leftarrow \quad q(X, I, T), not\_r(X, T, O).$$

Notice that, differently from before, the rule is now defined by introducing all positive literals that appear before r in the original rule; in this case we introduce $q/3$ before $not\_r/3$. As the result, the argument X in $not\_r/3$ is instantiated to 1, due to the invocation of $q/3$, just like the case in the original rule. It eventually helps ground the the negated abducible $not\ a(X)$, when it is abduced, and the correct abductive solution $[not\ a(1)]$ to query $q(1)$ is returned. By implementing this technique, we are also able to deal with non-ground positive goals, e.g., query $q(X)$ gives the correct abductive solution as well, i.e. $[(not\ a(1)]$ for $X = 1$.

There are some points to remark on regarding this implementation technique. First, the semantics of dual rules does not change because the conditions for failure of their positive counterpart rules are that one literal must fail, even if the others succeed. The cases where the others do not succeed are handled in the other alternatives of dual rules. Second, this technique may benefit from the TABDUAL's tabled predicate, e.g. q_{ab} for predicate q, as it helps avoid redundant derivations of the newly introduced positive literals in dual rules. Finally, knowledge of shared variables in the body and whether they are local or not, may be useful to avoid introducing positive literals that are not contributing to further grounding.

6.2.2 Non-ground negative goals

Example 14. *Consider program P_7, with $a/1$ abducible:*

$$p(1) \leftarrow a(1). \qquad p(2) \leftarrow a(2).$$

Query $p(X)$ to program P_7 succeeds under TABDUAL, giving two abductive solutions: $[a(1)]$ and $[a(2)]$ for $X = 1$ and $X = 2$, respectively. But query $not\ p(X)$ does not deliver the expected solution. Instead of returning the abductive solution $[not\ a(1), not\ a(2)]$ for any instantiation of X, it returns $[not\ a(1)]$ for a particular $X = 1$. In order to find the culprit, we first look into the definition of $not\_p/3$:

$$\begin{array}{lll}
1. & not\_p(X, I, O) & \leftarrow & p^{*1}(X, I, T), p^{*2}(X, T, O). \\
2. & p^{*1}(X, I, I) & \leftarrow & X \backslash= 1. \\
3. & p^{*1}(1, I, O) & \leftarrow & not\_a(1, I, O). \\
4. & p^{*2}(X, I, I) & \leftarrow & X \backslash= 2. \\
5. & p^{*2}(2, I, O) & \leftarrow & not\_a(2, I, O).
\end{array}$$

Recall that query $not\ p(X)$ is transformed into:

$$?\text{-}\ \ not\_p(X, [\,], N), not\_false(N, O).$$

When the goal $not\_p(X, [\,], N)$ is launched, it first invokes $p^{*1}(X, [\,], T)$. It succeeds by the second rule of p^{*1}, in line 3 (the first rule, in line 2, fails it), with variable X is instantiated to 1 and T to $[not\ a(1)]$. The second subgoal of $not\_p(X, [\,], N)$ is subsequently invoked with the same instantiation of X and T, i.e. $p^{*2}(1, [not\ a(1)], O)$, and it succeeds by the first rule of p^{*2}, in line 4, and results in $N = [not\ a(1)]$. Since there is no IC in P_6, the abductive solution $[not\ a(1)]$ is just relayed from N to O, due to the fact $not\_false(I, I)$ in the transformed program (cf. Section 3.2.5), thus returning the abductive solution $[not\ a(1)]$ with $X = 1$ for the given query.

The culprit of this wrong solution is that both subgoals of $not\_p/3$, i.e. $p^{*1}/3$ and $p^{*2}/3$, share the argument X of $p/1$. This should not be the case, as $p^{*1}/3$ and $p^{*2}/3$ are derived from two different rules of $p/1$, hence failing p should be achieved by invoking p^{*1} and p^{*2} with an independent argument X. In other words, different variants of the calling argument X should be used in $p^{*1}/3$ and $p^{*2}/3$, as shown for rule $not\_p/3$ (line 1) below:

$$\begin{array}{llll}
1. & not\_p(X, T_0, T_2) & \leftarrow & copy\_term([X], [X_1]), p^{*1}(X_1, T_0, T_1), \\
 & & & copy\_term([X], [X_2]), p^{*2}(X_2, T_1, T_2).
\end{array}$$

where the Prolog built-in predicate $copy\_term/2$ provides a variant of the list of arguments; in this example, we simply have only one argument, i.e. $[X]$.

Now, $p^{*1}/3$ and $p^{*2}/3$ are invoked using variant independent calling arguments, viz., X_1 and X_2, respectively. The same query first invokes $p^{*1}(X_1, [\,], T_1)$, which results in $X_1 = 1$ and $T_1 = [\mathit{not}\ a(1)]$ (by the second rule of p^{*1}), and subsequently invokes $p^{*2}(X_2, [\mathit{not}\ a(1)], T_2)$, resulting in $X_2 = 2$ and $T_2 = [\mathit{not}\ a(1), \mathit{not}\ a(2)]$ (by the second rule of p^{*2}). It eventually ends up with the expected abductive solution: $[\mathit{not}\ a(1), \mathit{not}\ a(2)]$ for any instantiation of X, i.e. X remains unbound.

The technique ensures, as this example shows, that $p(X)$ fails for every X, and its negation, $\mathit{not}\ p(X)$, hence succeeds. The dual rules produced for the negation are tailored to be, by definition, an 'if and only if' with regard to their corresponding positive rules. If we added the fact $p(Y)$ to P_7, then the same query $\mathit{not}\ p(X)$ would not succeed because now we have the first layer dual rule:

$$\begin{array}{lcl} \mathit{not\_p}(X, T_0, T_3) & \leftarrow & \mathit{copy\_term}([X],[X_1]), p^{*1}(X_1, T_0, T_1), \\ & & \mathit{copy\_term}([X],[X_2]), p^{*2}(X_2, T_1, T_2), \\ & & \mathit{copy\_term}([X],[X_3]), p^{*3}(X_3, T_2, T_3). \end{array}$$

and an additional second layer dual rule $p^{*3}(X, \_, \_) \leftarrow X \neq \_$ that always fails; its abductive contexts are thus irrelevant.

6.3 Transforming predicates with facts only

TABDUAL transforms predicates that comprise of just facts as any other rules in the program (cf. fact $q(1)$ and its transformed rules, in Example 13). This is clearly superfluous as facts do not induce any abduction, and the transformation would be unnecessarily heavy for programs with large factual data, which is often the case in many real world problems.

A predicate, say $q/1$, comprised of just facts, can be much more simply transformed. The transformed rules $q_{ab}/2$ and $q/3$ can be substituted by a single rule:

$$q(X, I, I) \leftarrow q(X).$$

and their negations, rather than using dual rules, can be transformed to a single rule:

$$\mathit{not\_q}(X, I, I) \leftarrow \mathit{not}\ q(X).$$

independently of the number of facts $q/1$ are there in the program. Note that the input and output context arguments are added in the head, and the input context is just passed intact to the output context. Both rules simply execute the fact calls.

Facts of predicate $q/1$ can thus be defined in the non-abductive part of the input program. For instance, if a program contains facts $q(1)$, $q(2)$, and $q(3)$, they are listed as:

$$beginProlog. \qquad q(1). \qquad q(2). \qquad q(3). \qquad endProlog.$$

Though this new transformation for facts seems trivial, it considerably improves the performance, in particular if we deal with abductive logic programs having large factual data. In this case, not only the time and space in the transformation stage can be reduced, but also in the abduction.

6.4 Dealing with loops

The tabling mechanism in XSB supports the Well-Founded Semantics, therefore it allows dealing with loops in the program, ensuring termination of looping queries. In TABDUAL, tabling for loops is taken care by its transformation, as shown in this section. TABDUAL relies as much as possible on XSB's tabling mechanism in dealing with loops; this is specifically the case for direct positive loops (Section 6.4.1). Nevertheless, the presence of tabled abduction requires some varieties of loops, viz., positive loops in (dualized) negation and negative loops over negation, to be handled carefully in the transformation, as we detail in Sections 6.4.2 and 6.4.3, respectively. Additional examples, besides the ones below, are to be found in Appendix B.

6.4.1 Direct positive loops

Example 15. *Consider program P_8 which involves a direct positive loop between predicates:*

$$p \leftarrow q. \qquad q \leftarrow p.$$

The tabling mechanism in XSB would detect direct positive loops and fail predicates involved in such loops. The TABDUAL transformation may simply benefit from it. For P_8, query p fails, due to the direct positive loop between tabled predicates p_{ab} and q_{ab}:

$$p_{ab}(E) \leftarrow q([\,],E). \qquad p(I,O) \leftarrow p_{ab}(E), produce\_context(O,I,E).$$
$$q_{ab}(E) \leftarrow p([\,],E). \qquad q(I,O) \leftarrow q_{ab}(E), produce\_context(O,I,E).$$

On the other hand, query $not\ p$ should succeed with the abductive solution: $[\,]$. But, instead of succeeding, this query will loop indefinitely! Recall that the call to query $not\ p$, after the transformation, becomes $not\_p([\,],T), not\_false(T,O)$. The indefinite loop occurs in $not\_p([\,],T)$ because of the mutual dependency between not_p and not_q through p^{*1} and q^{*1}:

$$not\_p(I,O) \leftarrow p^{*1}(I,O). \qquad p^{*1}(I,O) \leftarrow not\_q(I,O).$$
$$not\_q(I,O) \leftarrow q^{*1}(I,O). \qquad q^{*1}(I,O) \leftarrow not\_p(I,O).$$

The dependency creates a positive loop on negative non-tabled predicates, and such loops should succeed, precisely because the corresponding source program's loop is a direct one on positive literals, which hence must fail. We now turn to how to deal with such loops in TABDUAL.

6.4.2 Positive loops in (dualized) negation

Since any source program's direct positive loops must fail, the loops between their corresponding transformed negations, i.e. positive loops in dualized negation (introduced via the dual transformation), must succeed [4]. For instance, whereas $r \leftarrow r$ fails query r, perforce

$not\_r \leftarrow not\_r$ succeeds query not_r. This is formalized in the notion of co-unfounded set of literals in ABDUAL (cf. Definition 3.5 of [4])

We detect positive loops in (dualized) negation, PLoN for short, by tracking the ancestors of negative subgoals, whenever they are called from other negative subgoals. In the transformation, a list of ancestors, dubbed the *close-world-assumption (CWA) list* is maintained. It contains only negative literals and serves as another extra argument in the first and second layers of dual rules. Indeed, the role of this ancestor list to deal with PLoN implements the co-unfounded set of literals in ABDUAL.

The TABDUAL transformation, with PLoN detection, of P_8 results in the following first and second layers of dual rules (other transformed rules remain the same):

$$\begin{array}{lllll}
1. & not\_p(I,I,C) & \leftarrow & member(not\,p,C),!. \\
2. & not\_p(I,O,C) & \leftarrow & p^{*1}(I,O,C). \\
3. & p^{*1}(I,O,C) & \leftarrow & not\_q(I,O,[not\,p \mid C]). \\
\\
4. & not\_q(I,I,C) & \leftarrow & member(not\,q,C),!. \\
5. & not\_q(I,O,C) & \leftarrow & q^{*1}(I,O,C). \\
6. & q^{*1}(I,O,C) & \leftarrow & not\_p(I,O,[not\,q \mid C]).
\end{array}$$

The CWA list C is only updated in the second layer of dual rules (cf. rules p^{*1} and q^{*1} in line 3 and 6, respectively), i.e. by adding the negative literal corresponding to the dual rule into list C. For example, in case of p^{*1} (line 3), $not\ p$ is added into the CWA list C. Note that, since the CWA list is intended to detect PLoN, the list is reset in positive subgoals occurring in the body of a dual rule. This guarantees that there are no interposing positive calls between the negative calls and their ancestor, which would break such loops.

The updated CWA list C is then used to detect PLoN via an additional rule of not_p (line 1, and similarly in line 4, for not_q). The idea is to test, whether we are returning to the same call of not_p, which is simply realized by a membership testing. If that is the case, the output context is set equal to the input context, and PLoN is anticipated by immediately succeeding not_p with the extra cut to prevent the call to the next not_p rule (which would otherwise lead to looping). This detection technique thus establishes the CO-UNFOUNDED SET REMOVAL operation in the ABDUAL theory.

This technique of PLoN detection consequently requires query $not\ p$ to be transformed into:

$$?\text{-}\ \ not\_p([\,],T,[\,]), not\_false(T,O).$$

i.e. it is initially called with an empty CWA list.

6.4.3 Negative loops over negation

The other type of loops that XSB's tabling mechanism already properly deals with, is the negative loops over negation (NLoN).

Example 16. *Consider program P_9:*

$$p \leftarrow q. \qquad\qquad q \leftarrow not\ p.$$

In XSB, the tabling mechanism makes p and q (also their default negations) undefined. But under TABDUAL, query p (also q) will fail, instead of being undefined. It fails, because the tabled predicate p_ab is involved in a direct positive loop as shown in the transformation below:

$$\begin{array}{lcl}
p(I,O) & \leftarrow & p_{ab}(E), produce\_context(O,I,E).\\
p_{ab}(E) & \leftarrow & q([\,],E).\\
q(I,O) & \leftarrow & q_{ab}(E), produce\_context(O,I,E).\\
q_{ab}(E) & \leftarrow & not\_p([\,],E).\\
\\
not\_p(I,O) & \leftarrow & p^{*1}(I,O).\\
p^{*1}(I,O) & \leftarrow & not\_q(I,O).\\
not\_q(I,O) & \leftarrow & q^{*1}(I,O).\\
q^{*1}(I,O) & \leftarrow & p(I,O).
\end{array}$$

More precisely, whereas in the original program P_9, q is defined by the negative subgoal $not\ p$, in the resulting transformation q is defined by the positive subgoal not_p via the tabled predicate q_{ab}.

One way to resolve the problem is to wrap the positive subgoal not_p in the body of the rule q_{ab} with the tabled negation predicate ($tnot/1$ in XSB) twice: on the one hand it preserves the semantics of the rule (keeping the truth value by applying $tnot$ twice), and on the other hand introducing $tnot$ creates NLoN (instead of direct positive loops). Predicate q_{ab} is thus defined as follows (other transformed rules remain the same):

$$\begin{array}{rlcl}
1. & q_{ab}(E) & \leftarrow & not\_p_{tu}([\,],E).\\
\\
2. & not\_p_{tu}(I,I) & \leftarrow & call\_tv(tnot\ over(not\_p(I)), \textit{undefined}).\\
3. & not\_p_{tu}(I,O) & \leftarrow & call\_tv(tnot\ over(not\_p(I)), \textit{true}),\ p^{*1}(I,O).\\
\\
4. & not\_p(I) & \leftarrow & p^{*1}(I,\_).
\end{array}$$

Here, $tnot\ over(not\_p(I))$ is the double-wrapping of not_p with $tnot$. It is realized via the intermediate tabled predicate $over/1$, defined as:

$$over(G) \leftarrow tnot(G).$$

The double-wrapping is called through a new auxiliary predicate $not\_p_{tu}/2$. The XSB system predicate $call\_tv/2$ calls the double-wrapping and is used to distinguish the two cases (lines 2 and 3): whether NLoN exists or not. In the former case, the returned truth value is undefined; therefore not_p_{tu} itself is undefined and its input context is simply relayed to the output context. In the latter case, where the returned truth value is true, the output context O of not_p_{tu} is obtained from the input context I as usual, i.e. by invoking $p^{*1}(I, O)$.

Notice that, instead of using the existing $not\_p(I, O)$ in the double-wrapping, we use an auxiliary predicate $not\_p(I)$ to avoid floundering in the call to $over/1$, due to the uninstantiated output context O. For this reason, the newly introduced $not\_p/1$ is thus free from the output context, but otherwise defined exactly as $not\_p/2$.

6.5 Dual transformation by-need

The TABDUAL transformation conceptually constructs *all* (first and second layer) dual rules, in advance, for every defined atom in an input program, regardless whether they are needed in abduction. We refer to this conceptual construction of dual rules in the sequel as the dual transformation STANDARD.

The dual transformation STANDARD should be avoided in practice, as potentially large sets of dual rules are created in the transformation, though only a few of them might be invoked during abduction. As real world problems typically consist of a huge number of rules, such dual transformation may suffer from a heavy computational load, and therefore hinders the subsequent abduction phase to take place, not to mention the compile time, and space requirements, of the large thus produced transformed program.

One solution to this problem is to compute dual rules *by-need*. That is, dual rules are concretely created in the abduction stage (rather than in the transformation stage), based on the need of the on-going invoked goals. The transformed program still contains the single first layer rule of the dual transformation, but its second layer is defined using a newly introduced TABDUAL system predicate, which will be interpreted by the TABDUAL system on-the-fly, during abduction, to produce the concrete rule definitions of the second layer.

Example 17. *Recall Example 5. The dual transformation by-need contains the same first layer:* $not\_p(T_0, T_2) \leftarrow p^{*1}(T_0, T_1), p^{*2}(T_1, T_2)$. *But the second now contains, for each* $i \in \{1, 2\}$*:*

$$p^{*i}(I, O) \leftarrow dual(i, p, I, O).$$

Predicate $dual/4$ is a TABDUAL system predicate, introduced to facilitate the dual transformation by-need:

1. It constructs generic dual rules, i.e. dual rules without any context attached to them, by-need, from the i-th rule of $p/1$, during abduction,
2. It instantiates the generic dual rules with the provided arguments and input context,
3. Finally, it subsequently invokes the instantiated dual rules.

The dual transformation by-need clearly reduces the size of the second layer dual rules. Recall from Definition 6, that the size of a rule r is the total number of (head and body) literals in p. Therefore, each rule r^{*i} has a constant size 4, which is obtained from its two rules: (1) a rule that disunifies its arguments, viz. $r^{*i}(\bar{X}, I, I) \leftarrow \bar{X} \neq \bar{t}_i$ and, (2) a rule defined with the above TABDUAL system predicate $dual/4$, viz. $r^{*i}(\bar{t}_i, I, O) \leftarrow dual(i, r(\bar{t}_i), I, O)$. Compared to the dual transformation STANDARD, whose size of $\tau^*(P)$ for constructing its second layer dual rules $size(\tau^*(P)) = 2.size(P)$ (cf. proof of Theorem 1, in Appendix A), the size of $\tau^*_{need}(P)$ for the dual transformation by-need, assuming there are m rules of r, is $size(\tau^*_{need}(P)) = \Sigma_{i=1}^{heads(P)} 4m = 4.rules(P)$. Given the definition of $size(P)$ in Definition 6, it is straightforward to verify, that the advantage of the dual transformation by-need over the STANDARD one becomes apparent when the body of r contains as little as two literals, rendering $size(\tau^*_{need}(P)) < size(\tau^*(P))$. More concrete evaluation of the dual-transformation by-need is exemplified and discussed in Section 7.3.

Having said that, constructing dual rules on-the-fly clearly introduces some extra cost in the abduction stage. Such extra cost can be reduced by memoizing the already constructed generic dual rules. Therefore, when such dual rules are later needed, they are available for reuse and their recomputation avoided.

We examine two approaches for memoizing generic dual rules for the dual transformation by-need. Their definitions of system predicate $dual/4$ are different, which distinguish how generic dual rules are constructed by-need. The first approach (Section 6.5.1) benefits from tabling to memoize generic dual rules, whereas the second one (Section 6.5.2) employs XSB's trie data structure [50]. They are referred in the sequel as BY-NEED(EAGER) and BY-NEED(LAZY), respectively, due to their dual rules construction mechanisms.

6.5.1 Dualization BY-NEED(EAGER): tabling generic dual rules

The straightforward choice for memoizing generic dual rules is to use tabling. The system predicate $dual/4$ is defined as follows (abstracting away irrelevant details):

$$dual(N, P, I, O) \leftarrow dual\_rule(N, P, Dual), call\_dual(P, I, O, Dual).$$

where $dual\_rule/3$ is a *tabled* predicate that constructs a generic dual rule $Dual$ from the N-th rule of atom P, and $call\_dual/4$ instantiates $Dual$ with the provided arguments of P

and the input context I. It also invokes the instantiated dual rule to produce the abductive solution in O.

Though predicate $dual/4$ helps realize the construction of dual rules by-need, i.e. only when a particular p^{*i} is invoked, this approach results in the *eager* construction of all dual rules for the i-th rule of predicate p, because of tabling (assuming XSB's local table scheduling is in place, rather than its alternative, in general less efficient, batched scheduling). For instance, in Example 5, when $p^{*2}(I, O)$ is invoked, which subsequently invokes $dual\_rule(2, p, Dual)$, all two alternatives of dual rules from the second rule of p, i.e. $p^{*2}(I, O) \leftarrow not\_q(I, O)$ and $p^{*2}(I, O) \leftarrow r(I, O)$ are constructed before $call\_dual/4$ is invoked for each of them. This is a bit against the spirit of a full by-need dual transformation, where only one alternative dual rule is constructed at a time, just before it is invoked. That is, generic dual rules could be constructed *lazily*.

As mentioned earlier, the reason behind this eager by-need construction is the local table scheduling strategy, that is employed by default in XSB. This scheduling strategy may not return any answers out of a strongly connected component (SCC) in the subgoal dependency graph, until that SCC is completely evaluated [48].

Alternatively, batched scheduling is also implemented in XSB, which allows returning answers outside of a maximal SCC as they are derived. In terms of the dual rules construction by-need, this means $dual\_rule/3$ would allow dual rules to be lazily constructed. That is, only one generic dual rule is produced at a time before it is instantiated and invoked. Since the choice between the two scheduling strategies can only be made for the whole XSB installation, and is not (as yet) predicate switchable, we pursue another approach to implement lazy dual rule construction.

6.5.2 Dualization BY-NEED(LAZY): storing generic dual rules in a trie

Trie is a tree data structure that allows data, such as strings, to be compactly stored by a shared representation of their prefixes. That is, all the descendants of a node in a trie have a common prefix of the string associated with that node.

XSB offers a mechanism for facts to be directly stored and manipulated in tries. Figure 1, taken from [50], depicts a trie that stores a set of Prolog facts:

$$\{rt(a, f(a, b), a), rt(a, f(a, X), Y), rt(b, V, d)\}.$$

For trie-dynamic code, trie storage has advantages, both in terms of space and time [50]:

- A trie can use much less space to store many sets of facts than standard dynamic code, as there is no distinction between the index and the code itself.

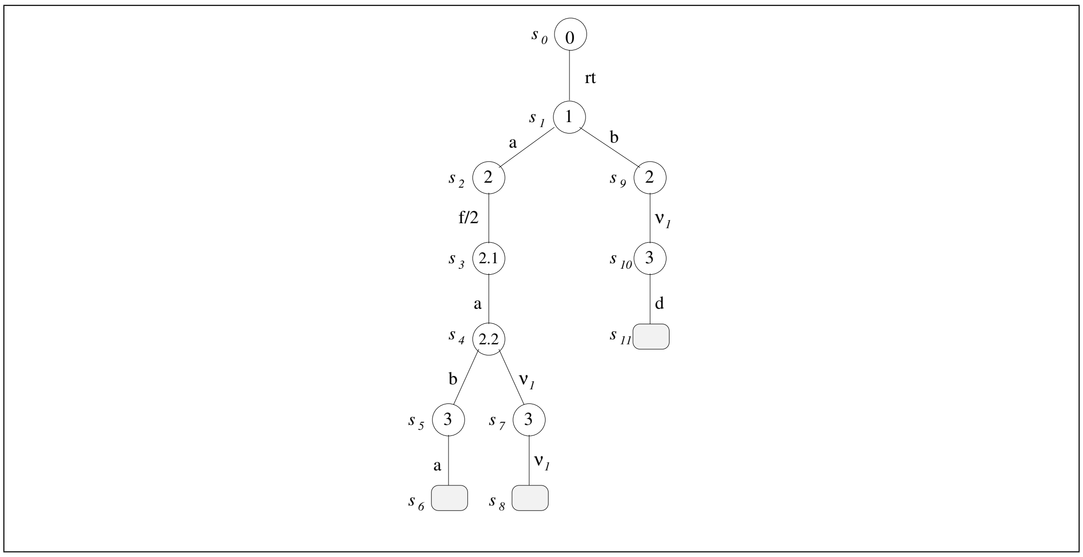

Figure 1: Facts stored as a trie.

- Directly inserting into or deleting from a trie is faster (up to 4-5 times) than with standard dynamic code, as discrimination can be made on a position anywhere in a fact.

XSB provides predicates for inserting terms into a trie, unifying a term with terms in a trie, and other trie manipulation predicates, both in the low-level and high-level API.

Generic dual rules can be represented as facts, thus once they are constructed, they can be memoized and later (a copy) retrieved and reused. Given the aforementioned advantages for storing dynamic facts and XSB support for its manipulation, a trie is preferable to the common Prolog database to store dynamically generated (i.e., by-need) dual rules. The availability of XSB system predicates to manipulate terms in a trie permits explicit control in lazily constructing generic dual rules compared to the more eager tabling approach (Section 6.5.1), as detailed below.

A fact of the form $d(N, P, Dual, Pos)$ is used to represent a generic dual rule $Dual$ from the N-th rule of P with the additional tracking information Pos, which informs the position of the literal used in constructing each dual rule. In the current TABDUAL implementation, we opt for the low-level API trie manipulation predicates, as they can be faster than the higher-level API.

Using this approach, the system predicate $dual/4$ is defined as follows (abstracting away irrelevant details):

1. $dual(N, P, I, O) \quad \leftarrow \quad trie\_property(T, alias(dual)), dual(T, N, P, I, O).$

2. $dual(T, N, P, I, O) \quad \leftarrow \quad trie\_interned(d(N, P, Dual, \_), T),$
   $call\_dual(P, I, O, Dual).$
3. $dual(T, N, P, I, O) \quad \leftarrow \quad current\_pos(T, N, P, Pos),$
   $dualize(Pos, Dual, NextPos),$
   $store\_dual(T, N, P, Dual, NextPos),$
   $call\_dual(P, I, O, Dual).$

Assuming that a trie T with alias $dual$ has been created, predicate $dual/4$ (line 1) is defined by an auxiliary predicate $dual/5$ with an access to the trie T, the access being provided by the trie manipulation predicate $trie\_property/2$. Lines 2 and 3 give the definition of $dual/5$. In the first definition (line 2), an attempt is made to reuse generic dual rules, which are stored already as facts $d/4$ in trie T. This is accomplished by unifying terms in T with $d(N, P, Dual, \_)$, one at a time through backtracking, via the trie manipulation predicate $trie\_interned/2$. Predicate $call\_dual/4$ then does the job as before. The second definition (line 3) constructs generic dual rules lazily. It finds, via $current\_pos/4$, the current position Pos of the literal from the N-th rule of P, which can be obtained from the last argument of fact $d(N, P, Dual, Pos)$ stored in trie T. Using this Pos information, a new generic dual rule $Dual$ is constructed by means of $dualize/3$. The latter predicate additionally updates the position of the literal, $NextPos$, for the next dualization. The dual rule $Dual$, together with the tracking information, is then memoized as a fact $d(N, P, Dual, NextPos)$ in trie T, via $store\_dual/5$. Finally, the just constructed dual $Dual$ is instantiated and invoked using $call\_dual/4$.

Whereas the first approach constructs generic dual rules by-need eagerly, the second one does it lazily. But this requires memoizing dual rules to be carried out explicitly, and additional tracking information is needed to correctly pick up on dual rule generation at the point where it was last left. This approach affords us a simulation of batched table scheduling for $dual/5$, within the default local table scheduling.

6.6 Accessing ongoing abductive solutions

TABDUAL encapsulates the ongoing abductive solution in an abductive context, which is relayed from one subgoal to another. In many problems, it is often the case that one needs to access the ongoing abductive solution in order to manipulate it dynamically, e.g. to filter abductive solutions using preferences, or eliminate so-called *nogood* combinations (those known to violate constraints). But since it is encapsulated in an abductive context, and such a context is only introduced in the transformed program, the only way to accomplish it would be to modify directly the transformed program rather than the original problem representation. This is inconvenient and clearly unpractical when we deal with real world problems with a huge number of rules.

The aforementioned issue is overcome by introducing the TABDUAL system predicate $process\_ongoing(P)$ that allows to access the ongoing abductive solution and to manipulate it, while also allowing to abduce further, using the rules of P. This system predicate is transformed by unwrapping it and adding an extra argument to P (besides the usual input and output context arguments) for the ongoing abductive solution.

Example 18. *Consider a fragment of an input program:*

$$q \leftarrow r, process\_ongoing(s). \qquad\qquad s(X) \leftarrow v(X).$$

Notice that, predicate s wrapped by $process\_ongoing/1$ has no argument; more precisely, one less argument than its definition, i.e. rule s on the right. The extra argument of rule s is indeed dedicated for the ongoing abductive solution. The tabled predicate q_{ab} in the transformed program is defined as follows:

$$q_{ab}(E) \leftarrow r([\,], T), s(T, T, E).$$

That is, $s/3$ now gets access to the ongoing abductive solution T from r$/2$, via its additional first argument. It still has the usual input and output contexts, T and E, respectively, in its second and third arguments. It indicates that, while manipulating the ongoing abduction solution, abduction may take place in s. Rule $s/1$ transforms as usual.

The system predicate $process\_ongoing/1$ permits modular mixes of abductive and non-abductive program parts. For instance, the rule of $s/1$ in P_3 may be defined by some predicates from the non-abductive program part, e.g. the rule of $s/1$ can be defined instead as:

$$s(X) \leftarrow prolog(preferred(X)), a(X).$$

where $a/1$ is an abducible and $preferred(X)$ defines, in the non-abductive program part, some preference rule on a given solution X.

6.7 Other implementation aspects

Various other aspects have also been considered in implementing TABDUAL:

- XSB's built-in predicate $numbervars/1$ is used to help writing variables, e.g. arguments of a predicate, in transformed programs. This is to avoid the problem of mixing of variables writing due to stack expansion (or garbage collection), a bug that occurs in most Prolog systems [49]. This problem particularly arises when we deal with rather big input programs.

- The list of abductive solutions is represented using two separate lists: the lists of positive and negative parts. This enables faster consistency checking of abductive solutions, in predicates $insert\_abducible/3$ and $produce\_context/3$. That is, to check consistency with respect to a literal, only the list of literals with different polarity is inspected; there is no need to traverse all literals. Moreover, both lists are ordered, in order to improve efficiency.

- The second layer dual rules are defined by giving priority to abducibles. For instance, given rule $p \leftarrow q, a$ (where a is an abducible), the first rule for p^{*1} will be $p^{*1} \leftarrow not\_a$, instead of $p^{*1} \leftarrow not\_q$ (even though, in the body of the corresponding positive rule, a comes later than q). In this way, it allows obtaining abductive solutions to negative goals earlier: not_a is returned first before not_q is invoked (the latter could involve a deep derivation before it successfully abduces a solution). Also, since the abducible will be required anyway, giving it priority may constrain earlier any solutions. Of course, care has to be taken when we deal with rules having variables, in particular concerning grounding issues (cf. Section 6.2.1). Knowledge of shared variables in the body, and whether they are local or not, may help in this case. Furthermore, the use of a domain predicate for abducibles may come in handy.

- When a program contains NLoN, the dual rules of some predicates are also tabled. These are the predicates that appear as negative subgoals in the bodies of rules. Recall the definition of q_{ab}, in Section 6.4.3, where rules not_p_{tu} are introduced for the negative goal $not\ p$ that appears in the body of rule q. Predicate not_p_{tu} is in turn defined by $not\_p/1$; the latter predicate is defined by invoking the dual rules of p: in that example, $p^{*1}/2$ (line 4). By tabling $p^{*1}/2$, its recomputation, when it is subsequently invoked as the last subgoal of the not_p_{tu}'s second rule (line 3), can be avoided.

7 Evaluation of TABDUAL

We evaluate TABDUAL from various standpoints:

1. Our first evaluation (Section 7.2) aims at evaluating the benefit of tabling abductive solutions.

2. We evaluate in Section 7.3 the relative worth of three approaches of the dual transformation (as discussed in Section 6.5), viz., STANDARD, BY-NEED(EAGER), BY-NEED(LAZY).

3. We touch upon the evaluation of tabling nogoods of subproblems in abduction, in Section 7.4.

4. Finally, in Section 7.5 we evaluate TABDUAL, implementing the technique discussed in Section 6.4, for programs with loops, and compare the results with those returned by the ABDUAL meta-interpreter of [5].

In the first two evaluations, examples from declarative debugging are employed, which are now characterized as abduction [39], rather than as belief revision [29, 30]. We specifically consider two cases of declarative debugging: missing solutions and incorrect solutions, in the evaluation of Sections 7.2 and 7.3, respectively. Before discussing the aforementioned four evaluations, we revisit in Section 7.1 our declarative debugging of definite logic programs viewed as abduction. For normal logic programs, the reader is referred to [39].

7.1 Declarative debugging of definite logic programs

Example 19. *Consider program P_{10} [29] as a buggy program:*

$$a(1). \qquad a(X) \leftarrow b(X), c(Y,Y).$$
$$b(2). \qquad b(3). \qquad c(1,X). \qquad c(2,2).$$

7.1.1 Incorrect solutions

Suppose that $a(3)$ is an incorrect solution. To debug its cause, the program is first processed using the transformation in [30], by adding default literal $not\ incorrect(i, [X_1, \ldots, X_n])$ to the body of each i-th rule of P_{10}, to defeasibly assume their correctness by default, where n is the rule's arity and X_is, for $1 \leq i \leq n$, its head arguments. This yields program P'_{10}:

$$\begin{array}{ll} a(1) \leftarrow not\ incorrect(1,[1]). & a(X) \leftarrow b(X), c(Y,Y), not\ incorrect(2,[X]). \\ b(2) \leftarrow not\ incorrect(3,[2]). & b(3) \leftarrow not\ incorrect(4,[3]). \\ c(1,X) \leftarrow not\ incorrect(5,[1,X]). & c(2,2) \leftarrow not\ incorrect(6,[2,2]). \end{array}$$

In terms of abduction, one can envisage $incorrect/2$ as an abducible. To express, while debugging, that $a(3)$ is an incorrect solution, we add to P'_{10} an IC: $\leftarrow a(3)$. Running TABDUAL on P'_{10} returns three solutions as the possible sufficient causes of the incorrect solution:

$$[incorrect(2,[3])],\ [incorrect(4,[3])],\ [incorrect(5,[1,1]), incorrect(6,[2,2])].$$

7.1.2 Missing solutions

Suppose $a(5)$ should be a solution of P_{10}, but is missing. To find this bug, P_{10} is transformed [29] by adding to each predicate p/n a rule:

$$p(X_1, \ldots, X_n) \leftarrow missing(p(X_1, \ldots, X_n)).$$

That is, P_{10} is transformed into P''_{10} that contains all rules from P_{10} plus three new rules:

$$a(X) \leftarrow missing(a(X)). \quad b(X) \leftarrow missing(b(X)). \quad c(X,Y) \leftarrow missing(c(X,Y)).$$

Similarly to before, $missing/1$ can be viewed as an abducible. But now, to express that we miss $a(5)$ as a solution, we add to P''_{10} an IC: $\leftarrow not\ a(5)$. TABDUAL returns the three abductive solutions on P''_{10} as the causes of missing solution $a(5)$ in P_{10}:

$$[missing(a(5))],\ [missing(b(5))],\ [missing(b(5)), missing(c(X,X))].$$

Differently from [29,30], where minimal solutions are targeted, TABDUAL also returns non-minimal solution $[missing(b(5)), missing(c(X,X))]$. Finding minimal abductive solutions is not always desired – here bugs may well not be minimal and, in this case, TABDUAL allows one to identify and choose those bugs that satisfice so far, and to continue searching for more solutions if needed.

Next, we discuss the evaluation of TABDUAL. The experiments in all evaluations were run under XSB-Prolog 3.3.7 on a 2.26 GHz Intel Core 2 Duo with 2 GB RAM. The time indicated in all results refers to the CPU time (as an average of several runs) to aggregate all abductive solutions, unless otherwise stated.

7.2 Evaluation of tabling abductive solutions

The first evaluation aims at ascertaining the relative benefit of TABDUAL's main feature, i.e. tabling abductive solutions. We employ the case of missing solutions from declarative debugging (Section 7.1.2). Consider program P_{Eval} below as buggy:

$$\begin{array}{ll}
q_0(0,1). & q_0(X,0). \\
q_1(1). & q_1(X) \leftarrow q_0(X,X). \\
q_2(2). & q_2(X) \leftarrow q_1(X). \\
q_3(3). & q_3(X) \leftarrow q_2(X). \\
\vdots & \vdots \\
q_{1000}(1000). & q_{1000}(X) \leftarrow q_{999}(X).
\end{array}$$

Following the transformation for debugging missing solutions in Section 7.1.2, program P_{Eval} transforms into an abductive logic program P''_{Eval} that contains all rules of P_{Eval} plus rules below (with abducible $missing/1$):

$$\begin{array}{l}
q_0(X,Y) \leftarrow missing(q_0(X,Y)) \\
q_1(X) \leftarrow missing(q_1(X)) \\
q_2(X) \leftarrow missing(q_2(X)) \\
\vdots \\
q_{1000}(X) \leftarrow missing(q_{1000}(X))
\end{array}$$

Program P''_{Eval} thus serves as the input for TABDUAL.

In order to evaluate tabling abductive solutions, a set of benchmarks is created that corresponds to a set of missing solutions in the buggy program P_{Eval}. More precisely, we want to debug this program for missing solutions $q_m(1001)$, where $m \in \{100, 200, \dots, 1000\}$. Recall from Section 7.1.2, that missing a solution $q_m(1001)$, for a particular m, is expressed by adding IC $\leftarrow$ *not* $q_m(1001)$ to the program P''_{Eval}. In our experiments, we alternatively pose query $q_m(1001)$ for each m, which is equivalent to satisfying its corresponding IC.

This set of benchmarks is suitable for showing the benefit of tabling abductive solutions. It is easy to verify that debugging missing solution $q_{100}(1001)$ obtains 101 abductive solutions: $[missing(q_{100}(1001))]$,$[missing(q_{99}(1001))]$, $\dots$, $[missing(q_1(1001))]$, $[missing(q_0(1001, 1001))]$. By employing tabled abduction, the causes of missing solution $q_{200}(1001)$: $[missing(q_{200}(1001))]$, $[missing(q_{199}(1001))]$, $\dots$, $[missing(q_1(1001))]$, $[missing(q_0(1001, 1001))]$, is subsequently found without recomputing those 101 abductive solutions priorly obtained from the query $q_{100}(1001)$. This advantage is accumulatively enjoyed by subsequent values of m ($m = 300, \dots, 1000$).

Since in this first evaluation we focus on the benefit of tabling abductive solutions, we consider a variant of TABDUAL (with the same underlying implementation) where its feature of tabled abduction is stripped off. That is, by disabling the table declarations of abductive predicates $q_{i_{ab}}$, for every predicate q_i, $0 \leq i \leq 1000$. We refer below this variant as TABDUAL(NO TABLING).

Figure 2 shows the time required in the abduction stage of TABDUAL and TABDUAL(NO TABLING), by consecutively evaluating query $q_m(1001)$, where $m \in \{100, 200, \dots, 1000\}$. Note that, the time needed in the transformation stage between the two variants do not differ, as both of them rely on the same implementation of the transformation. The result reveals that, with some little cost of tabling abductive solutions in earlier values of m (i.e. $m \leq 300$), TABDUAL consistently outperforms its counterpart in performance. Tabling pays off for subsequent values of m in TABDUAL, as greater m may reuse tabled abductive solutions of smaller m, due to the consecutive evaluation of queries. Moreover, TABDUAL scales better than TABDUAL(NO TABLING), i.e. as the values of m grows, its abduction time increases slower than the other. We also observe, that TABDUAL's abduction time tends to grow linearly, whereas that of its counterpart exponentially.

7.3 Evaluation of the dual transformations

For this evaluation, we resort to the same buggy program P_{Eval} used in the evaluation of tabling abductive solutions (cf. Section 7.2). But instead of debugging the program for missing solutions, we consider the case of incorrect solutions.

The transformation for debugging incorrect solutions (Section 7.1.1) turns P_{Eval} into an

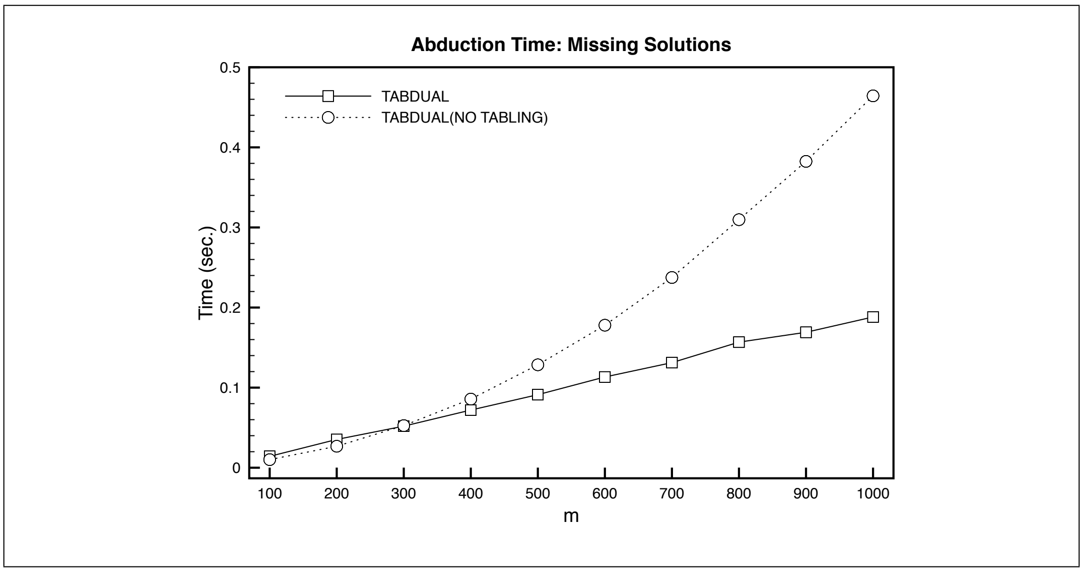

Figure 2: The abduction time for finding the causes of missing solutions $q_m(1001)$, where $m \in \{100, 200, \ldots, 1000\}$ with respect to program P_{Eval}.

abductive logic program P'_{Eval} below (with abducible $incorrect/2$ abbreviated as $inc/2$):

$$\begin{array}{ll}
q_0(0,1) \leftarrow not\ inc(1,[0,1]). & q_0(X,0) \leftarrow not\ inc(2,[X,0]). \\
q_1(1) \leftarrow not\ inc(3,[1]). & q_1(X) \leftarrow not\ inc(4,[X]), q_0(X,X). \\
q_2(2) \leftarrow not\ inc(5,[2]). & q_2(X) \leftarrow not\ inc(6,[X]), q_1(X). \\
q_3(3) \leftarrow not\ inc(7,[3]). & q_3(X) \leftarrow not\ inc(8,[X]), q_2(X). \\
\vdots & \vdots \\
q_{1000}(1000) \leftarrow not\ inc(2001,[1000]). & q_{1000}(X) \leftarrow not\ inc(2002,[X]), q_{999}(X).
\end{array}$$

Let us consider a set of incorrect solutions $q_m(0)$, for $m \in \{100, 200, \ldots, 1000\}$, with respect to P_{Eval}. The set of benchmarks for the purpose of this evaluation amounts to debugging each of these incorrect solutions. Recall from Section 7.1.1, that debugging an incorrect solution $q_m(0)$ for a specific m is realized by adding to program P'_{Eval} an IC $\leftarrow q_m(0)$. In our experiments, we alternatively pose query $not\ q_m(0)$ for each m, which is equivalent to satisfying its corresponding IC. Because finding the causes of incorrect solution bugs $q_m(0)$ amounts to satisfying negative goal $not\ q_m(0)$, this debugging case appropriately serves as benchmarks for evaluating dual transformations: such a negative query should be answered using dual rules computed by TABDUAL from program P'_{Eval}.

Since our aim is this particular evaluation focuses on the relative worth of the dual transformation by-need, we consider the three versions of dual transformations (Section 6.5) implemented in TABDUAL, viz.: STANDARD, BY-NEED(EAGER), and BY-NEED(LAZY). By experimenting on these three variants, we obtain results as follows:

1. It takes 1.164 seconds for TABDUAL that implements either BY-NEED(EAGER) or BY-NEED(LAZY), whereas the implementation STANDARD 1.674 seconds. It is obvious that BY-NEED(EAGER) and BY-NEED(LAZY) require less transformation time than STANDARD, since they do not produce all dual rules in advance as STANDARD does.

   Take an example q_2. The second layer dual rules produced, in advance, by STANDARD are: (apart from the dual rule that disunifies arguments $q_2^{*1}(X, I, I) \leftarrow X \neq 2$):

$$\begin{array}{lcl} q_2^{*1}(2, I, O) & \leftarrow & \mathit{incorrect}(5, [2], I, O). \\ q_2^{*2}(X, I, O) & \leftarrow & \mathit{incorrect}(6, [X], I, O). \\ q_2^{*2}(X, I, O) & \leftarrow & \mathit{not\_}q_1(X, I, O). \end{array}$$

   whereas BY-NEED(EAGER) and BY-NEED(LAZY) just produce their skeleton:

$$\begin{array}{lcl} q_2^{*1}(2, I, O) & \leftarrow & \mathit{dual}(1, q_2(2), I, O). \\ q_2^{*2}(X, I, O) & \leftarrow & \mathit{dual}(2, q_2(X), I, O). \end{array}$$

   and only construct dual rules, by-need, during abduction.

2. In terms of the number of dual rules (apart from those dual rules defined by disunifying arguments), STANDARD creates 3002 second layer dual rules during the transformation, regardless their need. On the other hand, BY-NEED(EAGER) and BY-NEED(LAZY) create only 2002 (skeleton of) second layer dual rules, shown above.

3. During abduction, BY-NEED(EAGER) and BY-NEED(LAZY) construct only 60% of second layer dual rules produced by STANDARD. That is, 40% of dual rules constructed by STANDARD are actually not needed. Take again the above case of q_2. In finding the causes of an incorrect solution $q_m(0)$ (for some m), query $\mathit{not}\ q_m(0)$ does not need to invoke dual rule $q_2^{*1}(2, I, O) \leftarrow \mathit{incorrect}(5, [2], I, O)$, as they are bound to fail. In this case, the other dual rule that disunifies arguments, viz., $q_2^{*1}(X, I, I) \leftarrow X \neq 2$, already succeeds (as X is instantiated by 0) and is sufficient to satisfy the q_2^{*1} part.

Point (1) above tells us that the dual transformation by-need, either BY-NEED(EAGER) or BY-NEED(LAZY), requires less time in the transformation stage compared to STANDARD. On the other hand, as explained in Section 6.5 (cf. explanation for Example 17), predicate $\mathit{dual}/4$, in the rule skeleton of point (2) above, constructs generic dual rules and instantiates them only in the abduction stage. Figure 3 shows the performance of each dual transformation in terms of time needed in the abduction stage of consecutively running query

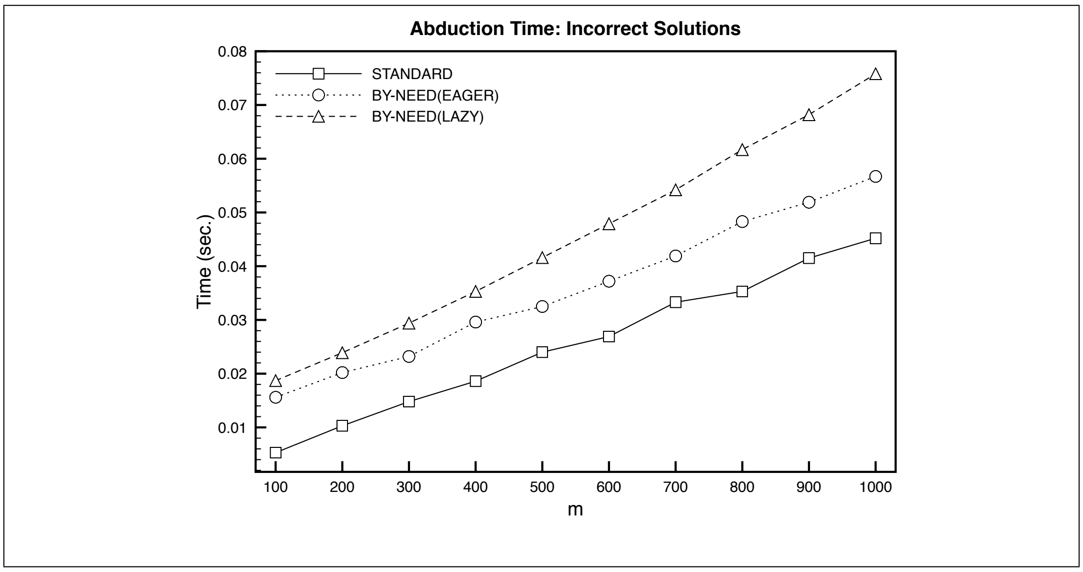

Figure 3: The abduction time for finding the causes of incorrect solutions $q_m(0)$, where $m \in \{100, 200, \ldots, 1000\}$ with respect to program P_{Eval}.

not $q_m(0)$ with different values of $m \in \{100, 200, \ldots, 1000\}$. Note that, the time needed for the transformation is not included in the figure, as we want to measure the cost incurred by BY-NEED(EAGER) and BY-NEED(LAZY) in the abduction stage. In Figure 3, we observe that STANDARD is faster than BY-NEED(EAGER) and BY-NEED(LAZY). This is expected, due to the overhead incurred for computing dual rules on-the-fly, by need, during the abduction stage of BY-NEED(EAGER) and BY-NEED(LAZY).

On the other hand, if we consider the whole process (i.e., the time needed in the transformation plus the abduction stages), the time overhead of BY-NEED(EAGER) and BY-NEED(LAZY) in the abduction stage is well-compensated by their transformation time. That is, the time for the whole process (transformation plus abduction) of STANDARD is 1.929 seconds, whereas BY-NEED(EAGER) and BY-NEED(LAZY) need 1.521 and 1.620 seconds, respectively.

In this evaluation scenario, where all abductive solutions are aggregated, the performance of BY-NEED(LAZY) is slightly worse than BY-NEED(EAGER). This can be explained by the extra maintenance of the tracking information needed for the explicit memoization in BY-NEED(LAZY). It may as well explain that the time gap between BY-NEED(LAZY) and BY-NEED(EAGER) is wider as m grows, meaning more dual rules are stored in the trie. Nevertheless, BY-NEED(LAZY) returns the first abductive solution much faster than BY-NEED(EAGER), e.g. at $m = 1000$ the lazy one needs 0.0003 seconds, whereas the eager one 0.0146 seconds. Aggregating all solutions may not be a realistic scenario in abduction as one cannot wait indefinitely for all solutions, whose number might even be infinite. Instead, one chooses a solution that satisfices so far, and may continue searching for more, if needed. In that case, it seems reasonable that the lazy dual rules computation may be competitive against the eager one. Nevertheless, the two approaches are available as options for TABDUAL customization.

7.4 Evaluation of tabling nogoods of subproblems

The technique of recording nogoods of subproblems, i.e. inconsistent solutions of subproblems that cannot be extended to derive any solution of the given problem, has been employed in truth maintenance systems [10,13], constraint satisfaction problems [45], SAT solvers [27], and in answer set solvers [19], to help prune search space.

We employ TABDUAL and show that tabling abductive solutions can be appropriate for tabling nogoods of subproblems. For this purpose, we consider the well-known N-queens problem, where abduction is used to find safe board configurations of N queens. The problem is represented in TABDUAL as follows:

$$\begin{array}{lcl} q(0,N). & & \\ q(M,N) & \leftarrow & M>0,\ q(M-1,N),\ d(Y),\ pos(M,Y), \\ & & process\_ongoing(not\ conflict). \\ & & \\ conflict(Conf) & \leftarrow & prolog(conflicting(Conf)). \end{array}$$

and the query is $q(N,N)$ for N queens. Here, $pos/2$ is the abducible representing the position of a queen, and $d/1$ is a column generator predicate, available as facts $d(i)$ for $1 \leq i \leq N$. Predicate $conflicting/1$ is defined in a non-abductive program module, to check whether the ongoing board configuration $Conf$ of queens is conflicting. By scaling up the problem, i.e. increasing the value of N, we aim at evaluating the scalability of TABDUAL, concentrating on tabling nogoods of subproblems (essentially, tabling nogoods for use by ongoing abductive solutions); in this case, it means tabling conflicting configurations of queens.

Since this benchmark is used to evaluate the benefit of *tabling* nogoods of subproblems (as abductive solutions), and *not* the benefit of the dual by-need improvement, we evaluate TABDUAL compared to its non-tabling variant TABDUAL(NO TABLING), as in Section 7.2. The transformation time of the problem representation is similar for both of them, i.e. around 0.003 seconds. Figure 4 shows abduction time for N queens, $4 \leq N \leq 11$. The reason that TABDUAL performs worse than TABDUAL(NO TABLING) is that the conflict constraints in the N-queens problem are quite simple, i.e. consist of only column and diagonal checking. It turns out that tabling such simple conflicts does not pay off, that the cost of tabling overreaches the cost of Prolog recomputation. But what if we increase the

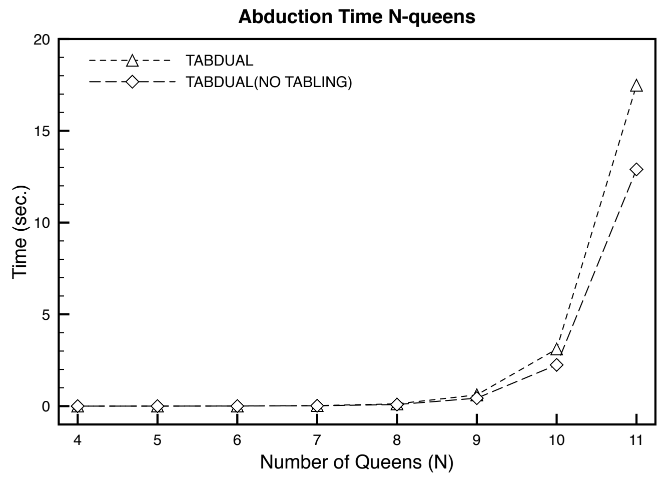

Figure 4: The abduction time of different N queens.

complexity of the constraints, e.g. adding more queen's attributes (colors, shapes, etc.) to further constrain its safe positioning?

Figure 5 shows abduction time for 11 queens with increasing complexity of the conflict constraints. To simulate different complexity, the conflict constraints are repeated m number of times, where m varies from 1 to 400. It shows that TABDUAL's performance is remedied and, benefitting from tabling the ongoing conflict configurations, it consistently surpasses the performance of TABDUAL(NO TABLING), showing increasing improvement as m increases, up to 15% for $m = 400$. That is, it is scale consistent with respect to the complexity of the constraints.

7.5 Evaluation of programs with loops

We also evaluate TABDUAL for programs with loops in the presence of tabled abduction, as detailed in Section 6.4. For that purpose, we employ a set of ground programs with various combination of loops, many of which cover difficult known cases of such programs. The test-suite has previously been used in evaluating the ABDUAL meta-interpreter in [5].

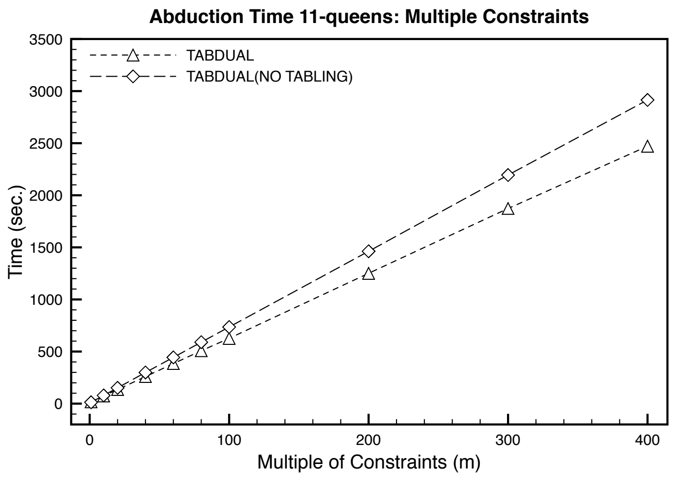

Figure 5: The abduction time of 11 queens with increasing complexity of conflict constraints.

Consider the following six ground programs from the test-suite:

$p_0 \leftarrow q_0.$	$p_3 \leftarrow q_3.$	$p_4 \leftarrow q_4.$
$p_0 \leftarrow a.$	$q_3 \leftarrow not\ r_3.$	$q_4 \leftarrow p_4.$
$q_0 \leftarrow p_0.$	$r_3 \leftarrow p_3.$	$q_4 \leftarrow not\ a,\ not\ b.$
$q_0 \leftarrow b.$		

$p_8 \leftarrow not\ q_8,\ a.$	$p_{11} \leftarrow not\ q_{11},\ a.$
$q_8 \leftarrow not\ p_8.$	$q_{11} \leftarrow p_{11},\ not\ a.$
$q_8 \leftarrow b.$	

where a and b are abducibles.

We provide a comparison of the results returned by both implementations, focusing particularly on those that differ. Table 1 lists the answers of the given queries to the corresponding programs, returned by TABDUAL and the ABDUAL meta-interpreter. It shows that TABDUAL returns correct answers according to the ABDUAL theory which underpins it, even now in the presence of tabled abduction. On the other hand, in the cases we are showing here, the ABDUAL meta-interpreter does not correctly return answers as computed by its theory. The answers returned by TABDUAL for queries in Table 1 are explained as follows:

- For query $not\ p_0$, $[not\ a, not\ b]$ should be the only solution, because $not\ p_0$ succeeds

Table 1: Answers by TABDUAL vs. ABDUAL meta-interpreter.

Queries	TABDUAL	ABDUAL meta-interpreter
$not\ p_0$	$[not\ a, not\ b]$	$[not\ a, not\ b], [not\ a]$
p_3	[ ] *undefined*	[ ]
$not\ p_3$	[ ] *undefined*	[ ]
$not\ p_4$	$[a], [b]$	$[a], [b], [a, b]$
q_8	[ ], $[not\ a], [b]$	$[not\ a], [b]$
$not\ q_{11}$	$[a], [not\ a]$	[ ], $[a], [not\ a]$

by abducing $not\ a$ *and* failing q_0. To fail q_0, $not\ b$ has to be abduced *and* p_0 has to fail. Here, there is a positive loop on negation between $not\ p_0$ and $not\ q_0$, so the query succeeds and gives the solution $[not\ a, not\ b]$ as the only solution.

- For queries p_3 and $not\ p_3$, unlike answers returned by the ABDUAL meta-interpreter, TABDUAL returns *undefined* (and abduces nothing) as expected, due to the negative loops over negation.

- Query $not\ p_4$ shows that TABDUAL does less abduction than the ABDUAL meta-interpreter, by abducing a or b only, but not both.

- For query q_8, TABDUAL has an additional solution [ ], i.e. nothing is abduced, making particularly a false and consequently p_8 false (or, $not\ p_8$ true). Thus, query q_8 is true (by its first rule) under this solution, which is missing in answers returned by the ABDUAL meta-interpreter.

- For query $not\ q_{11}$, the first solution is obtained by abducing a to fail q_{11}. Another way to fail q_{11} is to fail p_{11}, which gives another solution, by abducing $not\ a$. These are the only two abductive solutions which are returned by TABDUAL and follows correctly the definition of abductive solutions. There is no direct positive loop involving q_{11} in the program, hence $not\ q_{11}$ will never succeed with the [ ] abductive solution, as returned by the ABDUAL meta-interpreter.

In addition to ground programs, we also evaluate TABDUAL on non-ground programs, i.e. programs having variables (with or without loops), which is not afforded by ABDUAL. The latter system does not allow rules having variables, i.e. rules with variables in a program have first to be ground with respect to the Herbrand universe (like in answer set programming systems). The complete test-suite and the evaluations results are detailed in Appendix B.

8 Concluding remarks

We have addressed the issue of tabling abductive solutions, in a way that they can be reused from one abductive context to another. We do so by resorting to a program transformation approach, resulting in a tabled abduction prototype, TABDUAL, implemented in XSB Prolog. TABDUAL is underpinned by the ABDUAL theory and employs its dual transformation, which allows to more efficiently handle the problem of abduction under negative goals. In TABDUAL, abducibles are treated much like terminals in grammars, with an extra argument for input and another for output abductive context accumulation. A few other original innovative and pragmatic techniques are employed to handle programs with variables and loops, as well as to make TABDUAL more efficient and flexible. It has been evaluated with various objectives in mind, in order to show the benefit of tabled abduction and to gauge its suitability for likely applications. An issue that we have touched upon in the TABDUAL evaluation is that of tabling nogoods of subproblems in the context of tabled abduction, and how it may improve performance and scalability. The other evaluation result reveals that each approach of the dual transformation by-need may be suitable for different situations, i.e. both approaches, BY-NEED(EAGER) or BY-NEED(LAZY), are options for TABDUAL customization.

8.1 Related work

There have been a plethora of work on abduction in logic programming, cf. [12, 22] for a survey on this line of work. But, with the exception of ABDUAL [4], we are not aware of any other efforts that have addressed the use of tabling in abduction for abductive normal logic programs, which may be complicated with loops. Like ABDUAL, we use the dual transformation and rely on the same theoretic underpinnings, but ABDUAL does not allow variables in rules. The reader is referred to Section 5.2 of [4] on how the dual transformation and its properties relate to other works.

From the implementation viewpoint, tabling has only been employed in ABDUAL limitedly, i.e. to table its meta-interpreter, which in turn allows abduction to be performed (also in the presence of loops in a program), but it does not address at all the specific issues raised by the desirable reuse of tabled abductive solutions. TABDUAL generates a self-sufficient program transform (plus system predicates), which employs no meta-interpreter, even in the presence of loops in programs.

Our approach also differs from that of [2]. Therein, abducibles are coded as odd loops, it is compatible with and uses constructive negation, and it involves manipulating the residual program. It suffers from a number of problems, which it identifies, in its Sections 5 and 6, and its approach was not pursued further.

TABDUAL does not concern itself with constructive negation, like NEGABDUAL [3] and its follow-up [7]. NEGABDUAL uses abduction to provide constructive negation plus abduction, by making the disunification predicate an abducible. Again, it does not concern itself with the issues of tabled abductive solution reuse, which is the main purpose of TABDUAL. However, because of its constructive negation ability, NEGABDUAL can deal with problems that TABDUAL does not. Consider program below, with no abducibles, just to illustrate the point of constructive negation induced by dualization:

$$p(X) \leftarrow q(Y). \qquad q(1).$$

In NEGABDUAL, the query $not\ p(X)$ will return a qualified '*yes*', because it is always possible to solve the constraint $Y \neq 1$, as long as one assumes there are at least two constants in the Herbrand Universe. Distinct from NEGABDUAL, TABDUAL answers '*no*' to $not\ p(X)$. It is correct, in the absence of conditional answers; the former answer is afforded only by having constructive negation in place. It is interesting to explore in the future, whether TABDUAL can be extended to take care such constraints (as abducibles), given that XSB supports low-level constraint handling through attributed variables, and that attributed variables can be tabled in XSB.

TABDUAL, being implemented in XSB, is based on the WFS, which enjoys the relevance property, induced by the top-down query-oriented procedure, solely for finding the relevant abducibles and their truth value. This is not the case with the bottom-up approaches for abduction, e.g. [43], where stable models for computing abductive explanations, not necessarily related to an observation, are constructed. This disadvantage of the bottom-up TMS approach is in fact later avoided by adding a top-down procedure, as in [44]. TABDUAL also allows dealing with odd loops in programs because of its 3-valued program semantics, whilst retaining 2-valued abduction and the use of integrity constraints. This is not enjoyed by the bottom-up approach and its 2-valued implementation.

The tabling technique, within the context of statistical abduction, is employed in [42]. But it concerns itself with probabilistic logic programs, whereas TABDUAL concerns abductive normal logic programs. Moreover, the tabling technique in [42] imposes the so-called 'acyclic support condition', a constraint that does not allow loops in a program, which pose no restrictions at all in TABDUAL. Tabling is also used recently in PITA [35], for statistical abduction. Though PITA is also based on the Well-Founded Semantics like TABDUAL, tabling (in particular its feature, answer subsumption) applies specifically to probabilistic logic programs, e.g. to compute the number of different explanations for a subgoal (in terms of Viterbi path), which is not our concern in TABDUAL, and thus does not employ the dual transformation and other techniques described here.

8.2 Future work

TABDUAL still has much room for improvement. Future work will consist in continued exploration of our applications of abduction, which will provide feedback for system improvement. Tabled abduction may benefit from answer subsumption [47] in tabling abductive solutions to deal with redundant explanations, in the sense that it suffices to table only smaller abductive solutions (with respect to the subset relation). Another potential XSB feature to look into is the applicability of interning ground terms [52] for tabling abductive solutions, which are ground, and study how extra efficiency may be gained from it.

The implementation technique of BY-NEED(LAZY) consists in operational details that are facilitated by XSB's trie manipulation predicates, to simulate the batched-like table scheduling within XSB's current default local table scheduling. In order to have a more transparent implementation of those operations, it is desirable that XSB permits a mixture in using batched and local table scheduling strategies, or alternatively, stopping the evaluation at some first answers to a subgoal within the currently default local table scheduling.

In another related research line, it would be interesting to explore whether abduction could be used with XSB's partial support of Transaction Logic in its *storage* module. There is a certain similarity in that Transaction Logic rules may cause updates in a manner that is reminiscent of abduction, although Transaction Logic also allows a commit.

TABDUAL opens up a potential joint use with other non-monotonic LP features, having their own tabling requirements and attending benefits. In [40], we combine tabled abduction with LP updating [37]. The latter employs incremental tabling to automatically propagate updates bottom-up, being triggered by a top-down query. The implemented system of this joint LP abduction and updating is part and parcel of our research to employ LP in agent moral reasoning [41]. We envisage a couple of applications using this joint system, and tabled abduction particularly, in this field. For one, people often engage counterfactual thoughts in moral situations, where their function is not just evaluative (to correct wrong behavior in the past), but also reflective (to simulate possible alternatives for a careful consideration before making a moral decision) [15]. In [33], we propose a LP approach to model counterfactual reasoning, based on Pearl's structural approach [28] (abstaining from probability), by benefitting from the aforementioned joint system. The role of abduction is this LP counterfactual approach is to hypothesize background conditions from given evidences or observations, a required step in Pearl's approach, so as to provide an initial abductive context for the counterfactual being evaluated. In this case, tabled abduction permits reusing this initial abductive context in another subsequent context acquired by the agent during its life cycle.

Another morality related application we consider is the dual-process model in moral judgment [8] that stresses the interaction between deliberative and reactive behaviors in delivering moral judgment. Given that abductive solutions represent some actions according to a specific moral principle, tabled abduction may play several roles. For one thing, it allows an agent to deliver an action in exactly the same context without repeating the same deliberative reasoning, thus simulating a form of low-level reactive behavior (realized by system-level tabling) of the dual-process model. For another, in a dynamic environment (hence the need of LP updating), the agent may later be required to achieve new goals in addition to the former ones, due to a moral principle it follows. While achieving these new goals requires deliberative reasoning, the decisions that have been abduced for former goals can immediately be retrieved from the table and are subsequently involved in (as the context of) the deliberative reasoning for the new goals. It thus provides a computational model of collaborative interaction between deliberative and reactive reasoning in the dual-process model.

Abduction is by now a staple feature of hypothetical reasoning and non-monotonic knowledge representation. It is already mature enough in its concept, deployment, applications, and proof-of-principle, to warrant becoming a run-of-the-mill ingredient in a Logic Programming environment. We hope this work will lead, in particular, to an XSB System that can provide its users with specifically tailored tabled abduction facilities, by migrating some of TABDUAL's features to its engine level.

References

[1] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, and P. Torroni. Security protocols verification in abductive logic programming. In *6th Int. Workshop on Engineering Societies in the Agents World (ESAW)*, volume 3963 of *LNCS*. Springer, 2005.

[2] J. J. Alferes and L. M. Pereira. Tabling abduction. 1st Intl. Ws. Tabulation in Parsing and Deduction (TAPD'98), `http://centria.di.fct.unl.pt/~lmp/publications/online-papers/tapd98abd.ps.gz`, 1998.

[3] J. J. Alferes and L. M. Pereira. NEGABDUAL meta-interpreter. Available from `http://centria.di.fct.unl.pt/~lmp/software/contrNeg.rar`, 2007.

[4] J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, 2004.

[5] J. J. Alferes, L. M. Pereira, and T. Swift. ABDUAL meta-interpreter. Available from `http://www.cs.sunysb.edu/~tswift/interpreters.html`, 2004.

[6] J. Balsa, V. Dahl, and J. G. Pereira Lopes. Datalog grammars for abductive syntactic error diagnosis and repair. In *Proc. Natural Language Understanding and Logic Programming Workshop*, 1995.

[7] V. P. Ceruelo. Negative non-ground queries in well founded semantics. Master's thesis, Universidade Nova de Lisboa, 2009.

[8] F. Cushman, L. Young, and J. D. Greene. Multi-system moral psychology. In J. M. Doris, editor, *The Moral Psychology Handbook*. Oxford University Press, 2010.

[9] C. V. Damásio and L. M. Pereira. Abduction over 3-valued extended logic programs. In *Procs. 3rd. Intl. Conf. Logic Programming and Non-Monotonic Reasoning (LPNMR)*, volume 928 of *LNAI*, pages 29–42. Springer, 1995.

[10] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[11] M. Denecker and D. de Schreye. SLDNFA: An abductive procedure for normal abductive programs. In *Procs. of the Joint Intl. Conf. and Symp. on Logic Programming*. The MIT Press, 1992.

[12] M. Denecker and A. C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*. Springer Verlag, 2002.

[13] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[14] T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.

[15] K. Epstude and N. J. Roese. The functional theory of counterfactual thinking. *Personality and Social Psychology Review*, 12(2):168–192, 2008.

[16] K. Eshghi. Abductive planning with event calculus. In *Proc. Intl. Conf. on Logic Programming*. The MIT Press, 1988.

[17] T. H. Fung and R. Kowalski. The IFF procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.

[18] J. Gartner, T. Swift, A. Tien, C. V. Damásio, and L. M. Pereira. Psychiatric diagnosis from the viewpoint of computational logic. In *Procs. 1st Intl. Conf. on Computational Logic (CL 2000)*, volume 1861 of *LNAI*, pages 1362–1376. Springer, 2000.

[19] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Procs. 20th Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, 2007.

[20] K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *J. of Logic Programming*, 27(2):107–136, 1996.

[21] J. R. Josephson and S. G. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge U. P., 1995.

[22] A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford U. P., 1998.

[23] A. C. Kakas and P. Mancarella. Knowledge assimilation and abduction. In *Intl. Workshop on Truth Maintenance*, ECAI'90, 1990.

[24] A. C. Kakas and A. Michael. An abductive-based scheduler for air-crew assignment. *J. of Applied Artificial Intelligence*, 15(1-3):333–360, 2001.

[25] R. Kowalski and F. Sadri. Abductive logic programming agents with destructive databases. *Annals of Mathematics and Artificial Intelligence*, 62(1):129–158, 2011.

[26] P. Lipton. *Inference to the Best Explanation*. Routledge, 2001.

[27] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, 1996.

[28] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge U. P., 2009.

[29] L. M. Pereira, C. V. Damásio, and J. J. Alferes. Debugging by diagnosing assumptions. In *Automatic Algorithmic Debugging*, volume 749 of *LNCS*, pages 58–74. Springer, 1993.

[30] L. M. Pereira, C. V. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal in logic programs. In *Progress in Artificial Intelligence*, volume 727 of *LNAI*, pages

183–197. Springer, 1993.

[31] L. M. Pereira, P. Dell'Acqua, A. M. Pinto, and G. Lopes. Inspecting and preferring abductive models. In K. Nakamatsu and L. C. Jain, editors, *The Handbook on Reasoning-Based Intelligent Systems*, pages 243–274. World Scientific Publishers, 2013.

[32] L. M. Pereira and A. Saptawijaya. Abductive logic programming with tabled abduction. In *ICSEA 2012*, pages 548–556. ThinkMind, 2012.

[33] L. M. Pereira and A. Saptawijaya. Counterfactuals in logic programming with applications to agent morality. Available from `http://centria.di.fct.unl.pt/~lmp/publications/online-papers/moral_counterfactuals.pdf`, 2015.

[34] O. Ray, A. Antoniades, A. Kakas, and I. Demetriades. Abductive logic programming in the clinical management of HIV/AIDS. In *Proc. 17th. European Conference on Artificial Intelligence*. IOS Press, 2006.

[35] F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming*, 11(4-5):433–449, 2011.

[36] A. Saptawijaya and L. M. Pereira. Implementing tabled abduction in logic programs. In *Procs. 16th Portuguese Intl. Conf. on Artificial Intelligence (EPIA)*, Doctoral Symposium on Artificial Intelligence (SDIA), 2013.

[37] A. Saptawijaya and L. M. Pereira. Incremental tabling for query-driven propagation of logic program updates. In *LPAR-19*, volume 8312 of *LNCS*, pages 694–709. Springer, 2013.

[38] A. Saptawijaya and L. M. Pereira. Towards practical tabled abduction in logic programs. In *16th Portuguese Conference on Artificial Intelligence (EPIA)*, LNAI. Springer, 2013.

[39] A. Saptawijaya and L. M. Pereira. Towards practical tabled abduction usable in decision making. In *KES-IDT 2013*, Frontiers of Artificial Intelligence and Applications (FAIA). IOS Press, 2013.

[40] A. Saptawijaya and L. M. Pereira. Joint tabling of logic program abductions and updates (Technical Communication of ICLP 2014). *Theory and Practice of Logic Programming, Online Supplement*, 14(4-5), 2014. Available from `http://arxiv.org/abs/1405.2058`.

[41] A. Saptawijaya and L. M. Pereira. The potential of logic programming as a computational tool to model morality. In R. Trappl, editor, *A Construction Manual for Robots' Ethical Systems: Requirements, Methods, Implementations*, Cognitive Technologies (forthcoming). Springer, 2015. `http://centria.di.fct.unl.pt/~lmp/publications/online-papers/ofai_book.pdf`.

[42] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. of Artificial Intelligence Research (JAIR)*, 15:391–454, 2001.

[43] K. Satoh and N. Iwayama. Computing abduction by using the TMS. In *Procs. 8th Intl. Conf. on Logic Programming (ICLP)*, pages 505–518. The MIT Press, 1991.

[44] K. Satoh and N. Iwayama. Computing abduction by using TMS and top-down expectation. *Journal of Logic Programming*, 44(1-3):179–206, 2000.

[45] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. In *Procs. 5th. Intl. Conf. on Tools with Artificial Intelligence (ICTAI)*, 1993.

[46] T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial*

Intelligence, 25(3-4):201–240, 1999.

[47] T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *JELIA 2010*, volume 6341 of *LNCS*, pages 300–312. Springer, 2010.

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

[49] T. Swift and D. S. Warren. Personal communications, February 2012, 2013.

[50] T. Swift, D. S. Warren, K. Sagonas, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, D. Saha, S. Dawson, and M. Kifer. *The XSB System Version 3.3.x Volume 1: Programmer's Manual*, 2012.

[51] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[52] D. Warren. Interning ground terms in XSB. In *Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS 2013)*, 2013.

A Proof of Theorem 2

Theorem 1. *Let P be an abductive normal logic program and $\mathcal{A}_P$ be the set of abducible atoms in P. Then $size(\tau(P)) < 9.size(P) + 4.|\mathcal{A}_\mathcal{P}|$.*

Proof. Let p_i be a predicate for which there are $m > 0$ rules in P with the total size $size(P|_{p_i})$, and $c \geq 0$ be the number of abducibles in the body of a rule of p_i.

- Since the abducibles in the body of a rule are moved from the body to abductive context (cf. point (1) of Definition 1), we have the size of $\tau'(P)$ as $size(\tau'(P)) = size(P) - c.rules(P)$.

- Since $\tau^+(p_i)$ for every defined $p_i \in P$ has three literals (cf. point (2) of Definition 1), we have the size of $\tau^+(P)$ as $size(\tau^+(P)) = 3.heads(P)$.

- For τ^-, we have two cases, based on Definition 2:

  1. By point 1(a), i.e. for p_i defined in P, the size of $\tau^-(p_i)$ will be $m + 1$. Therefore, the size of the transformed program in P by τ^- for all predicates defined in P will be $heads(P).(m + 1) = rules(P) + heads(P)$.
  2. By point 2, the size of the transformed program in P by τ^- for all predicates that have no definition in P will be $preds(P) - heads(P)$.

  Summing up the size from both cases, we have:
  $size(\tau^-(P)) = rules(P) + preds(P)$.

- For τ^*, the total size of rules with heads of the form $p^{*i}(\bar{t}_i, I, O)$, cf. point 1(b) of Definition 2, will be $2.(size(P|_{p_i}) - m)$. Since the size of the other rule, i.e. the one with the head $p^{*i}(\bar{X}, I, I)$, is two (for each i), the total size of $\tau^*(p_i)$ is $2.(size(P|_{p_i}) - m) + 2m = 2.size(P|_{p_i})$. Therefore, the size of the transformed program in P by τ^* for all predicates defined in P will be $size(\tau^*(P)) = \Sigma_{i=1}^{heads(P)} 2.size(P|_{p_i}) = 2.size(P)$.

- Finally for τ°, since the size of rules for each abducible atom is four, we have $size(\tau^\circ(P)) = 4.|\mathcal{A}_\mathcal{P}|$, where $|\mathcal{A}_\mathcal{P}|$ denotes the cardinality of $\mathcal{A}_\mathcal{P}$.

Note that $preds(P) \leq size(P)$, as also for $heads(P)$ and $rules(P)$. Thus:
$size(\tau(P)) = size(\tau'(P)) + size(\tau^+(P)) + size(\tau^-(P)) + size(\tau^*(P)) + size(\tau^\circ(P)) < 9.size(P) + 4.|\mathcal{A}_\mathcal{P}|$. □

$p_0 \leftarrow q_0.$	$p_1 \leftarrow not\ q_1, r_1.$	$p_2 \leftarrow q_2.$
$p_0 \leftarrow a.$	$r_1 \leftarrow not\ q_1, p_1.$	$q_2 \leftarrow r_2.$
$q_0 \leftarrow p_0.$	$q_1 \leftarrow not\ p_1.$	$r_2 \leftarrow p_2.$
$q_0 \leftarrow b.$		
$p_3 \leftarrow q_3.$	$p_4 \leftarrow q_4.$	$p_5 \leftarrow q_5.$
$q_3 \leftarrow not\ r_3.$	$q_4 \leftarrow p_4.$	$q_5 \leftarrow not\ r_5.$
$r_3 \leftarrow p_3.$	$q_4 \leftarrow not\ a, not\ b.$	$r_5 \leftarrow not\ s_5.$
		$s_5 \leftarrow p_5.$
$p_6 \leftarrow not\ q_6.$	$p_7 \leftarrow not\ q_7, r_7, a.$	$p_8 \leftarrow not\ q_8, a.$
$q_6 \leftarrow r_6.$	$r_7 \leftarrow not\ q_7, p_7, b.$	$q_8 \leftarrow not\ p_8.$
$r_6 \leftarrow s_6.$	$q_7 \leftarrow not\ p_7, not\ r_7.$	$q_8 \leftarrow b.$
$s_6 \leftarrow not\ p_6.$		
$p_{10} \leftarrow not\ q_{10}, a.$	$p_{11} \leftarrow not\ q_{11}, a.$	$p_{12} \leftarrow a, not\ q_{12}.$
$q_{10} \leftarrow p_{10}, a.$	$q_{11} \leftarrow p_{11}, not\ a.$	$q_{12} \leftarrow not\ a, p_{12}.$

Figure 6: Collection of Ground Programs with Loops

B Test-suite

The test-suite consists of two collections of programs: ground programs with loops, and programs with variables (also containing loops). In the evaluation of ground programs with loops, a comparison with the ABDUAL meta-interpreter [5] is made. Both systems run on the same platform under XSB version 3.3.7.

B.1 Programs with loops

Collection of programs Figure 6 lists a collection of programs, including difficult cases, used to compare TABDUAL and the ABDUAL meta-interpreter. The collection is specific to ground programs, since ABDUAL caters only to ground programs and queries. The evaluation results are shown subsequently. These programs involve various loops: direct positive loops, negative loops over negation, positive loops in (dualized) negation, and some combinations amongst them. In this collection, a, b, and c are abducibles.

Evaluation results Table 2 compares the results returned by TABDUAL and the ABDUAL meta-interpreter for queries to the ground programs in Figure 6.

Table 2: Comparison of results: TABDUAL vs. ABDUAL meta-interpreter

Queries	TABDUAL	ABDUAL meta-interpreter
p_0	$[a], [b]$	$[a], [b]$
$not\ p_0$	$[not\ a, not\ b]$	$[not\ a, not\ b], [not\ a]$
$not\ p_1$	$[\,]$	$[\,]$
q_1	$[\,]$	$[\,]$
p_2	no	no
$not\ p_2$	$[\,]$	$[\,]$
p_3	$[\,]$ *undefined*	$[\,]$
$not\ p_3$	$[\,]$ *undefined*	$[\,]$
p_4	$[not\ a, not\ b]$	$[not\ a, not\ b]$
$not\ p_4$	$[a], [b]$	$[a], [b], [a, b]$
p_5	$[\,]$ *undefined*	no
$not\ p_5$	$[\,]$ *undefined*	$[\,]$
p_6	$[\,]$ *undefined*	$[\,]$
$not\ p_6$	$[\,]$ *undefined*	no
p_7	no	no
$not\ p_7$	$[\,], [not\ a], [not\ b], [not\ a, not\ b]$	$[\,], [not\ a], [not\ b], [not\ a, not\ b]$
q_8	$[\,], [not\ a], [b]$	$[not\ a], [b]$
$not\ p_8$	$[\,], [not\ a], [b]$	$[not\ a], [b]$
p_{10}	$[a]$ *undefined*	$[a]$
$not\ p_{10}$	$[not\ a], [a]$ *undefined*	$[a], [not\ a]$
p_{11}	$[a]$	$[a]$
$not\ p_{11}$	$[not\ a]$	$[not\ a]$
$not\ q_{11}$	$[a], [not\ a]$	$[\,], [a], [not\ a]$
p_{12}	$[a]$	$[a]$
$not\ p_{12}$	$[not\ a]$	$[not\ a]$
$not\ q_{12}$	$[a], [not\ a]$	$[\,], [a], [not\ a]$

B.2 Programs with variables

Collection of programs Figure 7 lists programs with variables; many of them contain loops as well. In this collection, $a/1$, $b/1$, and $c/1$ are abducibles.

Evaluation results Table 3 presents the evaluation results returned by TABDUAL and the ABDUAL meta-interpreter for queries to programs in Figure 7.

Received \jreceived

$p_0(X) \leftarrow q_0(X).$
$p_0(1) \leftarrow a(1).$
$q_0(X) \leftarrow p_0(X).$
$q_0(2) \leftarrow a(2).$

$p_1(X) \leftarrow \mathit{not}\ q_1(X), r_1(X).$
$r_1(X) \leftarrow \mathit{not}\ q_1(X), p_1(X).$
$q_1(X) \leftarrow s_1(X), \mathit{not}\ p_1(X).$
$s_1(1).$

$p_2(X) \leftarrow q_2(X).$
$q_2(X) \leftarrow r_2(X).$
$r_2(X) \leftarrow p_2(X).$

$p_3(1) \leftarrow q_3(1).$
$q_3(X) \leftarrow \mathit{not}\ r_3(X).$
$r_3(X) \leftarrow p_3(X).$

$p_4(X) \leftarrow q_4(X).$
$q_4(X) \leftarrow p_4(X).$
$q_4(1) \leftarrow \mathit{not}\ a(1), \mathit{not}\ a(2).$

$p_5(X) \leftarrow q_5(X).$
$q_5(X) \leftarrow t_5(X), \mathit{not}\ r_5(X).$
$r_5(X) \leftarrow t_5(X), \mathit{not}\ s_5(X).$
$s_5(X) \leftarrow p_5(X).$
$t_5(1).$

$p_6(X) \leftarrow t_6(X), \mathit{not}\ q_6(X).$
$q_6(X) \leftarrow r_6(X).$
$r_6(X) \leftarrow s_6(X).$
$s_6(X) \leftarrow t_6(X), \mathit{not}\ p_6(X).$
$t_6(1).$

$p_7(X) \leftarrow s_7(X), \mathit{not}\ q_7(X), r_7(X).$
$r_7(X) \leftarrow t_7(X), \mathit{not}\ q_7(X), p_7(X).$
$q_7(X) \leftarrow \mathit{not}\ p_7(X), \mathit{not}\ r_7(X).$
$s_7(1) \leftarrow a(1).$
$t_7(1) \leftarrow b(1).$

$p_8(X) \leftarrow s_8(X), \mathit{not}\ q_8(X).$
$q_8(X) \leftarrow \mathit{not}\ p_8(X).$
$q_8(2) \leftarrow a(2).$
$s_8(1) \leftarrow a(1).$
$s_8(2) \leftarrow a(2).$

$p_{10}(X) \leftarrow s_{10}(X), \mathit{not}\ q_{10}(X).$
$q_{10}(X) \leftarrow p_{10}(X), a(X).$
$s_{10}(1) \leftarrow a(1).$

$p_{11}(X) \leftarrow s_{11}(X), \mathit{not}\ q_{11}(X).$
$q_{11}(X) \leftarrow p_{11}(X), \mathit{not}\ a(X).$
$s_{11}(1) \leftarrow a(1).$

$p_{13}(X) \leftarrow r_{13}(X), \mathit{not}\ p_{13}(X).$
$p_{13}(1) \leftarrow a(1), b(1).$
$p_{13}(2) \leftarrow c(2).$
$r_{13}(1) \leftarrow a(1).$
$r_{13}(2) \leftarrow a(2).$

Figure 7: Collection of Programs with Variables

Received \jreceived

Table 3: Evaluation results of Programs with Variables

Queries	Results by TABDUAL[a]
$p_0(X)$	$[a(1)]$ for $X = 1$; $[a(2)]$ for $X = 2$
$q_0(X)$	$[a(1)]$ for $X = 1$; $[a(2)]$ for $X = 2$
$not\ p_0(X)$	$[not\ a(1), not\ a(2)]$ for $X = \_$
$not\ q_0(X)$	$[not\ a(1), not\ a(2)]$ for $X = \_$
$q_1(X)$	[ ] for $X = 1$
$not\ q_1(X)$	no
$not\ p_1(X)$	[ ] for $X = \_$
$p_2(X)$	no
$not\ p_2(X)$	[ ] for $X = \_$
$p_3(X)$	[ ] *undefined* for $X = 1$
$not\ p_3(X)$	[ ] *undefined* for $X = \_$
$p_4(X)$	$[not\ a(1), not\ a(2)]$ for $X = 1$
$not\ p_4(X)$	$[a(1)],\ [a(2)]$ for $X = \_$
$p_5(X)$	[ ] *undefined* for $X = 1$
$not\ p_5(X)$	[ ] *undefined* for $X = \_$
$p_6(X)$	[ ] *undefined* for $X = 1$
$not\ p_6(X)$	[ ] *undefined* for $X = \_$
$p_7(X)$	no
$not\ p_7(X)$	$[a(1)],\ [not\ a(1)],\ [a(1), b(1)],\ [a(1), not\ b(1)]$ for $X = \_$
$p_8(X)$	$[a(1)]$*undefined* for $X = 1$
$not\ p_8(X)$	$[a(1)],\ [a(2)],\ [a(1), a(2)],\ [not\ a(1), not\ a(2)]$ for $X = \_$
$p_{10}(X)$	$[a(1)]$*undefined* for $X = 1$
$not\ p_{10}(X)$	$[a(1)]$ *undefined*, $[not\ a(1)]$ for $X = \_$
$p_{11}(X)$	$[a(1)]$ *undefined* for $X = 1$
$not\ p_{11}(X)$	$[not\ a(1)]$ *undefined* for $X = \_$
$q_{13}(X)$	$[a(1), not\ b(1)]$ for $X = 1$; $[a(2), not\ c(2)]$ for $X = 2$
$not\ q_{13}(X)$	$[a(1), b(1)],\ [a(2), c(2)],\ [not\ a(1), not\ a(2)]$ for $X = \_$
$not\ p_{13}(X)$	$[not\ b(1), not\ c(2)],\ [not\ a(1), not\ c(2)]$ for $X = \_$

[a]Underscore (_) denotes some variable, for instance in $X = \_$ (i.e. X is left uninstantiated).

Received \jreceived