Tabled Abduction in Logic Programs

Ari Saptawijaya* and Luís Moniz Pereira

Centro de Inteligência Artificial (CENTRIA), Departamento de Informática
Faculdade de Ciências e Tecnologia, Univ. Nova de Lisboa, 2829-516 Caparica, 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: (1) We conceptualize tabled abduction for abductive normal logic programs, permitting abductive solutions to be reused, from one abductive context to another. The approach relies on a transformation into tabled logic programs that makes use of the dual transformation, and enables efficiently handling the problem of abduction under negative goals, by introducing dual positive counterparts for them. (2) We realize tabled abduction in TABDUAL, a system implemented in XSB Prolog, allowing dualization by-need only. (3) We refine the dual transformation in the context of TABDUAL to permit executing programs with variables and non-ground queries. (4) We foster pragmatic approaches in TABDUAL to cater to all varieties of loops in normal logic programs, now complicated by abduction. (5) We evaluate TABDUAL in practice by examining five variants, according to various evaluation objectives. (6) We detail how TABDUAL can be applied to declarative debugging and decision making. (7) Finally, we refer to related work, and discuss TABDUAL's correctness, complexity, and features that could migrate to the engine level, in Logic Programming systems wanting to encompass tabled abduction.

Keywords: abductive logic programming, tabled abduction, dual transformation, XSB Prolog, applications of abduction.

1 Introduction

Abduction has been well studied in the field of computational logic, and logic programming in particular, for a few decades by now [4, 9, 12, 14, 18, 20, 39]. Abduction in logic programs 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, 5, 13, 15, 21–23, 28, 32]. On the other hand, many Prolog systems have become mature and practical, and thus it makes sense to facilitate the use

* Affiliated with Fakultas Ilmu Komputer at Universitas Indonesia, Depok, Indonesia.

of abduction into such systems, be it two-valued abduction (as adopted in this work) or three-valued, e.g. [7].

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 logic programming, absent of abduction, goal solution reuse is commonly addressed by employing a tabling mechanism. 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 [31], 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. Tabling abductive solutions (tabled abduction) with its prototype TABDUAL, implemented in XSB Prolog [41], consists of a program transformation from abductive normal logic programs into tabled logic programs, plus a library of system predicates. It requires no meta-interpreter, but generates a self-sufficient program transform, on which abduction is subsequently enacted. TABDUAL is available at `http://sourceforge.net/projects/tabdual`.

We simplify the specification of tabled abduction in TABDUAL by introducing the core transformation, in Section 3, which abstracts away from implementation details of its subsequent refinements and more complex constructs, such as loops (i.e. positive loops and loops over negation) in abductive normal logic programs. That is, the core TABDUAL transformation focuses on an innovative re-uptake of prior abductive solution entries in tabled predicates, and the dual transformation [4], on which TABDUAL relies. The dual transformation, initially employed in ABDUAL [4], allows to more efficiently handle the problem of abduction under negative goals, by introducing their positive dual counterparts.

Originally, the dual transformation in ABDUAL does not concern itself with programs having variables. In Section 4, the dual transformation is further refined in the context of TABDUAL, to allow it dealing with such programs. More precisely, it is refined to help ground (dualized) negative subgoals, and to deal with non-ground negative goals. Regarding the latter, 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.

The tabling mechanism in XSB Prolog [41] supports the Well-Founded Semantics (WFS) [43], which allows dealing with loops in the program and ensuring termination of looping queries. TABDUAL is implemented in XSB, and it employs XSB's tabling as much as possible to deal with loops. Nevertheless, tabled abduction introduces a complication concerning some varieties of loops. In Section 5, we adapt the core TABDUAL transformation, resorting to a pragmatic approach, to cater to all varieties of loops in normal logic programs, which are now complicated by abduction.

Keeping in mind TABDUAL as a practical tabled abduction system, under the WFS with abduction [4], several pragmatic aspects have been examined from the implementation viewpoint [34]. First, because TABDUAL allows for modular mixes between abductive and non-abductive program parts, one can benefit in the latter part by enacting

a simpler translation of predicates in the program 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. Second, we address the issue of potentially heavy transformation load due to producing the *complete* dual rules (i.e. all dual rules regardless of their need), if these are constructed in advance by the transformation. 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: creating and tabling all dual rules for a predicate on the first invocation of its negation, or, in contrast, lazily generating and storing (instead of tabling) its dual rules in a trie, as new alternatives are required. The former approach leads to an eager dual rules tabling (albeit by-need) transformation (under local table scheduling strategy), whereas the latter permits a by-need driven lazy one (in lieu of batched table scheduling). Third, 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. These implementation aspects and others are examined in Section 6.

TABDUAL has been evaluated with various objectives, where five TABDUAL variants (of the same underlying implementation) are examined, by separately factoring out TABDUAL's most important distinguishing features [35]. In the first, we evaluate the benefit of tabling abductive solutions, where we employ an example from declarative debugging to debug incorrect solutions of logic programs, via a process now characterized as abduction [36], instead of as belief revision [26,27]. The other case of declarative debugging, that of debugging missing solutions, is used next to evaluate the three dual transformation variants: complete, eager by-need, and lazy by-need. We touch upon tabling so-called *nogoods* of subproblems in the context of abduction (i.e. abductive solution candidates that violate constraints), and show, in the third evaluation, that tabling abductive solutions can be appropriate for tabling nogoods of subproblems. We also evaluate TABDUAL in dealing with programs having loops, in the fourth evaluation, where we also compare its results with ABDUAL, showing that TABDUAL provides more correct and complete results. Finally, we describe how TABDUAL can be applied in action decision making under hypothetical reasoning, and in a real medical diagnosis case [36]. The evaluations and the applications of TABDUAL are detailed in Section 7 and Section 8, respectively.

We discuss the correctness and the complexity of TABDUAL, in Section 9, as well as related work and other issues that may lead to future work. In particular, we discuss which features of the TABDUAL transformation, currently deployed at the object language level, could possibly migrate to the engine level of Prolog systems that support tabling and, optionally, tries data structure, like XSB and others wanting to encompass tabled abduction.

2 Preliminaries

We start by reviewing some background definitions in logic programs and notation we use throughout this work.

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. In a rule, H is called the head of the rule and $B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$ its body. We use 'not' to denote default negation. The atom B_i and its default negation $not\ B_i$ are named positive and negative *literals*, respectively. When $n = 0$, we say the rule is a *fact* and render it simply 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* (LP) 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, we write p/n to denote predicate p with arity n.

Abduction (inference to the best explanation – a common designation in the philosophy of science [19,24]), 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 LPs, 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 have no rules, but instead to the explicitly assumed hypothetical negation of a. The truth value of abucibles 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 LPs 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) [43] enjoys the relevance property, and thus it allows abduction to be performed by need, 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, assuming the ICs are satisfied. Our prototype of tabled abduction TABDUAL, is based on the WFS with abduction [4], and implemented in XSB Prolog. 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 a program transformation of abductive normal logic programs into tabled 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 The Core Transformation

We now discuss the core TABDUAL transformation, through a sequence of examples, employing the idea of tabling and of reusing abductive solutions, and the dual transformation to deal with abduction under negative goals. The core transformation results in a tabled definite logic program, thanks to the dual transformation. A formal specification of the core transformation is detailed in Appendix A.

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. It is transformed into two rules:

$t_{ab}(E_2) \leftarrow s([\,], E_1), q(E_1, E_2).$ $\qquad$ $t(I, O) \leftarrow t_{ab}(E), produce(O, I, E).$

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 of t_{ab} is obtained by relaying the ongoing abductive solution stored in context T 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 rules 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 $produce(O, I, E)$. 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, $produce/3$ fails and another entry E is sought. In other words, $produce/3$ 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:

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

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:

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

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, which contains rules of non-nullary predicate $q/1$ (also with variables) with $a/1$ abducible:

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

The transformation results in rules as follows:

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

$q(X, I, O) \leftarrow q_{ab}(X, E), produce(O, I, E).$

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.

3.2.2 Abduction under Negative Goals

For abducing under negative goals, the program transformation employs the *dual transformation* [4], which 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 4 illustrates the main idea of how the dual transformation is employed in the TABDUAL transformation.

Example 4. 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:

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

  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 4 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 5.

Example 5. 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 $q/1$.
$$\begin{array}{ll} 1.\ not\_q(X, T_0, T_2) & \leftarrow q^{*1}(X, T_0, T_1), q^{*2}(X, T_1, T_2). \\ 2.\ q^{*1}(X, I, I) & \leftarrow X \setminus= 0. \\ 3.\ q^{*2}(X, I, I) & \leftarrow X \setminus= s(\_). \\ 4.\ q^{*2}(s(X), I, O) & \leftarrow not\_a(X, I, O). \end{array}$$
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 how the second layer of the dual rules can be refined 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).

3.2.3 Transforming Abducibles

In Example 4, $p^{*1}(I, O)$ is defined by abducing $not\ a$, achieved by invoking subgoal $not\_a(I, O)$. Abduction in TABDUAL is realized by transforming each abducible atom (and its negation) into a rule, which updates the abductive context with the abducible atom (or its negation, respectively). Say, abducible a of Example 4 translates to:
$$a(I, O) \leftarrow insert(a, I, O).$$
where $insert(A, I, O)$ 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:
$$not\_a(I, O) \leftarrow insert(not\ a, I, O).$$

3.2.4 Transforming Queries

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

- 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

$$not\_\mathit{false}(I, I).$$

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 $\mathit{false}(E)$ and $\mathit{false}(I, O)$ may be omitted. Finally, a query should always be conjoined with $not\_\mathit{false}/2$ to ensure that all integrity constraints are satisfied.

Example 6. Query

$$?-\ \ not\ p.$$

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

$$?-\ \ not\_p([\,], T),\ not\_\mathit{false}(T, O).$$

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.

4 Refining the Dual Transformation

Next, we refine the dual transformation in TABDUAL to touch better upon abduction in programs with variables. The refinement allows further grounding of the dualized negative subgoals in the dual transformation, and to deal correctly with non-ground negative goals.

4.1 Grounding Negated Subgoals

Example 7. Consider program P_6, with $a/1$ abducible:

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

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

1. $q_{ab}(1, [\,])$.
2. $q(X, I, O) \leftarrow q_{ab}(X, E), produce(O, I, E)$.
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. $r(X, I, O) \leftarrow r_{ab}(X, E), produce(O, I, E)$.
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. $\mathit{not\_false}(I, O) \leftarrow \mathit{false}^{*1}(I, O)$.
11. $\mathit{false}^{*1}(I, O) \leftarrow not\_q(X, I, O)$.
12. $\mathit{false}^{*1}(I, O) \leftarrow not\_r(X, I, O)$.

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

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

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. 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.

This brings us to the first refinement of the dual transformation: 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. Rule 12 can thus be refined as follows (all other rules remain the same):

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

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. Moreover, this refinement also allows us to deal with non-ground positive goals. For instance, 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 refinement. 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 refinement 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 help refine the transformation further, to avoid introducing positive literals that are not contributing to further grounding.

4.2 On Non-ground Negative Goals

Example 8. 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 $not\_p/3$:

$$\begin{array}{ll}
1.\ not_p(X,I,O) & \leftarrow p^{*1}(X,I,T), p^{*2}(X,T,O). \\
2.\ p^{*1}(X,I,I) & \leftarrow X \setminus= 1. \\
3.\ p^{*1}(1,I,O) & \leftarrow not_a(1,I,O). \\
4.\ p^{*2}(X,I,I) & \leftarrow X \setminus= 2. \\
5.\ p^{*2}(2,I,O) & \leftarrow not_a(2,I,O).
\end{array}$$

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

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

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 3.2.4), thus returning the abductive solution $[not\ a(1)]$ with $X = 1$ for the given query.

The culprit 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 $p^{*2}/3$, which leads us to the second refinement of the dual transformation, as shown for rule $not\_p/3$ (line 1) below:

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

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. $[X]$.

With this refinement, $p^{*1}/3$ and $p^{*2}/3$ are invoked using variant independent calling arguments: X_1 and X_2, respectively. Now, the same query first invokes $p^{*1}(X_1, [\,], T_1)$, which results in $X_1 = 1$ and $T_1 = [not\ a(1)]$ (by the second rule of p^{*1}), and subsequently invokes $p^{*2}(X_2, [not\ a(1)], T_2)$, resulting in $X_2 = 2$ and $T_2 = [not\ a(1), not\ a(2)]$ (by the second rule of p^{*2}). It eventually ends up with the expected abductive solution: $[not\ a(1), not\ a(2)]$ for any instantiation of X, i.e. X remains unbound. Indeed, the refinement ensures, as this example shows, that $p(X)$ fails for every X, and its negation, $not\ p(X)$, 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 $not\ p(X)$ would not succeed because now we have the first layer dual rule:

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

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

5 Programs with Loops in TABDUAL

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, XSB's tabling mechanism is employed as much as possible to deal with loops. Nevertheless, the presence of tabled abduction requires some varieties of loops to be handled carefully in the transformation, as we detail here. Additional examples, besides the ones below, are to be found in Appendix C.

5.1 Direct Positive Loops

Example 9. 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}:

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

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

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.

5.2 Positive Loops in (Dualized) Negation

Indeed, 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.

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, this ancestor list implements the co-unfounded set of literals, defined in Definition 3.5 of [4], in order to deal with PLoN.

The refined 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):

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

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).

With this refinement, query $not\ p$ is now transformed into:

$$?-\ \ not\_p([\,], T, [\,]), not\_\textit{false}(T, O).$$

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

5.3 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 10. 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:

$$\begin{array}{ll} p(I,O) & \leftarrow p_{ab}(E), produce(O,I,E). \\ p_{ab}(E) & \leftarrow q([\,], E). \\ q(I,O) & \leftarrow q_{ab}(E), produce(O,I,E). \\ q_{ab}(E) & \leftarrow not\_p([\,], E). \end{array}$$

$$\begin{array}{ll} not\_p(I,O) & \leftarrow p^{*1}(I,O). \\ p^{*1}(I,O) & \leftarrow not\_q(I,O). \\ not\_q(I,O) & \leftarrow q^{*1}(I,O). \\ q^{*1}(I,O) & \leftarrow p(I,O). \end{array}$$

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). The definition of q_{ab} is thus refined as follows (other transformed rules remain the same):

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

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 Implementation Aspects of TABDUAL

Thus far, we have conceptualized tabled abduction using a program transformation, and refined it to touch better upon programs with variables and with varieties of loops. We discuss here several aspects pertaining to the implementation of the TABDUAL transformation, which are introduced 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. The input program of TABDUAL, as shown in Example 11, 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 $prolog/1$. It executes Prolog calls to goals in its argument. These goals are not transformed and may be defined in the non-abductive part, or alternatively, defined by Prolog's built-in predicates.

Example 11. An example of input programs of TABDUAL:

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

$$\begin{array}{l} beginProlog. \\ \quad nat(0). \\ \quad nat(s(X)) \leftarrow nat(X). \\ endProlog. \end{array}$$

6.2 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 7). 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:

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

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 just the whole TABDUAL transformation time and space can be reduced, but also the abduction time itself.

6.3 Dual Transformation by-Need

TABDUAL conceptually performs a complete dual transformation, i.e. it produces *all* (first and second layer) dual rules, in advance and as an integral part of the transformation, for every defined atom in an input program. This 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 a complete 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 created during abduction, 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 12. Recall Example 4. 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, which is introduced to facilitate the dual transformation by-need:

- 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,
- It instantiates the generic dual rules with the provided arguments and input context, and
- It subsequently invokes the instantiated dual rules.

Constructing dual rules on-the-fly clearly introduces some extra cost during abduction. 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. They influence how generic dual rules are constructed and provide distinct definitions of the system predicate $dual/4$.

6.3.1 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 4, 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 [41].

Alternatively, batched scheduling is also implemented in XSB. It 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 construct only one generic dual rule 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.

6.3.2 Storing Generic Dual Rules in a Trie

XSB offers a mechanism for facts to be directly stored and manipulated in tries. It 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 in a trie and later (a copy) retrieved and reused. 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. $dual(N, P, I, O) \leftarrow trie\_property(T, alias(dual)), dual(T, N, P, I, O).$

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

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 the help of additional tracking information to 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.

6.4 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.

We overcome this issue by introducing the TABDUAL system predicate $abdQ(P)$ 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 13. Consider a fragment of an input program:

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

Notice that, predicate s wrapped by $abdQ/1$ 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 predicate $abdQ/1$ 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:

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

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

6.5 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 sytems [42]. 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 $insert/3$ and $produce/3$. 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 succesfully 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 4.1). 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 5.3, 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 objectives. As tabling abductive solutions is the main feature of TABDUAL, our first evaluation aims at evaluating its benefit, by employing TABDUAL in an example of declarative debugging. Second, we study the relative worth of the dual transformation by-need, both eagerly and lazily, in comparison with the one without it. Third, we touch upon the evaluation of tabling nogoods of subproblems in abduction. Fourth, we evaluate TABDUAL in dealing with all varieties of loops discussed in Section 5. Finally, we show how TABDUAL can be exploited in decision making and systems biology.

For the purpose of the evaluation, we consider five distinct TABDUAL variants (of the same underlying implementation), as shown in Table 1; they are characterized by the features of evaluation interest. Notice that TABDUAL+lazy-tab is the sole variant that does not exercise tabled abduction. It is accomplished by disabling, in the transformation, the table declarations of abductive predicates p^{ab}, for every predicate p.

Next, we detail and discuss the result of each evaluation. The experiments 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.

Table 1. Five TABDUAL Variants.

Variants	Tabling Abd. Solutions	Dual Transformation[a]	Loops Handling[b]
TABDUAL-need	✓	complete	✗
TABDUAL+eager	✓	by-need (eager)	✗
TABDUAL+lazy	✓	by-need (lazy)	✗
TABDUAL+lazy-tab	✗	by-need (lazy)	✗
TABDUAL$^{\infty}$+lazy	✓	by-need (lazy)	✓

[a] All variants implement the refinements discussed in Section 4.
[b] It concerns all varieties of loops discussed in Section 5.

7.1 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 an example from declarative debugging (cf. Section 8.1) as the benchmark for this evaluation. It takes the following program to debug, where the size $n > 1$ of the program can easily be customized:

$$\begin{array}{ll} q_0(0,1). & q_0(X,0). \\ q_1(1). & q_1(X) \leftarrow q_0(X,X). \\ q_n(n). & q_n(X) \leftarrow q_{n-1}(X). \end{array}$$

In order to evaluate tabling abductive solutions, we consider the case of debugging missing solutions. More precisely, TABDUAL is employed to debug the program for its missing solutions, i.e. missing $q_m(1001)$, for various values of $100 \leq m \leq 1000$, which is expressed by adding ICs $\leftarrow not\ q_m(1001)$ to the program. For this evaluation, we focus on two variants: TABDUAL+lazy and TABDUAL+lazy-tab, i.e. comparing two variants, with and without tabling abductive solutions.

Figure 1 shows the abduction time of both TABDUAL variants, where the size of the program $n = 1000$, and different values of m in the IC: $\leftarrow not\ q_m(1001)$, $100 \leq m \leq 1000$, are evaluated consecutively. The result reveals that, with some little cost of tabling abductive solutions in earlier values of m (i.e. $m \leq 300$), TABDUAL+lazy consistently outperforms TABDUAL+lazy-tab in performance. Tabling pays off for subsequent values of m in TABDUAL+lazy, as greater m may reuse tabled abductive solutions of smaller m, due to the consecutive evaluation of ICs. Moreover, TABDUAL+lazy scales better than TABDUAL+lazy-tab, i.e. as the values of m grows, its abduction time increases slower than its counterpart. We may observe, that its abduction time tends to grow linearly, whereas the latter variant exponentially.

7.2 Evaluation of The Dual Transformation Variants

For this evaluation, we resort to the same example of declarative debugging used in the evaluation of tabling abductive solutions (cf. Section 7.1):

$$\begin{array}{ll} q_0(0,1). & q_0(X,0). \\ q_1(1). & q_1(X) \leftarrow q_0(X,X). \\ q_n(n). & q_n(X) \leftarrow q_{n-1}(X). \end{array}$$

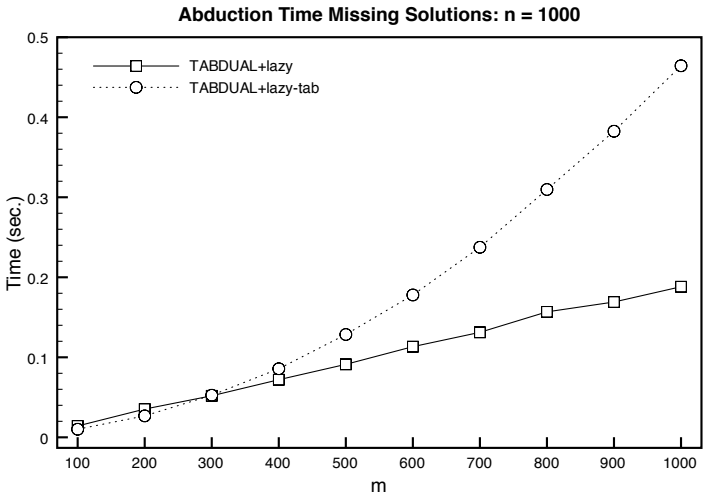

Fig. 1. The abduction time for debugging missing solutions $q_m(1001)$, for $100 \leq m \leq 1000$, and with the program size $n = 1000$.

But instead of debugging the program for missing solutions, we consider the case of incorrect solutions. That is, we look for the causes of incorrect $q_m(0)$, for various values of $100 \leq m \leq 1000$, expressed by adding ICs $\leftarrow q_m(0)$ to the program. Since our aim is particularly to evaluate the relative worth of the dual transformation by-need, we focus on three variants: TABDUAL-need, TABDUAL+eager, and TABDUAL+lazy. We evaluate the benchmark for the program size $n = 1000$, i.e. debugging a program with 2002 rules. After applying the declarative debugging transformation (of incorrect solutions, cf. Section 8.1), which results in an abductive logic program, we apply the TABDUAL transformation. It takes 1.1636 seconds for the TABDUAL variants that employ the dual transformation by-need (either eagerly or lazily), whereas TABDUAL-need takes 1.6737 seconds. TABDUAL+eager and TABDUAL+lazy clearly require less transformation time than TABDUAL-need, since they do not produce complete dual rules in advance as the latter variant does. Take an example q_2, whose rules in the abductive program (as the result of the declarative debugging transformation, cf. Section 8.1) are as follows:

$$q_2(2) \leftarrow not\ incorrect(5,[2]).$$
$$q_2(X) \leftarrow not\ incorrect(6,[X]),\ q_1(X).$$

The second layer dual rules produced, in advance, by TABDUAL-need are as follows (apart from those dual rules that are defined by disunifying arguments):

$$q_2^{*1}(2,I,O) \leftarrow incorrect(5,[2],I,O).$$
$$q_2^{*2}(X,I,O) \leftarrow incorrect(6,[X],I,O).$$
$$q_2^{*2}(X,I,O) \leftarrow not\_q_1(X,I,O).$$

whereas TABDUAL+eager and TABDUAL+lazy only produce their skeleton, which engenders the dual transformation by-need during abduction (cf. Example 12):

$$q_2^{*1}(2, I, O) \leftarrow dual(1, q_2(2), I, O).$$
$$q_2^{*2}(X, I, O) \leftarrow dual(2, q_2(X), I, O).$$

That is, apart from those dual rules which are defined by disunifying arguments, TABDUAL-need creates 3002 second layer dual rules during the transformation, whereas TABDUAL+eager and TABDUAL+lazy creates only 2002 second layer dual rules. And during abduction, the latter two variants construct only, by need, 60% of the complete second layer dual rules produced by the other variant: with respect to the ICs $\leftarrow q_m(0)$, for some m, there is no need to consider $q_2^{*1}(2, I, O) \leftarrow incorrect(5, [2], I, O)$, since it fails and $q_2^{*1}/3$ indeed succeeds by the other dual rule which disunifies arguments, i.e. $X \neq 2$ (where X is instantiated by 0, due to the ICs).

Figure 2 shows how the dual transformation by-need influences the abduction time, where different values of m in the IC: $\leftarrow q_m(0)$ are evaluated consecutively, $100 \leq m \leq 1000$; in this way, greater m may reuse generic dual rules constructed earlier by smaller m. We may observe that TABDUAL-need is faster than the two variants with the dual transformation by-need. This is expected, due to the overhead incurred for computing dual rules on-the-fly, by need, during abduction. On the other hand, the overhead is compensated for by the significantly less transformation time: the total (transformation plus abduction) time of TABDUAL-need is 1.9289 seconds, whereas TABDUAL+eager and TABDUAL+lazy are 1.5207 and 1.6203 seconds, respectively.

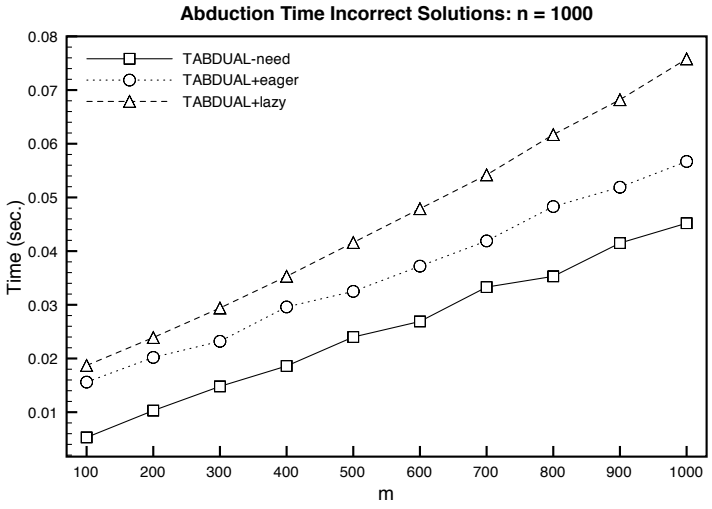

Fig. 2. The abduction time for debugging incorrect solutions $q_m(0)$, for $100 \leq m \leq 1000$, and with the program size $n = 1000$.

In this scenario, where all abductive solutions are aggregated, TABDUAL+lazy is slower than TABDUAL+eager; the culprit could be the extra maintenance of the tracking information needed for the explicit memoization. It may as well explain that, as its consequence, the time gap between TABDUAL+lazy and TABDUAL+eager is wider as m grows, i.e. more dual rules are stored in the trie. Nevertheless, TABDUAL+lazy returns the first abductive solution much faster than TABDUAL+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 with the eager one. Nevertheless, the two approaches may become options for TABDUAL customization.

7.3 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 diverse fields, such as truth maintenance systems [8, 11], constraint satisfaction problems [40], SAT solvers [25], and recently in answer set solvers [16], 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:

$$q(0, N).$$
$$q(M, N) \leftarrow M > 0,\ q(M-1, N),\ d(Y),\ pos(M, Y),\ abdQ(not\ conflict).$$

$$conflict(BoardConf) \leftarrow prolog(conflictual(BoardConf)).$$

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 $conflictual/1$ is defined in a non-abductive program module, to check whether the ongoing board configuration $BoardConf$ of queens is conflictual. 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 conflictual 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 focus only on two TABDUAL variants: one with tabling feature, represented by TABDUAL+lazy, and the other without it, i.e. TABDUAL+lazy-tab. The transformation time of the problem representation is similar for both variants, i.e. around 0.003 seconds. Figure 3 shows abduction time for N queens, $4 \leq N \leq 11$. The reason that TABDUAL+lazy performs worse than TABDUAL+lazy-tab 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 complexity of the constraints, e.g. adding more queen's attributes (colors, shapes, etc.) to further constrain its safe positioning?

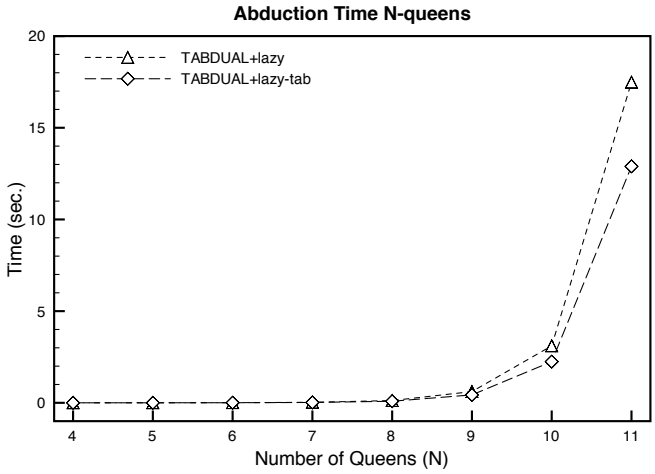

Fig. 3. The abduction time of different N queens.

Figure 4 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+lazy's performance is remedied and, benefitting from tabling the ongoing conflict configurations, it consistently surpasses the performance of TABDUAL+lazy-tab (with 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.

7.4 Evaluation of Programs with Loops

We also evaluate TABDUAL, in this case its variant TABDUAL$^{\infty}$+lazy with loops handling, to assess the effectiveness of our approach on dealing with programs having loops in the presence of tabled abduction, as detailed in Section 5. 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 ABDUAL [4]. We provide a comparison of the results returned by both systems, focusing particularly on those that differ.

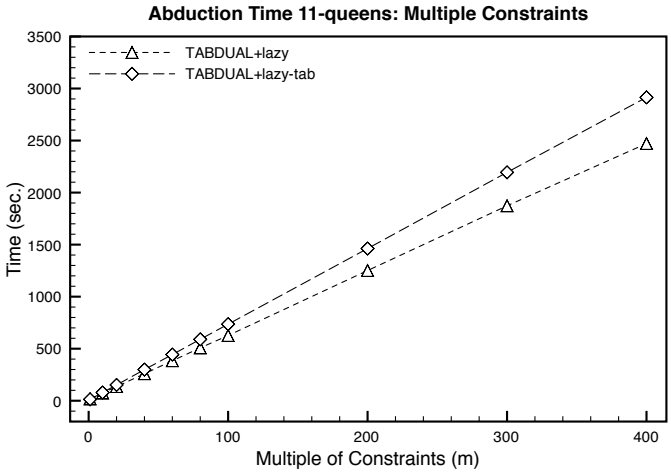

Fig. 4. The abduction time of 11 queens with increasing complexity of conflict constraints.

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

$p_0 \leftarrow q_0.$ $\quad p_0 \leftarrow a.$ $\quad q_0 \leftarrow p_0.$ $\quad q_0 \leftarrow b.$

$p_3 \leftarrow q_3.$ $\quad q_3 \leftarrow not\ r_3.$ $\quad r_3 \leftarrow p_3.$

$p_4 \leftarrow q_4.$ $\quad q_4 \leftarrow p_4.$ $\quad q_4 \leftarrow not\ a,\ not\ b.$

$p_8 \leftarrow not\ q_8,\ a.$ $\quad q_8 \leftarrow not\ p_8.$ $\quad q_8 \leftarrow b.$

$p_{11} \leftarrow not\ q_{11},\ a.$ $\quad q_{11} \leftarrow p_{11},\ not\ a.$

where a and b are abducibles. Table 2 lists the answers of the given queries to the corresponding programs, returned by TABDUAL and ABDUAL.

Table 2. Comparison of results for programs with loops: TABDUAL vs. ABDUAL.

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

TABDUAL provides more correct and complete results with respect to these programs, as detailed below:

– For query $not\ p_0$, $[not\ a, not\ b]$ should be the only solution, because $not\ p_0$ succeeds 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 ABDUAL, 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 ABDUAL, by abducing a or b only; 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 ABDUAL.
– 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 $[\,]$ abductive solution, as returned by ABDUAL.

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 C.

8 Applications of TABDUAL

We explore some applications of abduction, where TABDUAL can particularly be applied. We revisit declarative debugging and show that it can be viewed as abduction; an example of which is used in TABDUAL's evaluations (cf. Appendix 7.1 and 7.2). We also look into the application of TABDUAL in decision making under hypothetical reasoning, and into a medical case.

8.1 Declarative Debugging as Abduction

Declarative debugging of normal logic programs has been characterized before as belief revision [26, 27]. We recall the two cases of declarative debugging considered there, those of incorrect solutions and of missing solutions, and show that they can be viewed and implemented as abduction. We start by considering these two cases for definite logic programs, and next for normal logic programs.

8.1.1 Debugging of Definite Logic Programs

Example 14. Take a buggy program P_{10} [26]:

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

Incorrect Solutions Suppose that $a(3)$ is an incorrect solution. To debug its cause, the program is first changed using the simple transformation introduced in [27], i.e. 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}:

$a(1) \leftarrow not\ incorrect(1, [1]).$ $\quad$ $a(X) \leftarrow b(X), c(Y, Y), not\ incorrect(2, [X]).$
$b(2) \leftarrow not\ incorrect(3, [2]).$ $\quad$ $b(3) \leftarrow not\ incorrect(4, [3]).$
$c(1, X) \leftarrow not\ incorrect(5, [1, X]).$ $\quad$ $c(2, 2) \leftarrow not\ incorrect(6, [2, 2]).$

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: $\leftarrow a(3)$. We run TABDUAL on P'_{10}, which returns three solutions as the possible sufficient causes of the incorrect solution:

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

Missing Solutions Suppose $a(5)$ should be a solution of P_{10}, but is missing. To find this bug, P_{10} is transformed [26] 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_4 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 [26, 27], 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.

8.1.2 Debugging of Normal Logic Programs

Finally, in case of debugging normal logic programs, the two above debugging transformations are adroitly summed into one, as illustrated in Example 15. TABDUAL takes care of further transforming the default not's into positive atoms by dualizing them, to make the program a definite one.

Example 15. Consider program P_{11}:

$$a \leftarrow not\ b. \qquad a \leftarrow c. \qquad b.$$

We obtain P_{11}^* by applying the two transformations:

$a \leftarrow not\ b, not\ incorrect(1).$ $\quad a \leftarrow c, not\ incorrect(2).$ $\quad b \leftarrow not\ incorrect(3).$
$a \leftarrow missing(a).$ $\quad b \leftarrow missing(b).$ $\quad c \leftarrow missing(c).$

Suppose we want to explain the causes of missing solution a, then we add to P_{11}^* an IC: $\leftarrow not\ a$. Running TABDUAL on P_{11}^*, we obtain three abductive solutions:

$$[incorrect(3)],\ [missing(a)],\ [missing(c)]$$

which correctly enunciate the three possible causes of the problem.

This abductive approach to declarative debugging, by means of program dualization, avoids the conceptual and practical complications of the belief revision procedure specified and enacted in [26, 27].

8.2 Decision Making under Hypothetical Reasoning

In decision making under hypothetical reasoning, given an observation, one is typically confronted with several possible scenarios. These scenarios are characterized by the explanatory abducibles, and decisions are made on their basis. We illustrate with a basic example.

Example 16. Suppose an agent observes some smoke and its action decision with regard to this situation depends on the cause of the smoke. In case it is triggered by fire, the agent reacts by calling firefighters. But if it is explained by the presence of tear gas, then the agent better seeks police protection.

We extend TABDUAL with a system predicate to pick up actions based on available abductive solutions. That is, top-goal queries decision making can be enacted by TABDUAL system predicate $do(Act, Abds, Obs)$, defined as follows:

$$do(Act, Abds, Obs) \leftarrow abd(Obs, Abds),\ decide(Act, Abds).$$

where $abd(Obs, Abds)$ is another TABDUAL system predicate, which launches Obs as a query and returns $Abds$ as its abductive solution. Predicate $decide(Act, Abds)$ is defined in the agent's beliefs, and picks up an action Act based on the abductive solution $Abds$.

Example 16 can be modeled in TABDUAL as follows:

$smoke \leftarrow fire.$ $\qquad smoke \leftarrow tear\_gas.$
$beginProlog.$
$\quad decide(call\_firefighters, Abds) \leftarrow member(fire, Abds).$
$\quad decide(police\_protection, Abds) \leftarrow member(tear\_gas, Abds).$
$endProlog.$

where $fire$ and $tear\_gas$ are abducibles. Notice that $decide/2$ is defined in the non-abductive program part (between $beginProlog$ and $endProlog$ identifiers, cf. 6).

We may now launch a top-goal query $do(Act, Abds, smoke)$ to this program, which provides us with two scenarios and guides us with actions to take: $Act = call\_firefighters$ for $Abds = [fire]$ and $Act = police\_protection$ for $Abds = [tear\_gas]$.

8.3 Medical Diagnosis

Next, TABDUAL is applied to medical diagnosis, adapted from [28].

Example 17. A patient shows up at the dentist with signs of pain upon teeth percussion but without tooth mobility. The expected causes for the observed signs are periapical lesion, horizontal fracture, and vertical fracture of the root and/or crown.

An abductive logic program representing a partial medical knowledge base of the practitioner is as follows:

$$
\begin{array}{l}
percussion\_pain \leftarrow periapical\_lesion \\
percussion\_pain \leftarrow fracture \\
\\
radiolucency \leftarrow periapical\_lesion \\
\\
fracture \leftarrow horizontal\_fracture \\
elliptic\_fracture\_trace \leftarrow horizontal\_fracture \\
tooth\_mobility \leftarrow horizontal\_fracture \\
\\
fracture \leftarrow vertical\_fracture \\
decompression\_pain \leftarrow vertical\_fracture \\
\\
\leftarrow not\ percussion\_pain \\
\leftarrow tooth\_mobility
\end{array}
$$

where $periapical\_lesion$, $horizontal\_fracture$, and $vertical\_fracture$ are abducibles. The integrity constraints indicate that the practitioner must conclude $percussion\_pain$ but not $tooth\_mobility$ since these are the symptoms of the patient that requires explanation.

Suppose that during examination, the practitioner suspects that there is a fracture. This corresponds to query $fracture$ with its corresponding transformed top-goal:

$$fracture(I,T), not\_false(T,O).$$

Recall that integrity constraints are transformed like any other rules (cf. Section 3.2.4). In particular, predicate $not\_false/2$ is defined, by the dual transformation, as follows:

$$
\begin{array}{l}
not\_false(I,O) \leftarrow false^{*1}(I,T), false^{*2}(T,O). \\
false^{*1}(I,O) \leftarrow percussion\_pain(I,O). \\
false^{*2}(I,O) \leftarrow not\_tooth\_mobility(I,O).
\end{array}
$$

The first subgoal $fracture$ gives two abductive solutions: $T = [horizontal\_fracture]$ and $T = [vertical\_fracture]$. The second subgoal not_false, constrains these two solutions further:

- It eliminates $T = [horizontal\_fracture]$, due to rule $false^{*2}/2$. This rule, which eventually abduces $not\ horizontal\_fracture$, makes the abductive solution inconsistent.

– With respect to $T = [vertical\_fracture]$, the integrity constraint results in two final abductive solutions: $O = [periapical\_lesion, vertical\_fracture]$ and $O = [vertical\_fracture]$.

Notice that in obtaining $O = [vertical\_fracture]$, TABDUAL allows reusing the tabled abductive solution of the first subgoal $fracture$. That is, in the derivation of $false^{*1}$, $percussion\_pain$ is invoked by *false*^{*1}, which subsequently (re-)invokes $fracture$. This shows the benefit of tabled abduction in this problem.

9 Discussion

9.1 Correctness and Complexity

Correctness As the core TABDUAL transformation relies on the dual transformation, introduced in ABDUAL [4], its correctness stems from that of ABDUAL, shown formally there. It theoretically justifies, supports, and closely reflects the correctness of the core TABDUAL transformation. The details introduced in the core transformation (e.g. abductive contexts, the auxiliary predicates $produce/3$ and $insert/3$), and its subsequent refinements to deal with programs having loops, are just a concrete realization of the more abstract theory of ABDUAL. Its implementation aspects are extra complexities and refinements introduced for TABDUAL to achieve optimizations pertinent to the XSB features, like tabling and tries. Nevertheless, there are some points worthy of note:

– Whereas ABDUAL is restricted to ground programs and queries, TABDUAL caters to programs with variables and non-ground queries (cf. Example 5 and Section 4). Indeed, 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 core transformation. Recall Example 5, 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 5 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 $a(X')$:[1]

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

which is equivalent to rule 4 in Example 5, 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 core transformation) to rules with variables allows to avoid defining a specific dual transformation for particularly dealing with such

[1] Actually, the refinement of including all positive literals that precede the negated literal (cf. Section 4.1) is just another instance of requiring one failed literal in the body and allowing to assume other (preceding) positive literals to succeed.

rules. Inasmuch as head unifications of a rule are moved to equalities in its body, one can think of the ground instances of all the rules, and stick to the dual transformation of ABDUAL with its groundness assumption.

– The loops handling in TABDUAL is just implementing what ABDUAL specifies as unfounded and co-unfounded sets and their related operations. Additionally, TABDUAL benefits from XSB tabled negation treatment by employing a double $tnot$ wrapping to deal with negative loops over negation (NLoN), as discussed in Section 5.3.[2] In other words, the TABDUAL refinements to deal with programs having loops are theoretically underpinned by ABDUAL and readily implement loops handling; thus they do not require any particular proof of correctness, resorting as well to that of ABDUAL.

In Definition 3.1 of [4], a form of dual rules, i.e. folded dual form, is introduced. Its need is more theoretical, to show the correctness and the complexity results of ABDUAL. It also 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. 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 (cf. Definition 2 in A) is logically equivalent to the folded dual form, but it is simpler.

Complexity In terms of complexity, the size of the program produced by the core TABDUAL transformation is linear in the size of the input program, as shown in Theorem 1 of Appendix B, which is similar to that of ABDUAL using the folded dual form (cf. Lemma A.5 in [4]). It is known that the problem of query evaluation to abductive frameworks is NP-complete, even for those frameworks in which entailment is based on the WFS [12]. 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, since TABDUAL also employs the dual transformation. 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, the implementation aspects we mentioned in Section 6 may help improve performance in practice.

9.2 Related Work

There have been a plethora of work on abduction in logic programming, cf. [10, 20] for a survey on this line of work. But, with the exception of ABDUAL [4], we are not aware

[2] Dealing with NLoN is the only case where TABDUAL results in tabled normal logic programs, as it involves tabled negation $tnot$ in some transformed rules.

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.

Tabling has only been employed in ABDUAL limitedly, i.e. to table its meta-intepreter, 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 issues raised by the desirable reuse of tabled abductive solutions. TABDUAL generates a self-sufficient program transform, 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 [6]. 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 P, 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. However, 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.

TABDUAL, being implemented in XSB, is underpinned by WFS, which enjoys the relevance property, and thus it allows abduction to be performed by need only, induced by the top-down query-oriented procedure, solely for finding the relevant abducibles and their truth value, assuming the ICs are satisfied. This is not the case with the bottom-up approaches for abduction, e.g. [38], 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 [39]. TABDUAL also allows dealing with odd loops in programs because of its 3-valued program semantics, whilst retaining 2-valued abduction by-need 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 [37]. But it concerns itself with probabilistic logic programs, whereas TABDUAL concerns abductive normal logic programs. Moreover, the tabling technique in [37] 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 [33], 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.

9.3 Migration into Engine Level

The specification design of TABDUAL and its implementation, by means of a transformation in XSB-Prolog, produces a transformed program that aims at being near the potential uptake of certain operations by the underlying engine (and even other engines). We sketch some ideas on how to migrate key constructs of TABDUAL into an engine-level of Prolog systems that support tabling and, optionally, tries data structure, like XSB.

- *Tabling Abduction Entries*.
  This is the core feature of tabled abduction, which needs migration to the (tabling) engine to be more fruitful. At the object language level, we table only the output abductions entries and not the input abductive context to allow for reuse for one context to others, because the input abduction table entries are not included. Reuse and consistency are done at the language level, not inside the tabling level one.
  A new tabling mechanism could instead cater to the two extra table entries, concerning the input and output abducible sets, and provide the special lookup and update mechanisms pertaining to these special sets-arguments. Moreover, the sets would require an efficient store space recovery data structure representation consistent with the operations on them.
- *Hiding Data Structures*.
  The CWA list (and attending operations to detect positive loops in dualized negation - PLoN), which is being deployed at the language level, should migrate to the engine level, even disappearing from the generated code. New operations are needed concerning loop detection, in particular making PLoN succeed rather than fail, as it happens with direct positive loops.
  Similarly, the abductive context can be hidden from the object language and the operations on them moved into the engine level, but with the proviso that these could be inspected for debugging purposes. These signify that, avoiding the data structures being kept, and the operations on them carried out currently at language level, will much improve space and time efficiency.
- *Lazy Dual Transformation*.
  Currently the lazy (by-need) dual transformation is implemented using a trie data structure, allowing it to memoize generic dual rules which later can be retrieved and reused. Recall that the choice of trie is made to avoid generating generic dual rules eagerly, due to the default local table scheduling strategy of XSB.
  Implementing the lazy dual transformation at the language level consists in many operational details, which are realized by the trie manipulation predicates, e.g. tracking the choices of literals made in the dualization, etc. These details simulate the batched-like table scheduling, i.e. to return a generic dual rule at a time

(rather than to compute them all) before it is invoked. They can be lowered to the engine level, where we may still benefit from tabling (instead of using trie in the language level). But this requires that XSB permits admixtures of table scheduling strategies. That is, one would need to be able to use a batched table scheduling in the middle of the regular (and default) local table scheduling of the current XSB release.

9.4 Other Issues

Query Flexibility In transforming queries (cf. Section 3.2.4), the input context is set to [ ] by default. But one may explicitly launch a transformed query by constraining the input context, i.e. it is not necessarily [ ], but can be some given (non-empty) list of abducibles. Moreover, the output context can be constrained too. This is the case when one wants to verify that a given output context is indeed a desirable solution. This can be generalized to verifying, at the end, that the output context satisfies some property, e.g. it is a preferred one.

Optimizing Groundness We explained, in Section 4.1, a refinement to help ground negated subgoals in dual rules, by having all positive subgoals preceding the one that is negated due to the dualization. One may additionally keep the positive literals that succeed the negated literal, if that may help ground more; this often depends on the problem representation. It may be applicable to ICs better, as in the pragmatics of ICs that a user writes, the order of subgoals might be irrelevant. For instance, the (second layer) dual rules for $\mathit{false} \leftarrow q(X), r(X)$ may be defined as:

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

or better even, a symmetrical definition:

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

which is also declaratively correct. Some complication may arise, e.g. if the literal is an abducible which has variables, then it should be guaranteed that it is ground when it is moved forward in the body of the rule. This can easily be done by invoking, if necessary, a user supplied abducible's domain predicate that grounds it. The invocation of such domain predicate may be performed in the transformation of abducibles.

Refinement via Nogoods We have touched upon tabling nogoods of subproblems in Section 7.3, where we employed a particular TABDUAL system predicate to access and filter ongoing abductive solutions with respect to nogoods of subproblems. Indeed, ICs may also be relevant opportunities for tabling nogoods. Tabling nogoods can be provided by the evaluation of ICs, since ICs are transformed like any other rules, i.e. nogoods are tabled by predicate false_{ab}. This is a new orthogonal refinement that caters to the introduction and checking of nogoods within an abduction framework. The idea is similar to abducing solutions, but here we are also abducing some non-subsolutions and checking for compatibility. Such checking can be performed by the system predicate $produce/3$. Alternatively, we can leave it to the users to specify explicitly specific points where nogoods are to be generated and checked.

Quick-kill Option Another pertinent implementation aspect to tabled abduction is a 'quick-kill' in the first layer dual rule to immediately fail the rule, thereby avoiding the need to invoke the potentially more elaborate second layer dual rules, which will fail anyway. Indeed, conceptually an abductive solution of a negative subgoal $not\ p$ is construable as a set that negates the members of a hitting set for the abductive solutions of p. If one of the abductive solutions of p is empty then no hitting set exists. So the idea behind the 'quick-kill' is to permit to see whether goal '$not\ p$' has no hitting set at all, pertaining to the set of abductive solutions in p_{ab}. This is done by inspecting whether p_ab has an empty abductive solution entry. Taking Example 4, predicate not_p can now be defined as follows:

$$not\_p(I,O) \leftarrow tnot(p_{ab}([\,])), p^{*1}(I,T), p^{*2}(T,O).$$

i.e. by having the subgoal $tnot(p_{ab}([\,]))$ before the second layer dual rules are invoked. As a matter of fact, in TABDUAL, such hitting sets are generated incrementally, by means of finding abductive solutions to the dual rules of p, without thus having to wait for the explicitly availability of all abductive solutions for p. Nevertheless, the 'quick-kill' can be a readily available option, just in case there exists an empty abductive solution for p. It may be serve as an optimization, as it simply detects if such an entry is already in the table for p, rather than generating solutions for A trying to produce the empty one. Its use surely depends on the problem. If a given query to the problem consists of only negative subgoals, then this 'quick-kill' may instead be an overkill, as it unnecessarily invokes the corresponding positive goals and tables their solutions, only to check whether an empty abductive solution is obtained. In this case, one may simply benefit from the second layer dual rules in answering the query, without the 'quick-kill' option.

10 Conclusion and Future Work

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 employs the 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 program with variables and loops, as well as to make TABDUAL more efficient and flexible. It has been evaluated with various objectives in mind, by employing several variants of the same underlying implementation, 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, lazy or eager, are options for TABDUAL customization. TABDUAL still

has much room for improvement, which we discuss and detail in Section 9, including migrating its features to the engine level of Prolog systems.

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.

Future work will consist in continued exploration of our applications of abduction, which will provide feedback for system improvement. Another research line pertains to the issue that, whenever discovering abductive solutions, i.e., explanations, for some given primary observation, one may wish to check too whether some other given additional secondary observations are true, being a logical consequence of the abductive explanations found for the primary observation, i.e. side-effects of abduction [29]. In other words, whether the secondary observations are plausible in the abductive context of the primary one. We look forward to incorporating side-effects and integrating other logic programming features, e.g. program updates and uncertainty, into TABDUAL.

It is part and parcel of our research plan to employ the integrated sytem for moral reasoning; a field which has recently gained attention and a resurgence of interest from AI community, and on which we work [17, 30].

Acknowledgements 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.

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 System. `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. 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.
6. V. P. Ceruelo. Negative non-ground queries in well founded semantics. Master's thesis, Universidade Nova de Lisboa, 2009.
7. 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.
8. J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

9. 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.
10. M. Denecker and A. C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*. Springer Verlag, 2002.
11. J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
12. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.
13. K. Eshghi. Abductive planning with event calculus. In *Proc. Intl. Conf. on Logic Programming*. The MIT Press, 1988.
14. T. H. Fung and R. Kowalski. The IFF procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, 1997.
15. 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.
16. 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.
17. T. A. Han, A. Saptawijaya, and L. M. Pereira. Moral reasoning under uncertainty. In *Procs. of The 18th Intl. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-18)*, volume 7180 of *LNCS*, pages 212–227. Springer, 2012.
18. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *J. of Logic Programming*, 27(2):107–136, 1996.
19. J. R. Josephson and S. G. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge U. P., 1995.
20. 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.
21. A. C. Kakas and P. Mancarella. Knowledge assimilation and abduction. In *Intl. Workshop on Truth Maintenance*, ECAI'90, 1990.
22. 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.
23. R. Kowalski and F. Sadri. Abductive logic programming agents with destructive databases. *Annals of Mathematics and Artificial Intelligence*, 62(1):129–158, 2011.
24. P. Lipton. *Inference to the Best Explanation*. Routledge, 2001.
25. 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.
26. 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.
27. 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.
28. 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.
29. L. M. Pereira and A. M. Pinto. Inspecting side-effects of abduction in logic programs. In M. Balduccini and T. C. Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in honour of Michael Gelfond*, volume 6565 of *LNAI*, pages 148–163. Springer, 2011.
30. L. M. Pereira and A. Saptawijaya. Modelling Morality with Prospective Logic. In M. Anderson and S. L. Anderson, editors, *Machine Ethics*, pages 398–421. Cambridge U. P., 2011.

31. L. M. Pereira and A. Saptawijaya. Abductive logic programming with tabled abduction. In *Procs. 7th Intl. Conf. on Software Engineering Advances (ICSEA)*, pages 548–556. ThinkMind, 2012.
32. 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.
33. 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.
34. 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.
35. A. Saptawijaya and L. M. Pereira. Towards practical tabled abduction in logic programs. In *16th Portuguese Conference on Artificial Intelligence (EPIA)*, LNAI. Springer, 2013.
36. A. Saptawijaya and L. M. Pereira. Towards practical tabled abduction usable in decision making. In *Procs. 5th. KES Intl. Symposium on Intelligent Decision Technologies (KES-IDT)*, Frontiers of Artificial Intelligence and Applications (FAIA). IOS Press, 2013.
37. T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. of Artificial Intelligence Research (JAIR)*, 15:391–454, 2001.
38. 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.
39. K. Satoh and N. Iwayama. Computing abduction by using TMS and top-down expectation. *Journal of Logic Programming*, 44(1-3):179–206, 2000.
40. 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.
41. 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.
42. T. Swift and D. S. Warren. Personal communications, February 2012, 2013.
43. A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.

A Specification of the Core TABDUAL Transformation

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. 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 $p(t_1, \ldots, t_n)$,[3] 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.

Motivation, informal description, and examples related to the following definitions can be found in Section 3.2.3.

A.1 Program Transformation

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,[4] *and $E_0 = \mathcal{A}_r$.*
2. *For every predicate p/n defined in P, we define $\tau^+(p)$:*

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

   where $produce/3$ is a TABDUAL *system predicate.*[5]

Example 18. Consider the following program P, where rules are named with r_i and $a/1$ is an abducible.

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

We have $\mathcal{A}_{r_i}$ and r'_i, for $1 \leq i \leq 3$, as follows:[6]

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

[3] 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.

[4] Variables E_i serve as abductive contexts.

[5] Predicate $produce/3$ is explained in Section 3.2.1.

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

– $\mathcal{A}_{r_3} = [\,]$ and $r'_3 :\ v(X, X, s(X))$.

The transformation of Definition 1 results in:

$$\begin{array}{ll}
\tau'(r_1) : u_{ab}(0, _, [\,]). & \\
\tau'(r_2) : u_{ab}(s(X), Y, E_2) & \leftarrow v(X, Y, Z, [a(X)], E_1),\ not_w(Z, E_1, E_2). \\
\tau'(r_3) : v_{ab}(X, X, s(X), [\,]). & \\
\tau^+(u) : u(X_1, X_2, I, O) & \leftarrow u_{ab}(X_1, X_2, E),\ produce(O, I, E). \\
\tau^+(v) : v(X_1, X_2, X_3, I, O) & \leftarrow v_{ab}(X_1, X_2, X_3, E),\ produce(O, I, E).
\end{array}$$

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.

Definition 2 (The Dual Transformation).

1. *For every predicate p/n, with $n \geq 0$, defined in P:*

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

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.[7]

(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:

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

where σ is defined as follows:

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

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 $[\,]$.[8]

[7] Variables T_i serve as abductive contexts.

[8] This means, when $p/0$ is defined as a fact in P, we have $not\_p(T_0, T_1) \leftarrow p^{*1}(T_0, T_1)$ in the first layer, but there is no rule of $p^{*1}/2$ in the second layer. Equivalently, it may be defined as $not\_p(\_, \_) \leftarrow fail$. (cf. the dual rule of predicate $r/0$ in Example 4).

2. *For every predicate r/n, $n \geq 0$, in P, that has no definition, we define $\tau^-(r)$:*[9]

$$not\_r(\bar{X}, I, I).$$

Example 19. Recall program P in Example 18. The transformation of Definition 2 results in:

$$\begin{array}{rll}
\tau^-(u) : & not_u(X_1, X_2, T_0, T_2) & \leftarrow u^{*1}(X_1, X_2, T_0, T_1),\ u^{*2}(X_1, X_2, T_1, T_2). \\
\tau^-(v) : & not_v(X_1, X_2, X_3, T_0, T_1) & \leftarrow v^{*1}(X_1, X_2, X_3, T_0, T_1). \\
\tau^-(w) : & not_w(X, I, I). & \\
\tau^-(\mathit{false}) : & \mathit{not_false}(X, I, I). & \\
\tau^*(u) : & u^{*1}(X_1, X_2, I, I) & \leftarrow [X_1, X_2] \neq [0, _]. \\
 & u^{*2}(X_1, X_2, I, I) & \leftarrow [X_1, X_2] \neq [s(X), Y]. \\
 & u^{*2}(s(X), Y, I, O) & \leftarrow not_a(X, I, O). \\
 & u^{*2}(s(X), Y, I, O) & \leftarrow not_v(X, Y, Z, I, O). \\
 & u^{*2}(s(X), Y, I, O) & \leftarrow w(Z, I, O). \\
\tau^*(v) : & v^{*1}(X_1, X_2, X_3, I, I) & \leftarrow [X_1, X_2, X_3] \neq [X, X, s(X)].
\end{array}$$

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:

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

where $insert/3$ is a TABDUAL *system predicate.*[10] *Mark that, in the body of the second rule, 'not a' is used instead of 'not_a'.*

Example 20. Recall program P in Example 18. We have $\mathcal{A}_P = \{a(X)\}$. The transformation of Definition 3 results in:

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

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:*

- $\tau'(P) = \{\tau'(r) \mid r \in P\}$
- $\tau^+(P) = \{\tau^+(p) \mid p \in \mathcal{P}_P$ *and p is defined*$\}$
- $\tau^-(P) = \{\tau^-(p) \mid p \in \mathcal{P}_P\}$
- $\tau^*(P) = \{\tau^*(p) \mid p \in \mathcal{P}_P$ *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 21. The set of rules obtained in Example 18, 19, and 20 forms $\tau(P)$ of program P.

[9] In particular, if there is no integrity constraint in P, we have $\tau^-(\mathit{false}) :\ \mathit{not\_false}(I, I)$.
[10] Predicate $insert/3$ is explained in Section 3.2.3.

A.2 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:*

$$?-\ G_1,\ \ldots,\ G_m.$$

TABDUAL transforms query Q_P into:

$$?-\ \delta(G_1),\ \ldots,\ \delta(G_m), not\_\mathit{false}(T_m, O).$$

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), $1 \leq i \leq m$, T_i, O are fresh rule variables.[11]

Example 22. Recall program P in Example 18. Query:

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

is transformed by Definition 5 into:

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

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

B Complexity of the Core TABDUAL Transformation

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*

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

where $|\mathcal{B}_{r_i}|$ denotes the number of body literals in r_i.[12]

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

Theorem 1. *Let P be an abductive normal logic program and $\mathcal{A}_P$ be the set of abducible atoms in P. Then $size(\tau(P)) < 13.size(P) + 4.|\mathcal{A}_\mathcal{P}|$.*

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, the total size of $\tau^*(p_i)$ is $2.(size(P|_{p_i}) - m) + 2$. Therefore, the size of the transformed program in P by τ^* for all predicates defined in P will be $size(\tau^*(P)) = \Sigma_{i=1}^{heads(P)} (2.(size(P|_{p_i}) - m) + 2) = 2.size(P) - 2.rules(P) + 2.heads(P)$.

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

– 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 for $heads(P)$ and $rules(P)$.
Thus, $size(\tau(P)) = size(\tau'(P)) + size(\tau^{+}(P)) + size(\tau^{-}(P)) + size(\tau^{*}(P)) + size(\tau^{\circ}(P)) < 13.size(P) + 4.|\mathcal{A}_{\mathcal{P}}|$.

C Test-suite

The test-suite consists of two collections of programs: ground programs with loops, and programs with variables (also containing loops). The TABDUAL variant used in their evaluation is TABDUAL$^{\infty}$+lazy. In the evaluation of ground programs with loops, a comparison with the ABDUAL meta-interpreter [4] is made. Both systems run on the same platform under XSB version 3.3.7.

C.1 Programs with Loops

Collection of Programs Figure C.1 lists a collection of programs, expressly including difficult cases, used to compare TABDUAL and ABDUAL. 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.

$p_0 \leftarrow q_0.$ $p_0 \leftarrow a.$ $q_0 \leftarrow p_0.$ $q_0 \leftarrow b.$

$p_1 \leftarrow not\ q_1, r_1.$ $r_1 \leftarrow not\ q_1, p_1.$ $q_1 \leftarrow not\ p_1.$

$p_2 \leftarrow q_2.$ $q_2 \leftarrow r_2.$ $r_2 \leftarrow p_2.$

$p_3 \leftarrow q_3.$ $q_3 \leftarrow not\ r_3.$ $r_3 \leftarrow p_3.$

$p_4 \leftarrow q_4.$ $q_4 \leftarrow p_4.$ $q_4 \leftarrow not\ a, not\ b.$

$p_5 \leftarrow q_5.$ $q_5 \leftarrow not\ r_5.$ $r_5 \leftarrow not\ s_5.$ $s_5 \leftarrow p_5.$

$p_6 \leftarrow not\ q_6.$ $q_6 \leftarrow r_6.$ $r_6 \leftarrow s_6.$ $s_6 \leftarrow not\ p_6.$

$p_7 \leftarrow not\ q_7, r_7, a.$ $r_7 \leftarrow not\ q_7, p_7, b.$ $q_7 \leftarrow not\ p_7, not\ r_7.$

$p_8 \leftarrow not\ q_8, a.$ $q_8 \leftarrow not\ p_8.$ $q_8 \leftarrow b.$

$p_{10} \leftarrow not\ q_{10}, a.$ $q_{10} \leftarrow p_{10}, a.$

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

$p_{12} \leftarrow a, not\ q_{12}.$ $q_{12} \leftarrow not\ a, p_{12}.$

Fig. 5. Collection of Ground Programs with Loops

Evaluation Results Table C.1 compares the results returned by TABDUAL and ABDUAL for queries to the ground programs in Figure C.1.

C.2 Programs with Variables

Collection of Programs Figure C.2 lists programs with variables; many of them contain loops as well. In this collection, $a/1$, $b/1$, and $c/1$ are abducibles.

Table 3. Comparison of results: TABDUAL vs. ABDUAL

Queries	TABDUAL	ABDUAL
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}	$[a]$	$[a]$
$not\ p_{12}$	$[not\ a]$	$[not\ a]$
$not\ q_{12}$	$[a], [not\ a]$	$[\ ], [a], [not\ a]$

Evaluation Results Table C.2 presents the evaluation results returned by TABDUAL and ABDUAL for queries to the ground programs in Figure C.2.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 6. Collection of Programs with Variables

Table 4. 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).