Abductive Logic Programming with Tabled Abduction

Luís Moniz Pereira*, Ari Saptawijaya *†
*Centro de Inteligência Artificial (CENTRIA)
DI/FCT Universidade Nova de Lisboa
2829-516 Caparica, Portugal
Email: lmp@fct.unl.pt, ar.saptawijaya@campus.fct.unl.pt
†Fakultas Ilmu Komputer Universitas Indonesia
Kampus UI Depok 16424, Indonesia

***Abstract*—In abductive logic programming, abductive solutions are typically computed without attending to the abductive context. These abductive solutions can actually be reused in a different abductive context. In this paper, we employ a tabling mechanism and propose a tabled abduction mechanism, that consists of a transformation from abductive normal logic programs into tabled dual programs, by tabling abductive solution entries and without requiring any meta-interpreter. Recomputation of abductive solutions for a different context, but consistent with them, can then be avoided, by reusing the tabled abductive solution entries. Though our implementation is in XSB-Prolog, its concepts may be imported to other systems, not necessarily Logic Programming ones.**

***Keywords-tabled abduction*; *abduction transformation*; *well-founded semantics*; *XSB-Prolog*.**

I. Introduction

Abductive logic programming offers a formalism to declaratively express and solve problems in a variety of areas, e.g., decision-making, diagnosis, planning, belief revision and hypothetical reasoning (cf. [1]–[4]). On the other hand, the tabling mechanism, now supported by a variety of Prolog systems, ensures termination and optimal complexity for query evaluation to a large class of logic programs, viz. under the Well-Founded Semantics [5].

We explore the idea of how to benefit from the tabling mechanism in order to reuse priorly obtained abductive solutions in a given new abductive context. For so doing, we propose a tabled abduction system, dubbed TABDUAL, which includes a transformation (itself a logic program) from abductive normal logic programs into tabled dual programs, that allows query evaluation without resorting to a meta-interpreter. The transformation makes use of the formal dual transformation defined in [6], and implements an innovative and pragmatic re-uptake of prior abductive solution entries in tabled predicates. More precisely, we contribute on the following aspects:

- We cater for various critical issues in the program transformation, illustrated in successive prototypical examples. They incrementally show how the transformation evolves from the basic core idea into its current actual state, by gradually dealing with the unavoidable issues arising, viz. introducing tabled predicates, dealing with direct positive loops as well as positive and negative loops over negation, handling programs with variables and also non-ground queries.
- We have developed a prototype for TABDUAL based on its specification [7]. It has been tested with benchmark core examples pertaining to each of the above mentioned issues, under the latest version of XSB (version 3.3.6).
- We discuss some ideas on how to migrate core features of TABDUAL into the underlying engine-level of Prolog systems wanting to encompass tabled abduction.

The paper is structured as follows. Section 2 reviews basic logic programming and abductive logic programming notions. The motivation and the key concept of tabled abduction, along with its implementation, and related work are then discussed, in Section 3. The ideas on how to migrate key ingredients of TABDUAL into an engine-level of Prolog systems are discussed in Section 4, and we conclude in Section 5.

II. Abductive Logic Programming

We briefly review in this section the abductive logic programming formalism, which has been well studied for a few decades by now [8]–[13]. We start with basic logic programming background. A *logic rule* has the form

$$H \leftarrow B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$$

where $n \geq m \geq 0$ and H, B_i with $1 \leq i \leq n$ are atoms. H and $B_1, \ldots, B_m, not\ B_{m+1}, \ldots, not\ B_n$ are the head and the body of the rule, respectively. Any variables occurring in a rule are universally quantified. 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 write 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 with the empty head, or equivalently with *false* in the head, is called an *integrity constraint* (IC) or *denial*. A *normal logic program* is a set of such logic rules, where non-ground rules (i.e., rules containing variables) stand for all

 ISBN: 978-1-61208-230-1

their ground instances. A program may represent an infinite set of rules, when it contains at least one function symbol. For example, consider the following program with two rules:

$$nat(0). \qquad nat(s(X)) \leftarrow nat(X).$$

The second rule stands for an infinite number of its ground instances (as we have one function symbol s and a constant 0), namely:

$$\begin{array}{lcl} nat(s(0)) & \leftarrow & nat(0). \\ nat(s(s(0))) & \leftarrow & nat(s(0)). \\ nat(s(s(s(0)))) & \leftarrow & nat(s(s(0))). \\ \ldots \text{ etc.} & & \end{array}$$

Logic program theoretical semantics are usually defined by reference to all the ground instances of a program's rules. This set of instances may be infinite even when the set of rules (with variables) is finite, as exemplified above. Rule bodies however must, by definition, be finite. In programming practice, necessarily finite programs only, albeit with variables, are employed. Consequently, note that in the construction of the dual program, discussed in Section III, we ensure that its rules' bodies are still finite, because our transformation applies to possibly non-ground but nevertheless finite programs.

Abduction, or inference to the best explanation (a common designation in the philosophy of science), is a reasoning method whereby one chooses those hypotheses that would, if true, best explain observed evidence.

An *abductive logic program* is a normal logic program that allows for abducibles (i.e., positive literals with no rules in the program), or their default negations, to appear in the body of rules. Abducibles stand for hypotheses, whose truth value is not assumed initially. Queries to the program represent the evidence we would like to explain, by resorting to a given set of permitted abducibles. Abducibles in rules' bodies may have arguments, but must be a ground instance on the occasion of their abduction. Note that the negation '$not\ A$' of an abducible A does not refer to its default negation, but instead to the explicitly assumed negation of A. The truth value of abucibles may be independently assumed *true* or *false*, either in their positive or negated form, as the case may be, in order to produce an abductive solution to a query, by means of a consistent set of assumed hypotheses.

An *abductive solution* to a query is thus a consistent set of abducible instances or their negations that, when substituted by their assigned value *true* 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*. Because explicit negation entails default negation, the substitution above is warranted. Often, an abductive solution S for goal G in program P is defined as a set of abducibles consistent with P such that $P \cup S \models G$ in the semantics employed. The above definition of abductive solutions is consistent with this one, but avoids the complication of introducing the positive and the negative abducibles of a solution as facts. Instead, they are replace by true wherein the program. Note that though the semantics may be other than two-valued, the abductive solutions themselves are enforced two-valued by definition.

When performing abductive reasoning, we typically wish to find, by need only, via top-down computation, the abductive solutions to a query. This top-down computation, dubbed backward chaining, is possible only when the underlying semantics is relevant, a property enjoyed by the Well-Founded Semantics (WFS) [5]. That is, it guarantees that it is enough to use only the rules relevant to the query (those in its procedural call-graph) to find its truth value, thus it avoids computing a whole model in order to find an answer to a query. In order to satisfy the ICs too, we conjoin each query with '*not false*'. Mark that when a semantics enjoys relevancy, the values of abducibles not mentioned in the abductive solution are indifferent to the solution. We use WFS in the implementation of our prototype tabled abduction system, TABDUAL, described in Section III-E.

III. Tabled Abduction

We begin with issues that motivate the need for tabled abduction, and subsequently propose a novel abduction system, termed TABDUAL, that involves a transformation of an abductive logic program into another program with tabled abductive solutions.

A. Motivation and Related Work

An abductive solution to a problem is typically formed from the abductive solutions of its subproblems. Consider the following abductive logic program.

$$q \leftarrow a. \qquad r \leftarrow b, q. \qquad p \leftarrow r, q.$$

where a and b are abducibles. Suppose three queries: q, r and p, are launched, in that order. Query q simply gives $[a]$ as the abductive solution. The next query, r, is typically solved by abducing b and followed by invoking q. But since q has previously been invoked, query r can in fact be solved by reusing the previously obtained abductive solution $[a]$ of q and extending it with the newly abduced abducible b; yielding $[b, a]$ as the abductive solution to r. One may also view that solving query r amounts to extending the ongoing current abductive context $[b]$ when q is invoked, with the priorly discovered and registered in a table, abductive solution $[a]$ of q, resulting in $[b, a]$. Using similar reasoning, the final query p can be solved by reusing the abductive solutions of r and q. More precisely, we can view that the abductive solution $[b, a]$ of r in the body of p becomes the current abductive context of q and since it subsumes the previously obtained abductive solution $[a]$ of q we can safely have $[b, a]$ as the abductive solution to query p.

 ISBN: 978-1-61208-230-1

The above example illustrates how abductive solutions can actually be reused in a new abductive context, avoiding unnecessary recomputation of abductive solutions to sub-problems, and thereby gaining in efficiency (imagine if the definition of the first rule q is scaled up by another huge program). One may observe that the table size would be proportional to the number of distinct (positive) goals in the procedural call-graph, i.e., each first call of the goals in a given query will table, as the solution entry, the abductive solutions of the called goal.

Reusing solutions in logic programming, without abduction, is commonly performed using the tabling mechanism. It therefore allows dealing with loops in the program, ensuring termination of looping queries. Surprisingly, to the best of our knowledge, no work has addressed how reusing abductive solutions, as we hinted above, can be realized using the tabling mechanism. Tabling has only been employed limitedly, i.e., to table a meta-intepreter, which in turn allows abduction to be performed in the presence of loops in a program, but with no analysis of abductive solution subsumption at all (cf. ABDUAL [6], [14]).

Our current work pushes the benefit of the tabling mechanism to abduction, by employing it to table abductive solution entries (effectively achieved by tabling language-level predicates carrying these entries) for their later reuse in new abductive contexts, by abductive subsumption.

Our approach differs from that of [15]. Therein, abducibles are coded as odd loops, it is compatible with and uses constructive negation, and 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.

Like ABDUAL [6], we use the dual transformation and rely on the same theoretic underpinnings, but the ABDUAL code caters only for ground programs and queries. It also requires a meta-interpreter which makes tabled abduction awkward because it is enacted only at the level of tabled meta-interpreter predicates, and does not cater for abductive subsumption. In short, it affords no particular treatment for the tabling of abduction. Moreover, by design of omission, it does not address at all the issues raised by the desirable reuse of tabled solutions. We employ no meta-interpreter, but generate a self-sufficient program transform. Hence, it avoids meta-interpretation and, moreover, explicitly addresses the concerns of making better use of tabling for abduction, so that known abductive solutions may be appropriated by subsequent abductive goals. We do so not just in core conception, but also adumbrating optimizations and quick-kill trickery.

Our tabled abduction implementation, termed TABDUAL, also allows dealing with programs containing loops, and thus extending the usual tabling mechanisms. We have conducted an experiment to compare TABDUAL with the meta-interpreter ABDUAL [14] using our test suite of programs

```
abds([a/0,b/0,c/0]).

p0 <- q0.       p3 <- q3.          p4 <- q4.
p0 <- a.        q3 <- not r3.      q4 <- p4.
q0 <- p0.       r3 <- p3.          q4 <- not a,not b.
q0 <- b.

p8 <-  not q8, a.        p11 <-  not q11, a.
q8 <-  not p8.           q11 <-  p11,not a.
q8 <-  b.
```

Figure 1. Some programs with loops from the test suite. Predicate $abds$ lists the abducibles with their corresponding arity.

Table I
COMPARISON OF RESULTS: TABDUAL VS. ABDUAL FOR FIGURE 1

Queries	TABDUAL	ABDUAL
not p0	[not a, not b]	[not a, not b], [not a]
p3	[] undefined	[]
not p3	[] undefined	[]
not p4	[a], [b]	[a], [b], [a,b]
q8	[] undefined, [not a], [b]	[not a], [b]
not q11	[a], [not a]	[], [a], [not a]

with various kinds of loops. Some distinguishing results, with respect to the programs given in Figure 1, are shown in Table I; the complete result is available in [7]. TABDUAL provides more correct and complete results within the test suite, given that our scope of programs and queries is more general than ABDUAL's:

- For query $not\ p0$, $[not\ a, not\ b]$ should be the only solution, because $not\ p0$ succeeds by abducing $not\ a$ **and** failing $q0$. To fail $q0$, $not\ b$ has to be abduced **and** $p0$ has to fail. Here, there is a positive loop on negation between $not\ p0$ and $not\ q0$, so the query succeeds and gives the solution $[not\ a, not\ b]$ as the only solution.
- For queries $p3$ and $not\ p3$, unlike ABDUAL, TABDUAL returns *undefined* (and abduces nothing) as expected, due to the negative loops over negation.
- Query $not\ p4$ shows that TABDUAL does less abduction than ABDUAL, by abducing a or b only; not both.
- For query $q8$, TABDUAL has an additional answer: [] *undefined* (i.e., undefined by abducing nothing), due to the negative loop over negation between $q8$ and $p8$. Similar reasoning equally applies to query $not\ p8$. This additional answer is missing by ABDUAL.
- For query $not\ q11$, the first solution is obtained by abducing a to fail $q11$. Another way to fail $q11$ is to fail $p11$, 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 $q11$ in the program, hence $not\ q11$ will never succeed with [] abductive solution, as returned by ABDUAL.

We have not yet compared efficiency, but it seems apparent that TABDUAL is an improvement over ABDUAL.

TABDUAL does not concern itself with constructive negation, like the NegABDUAL system [16] and its follow-up [17]. NegABDUAL uses abduction to provide constructive negation plus abduction, by making the disunification predicate '$\backslash=/2$' an abducible. Again, it does not concern itself with the issues of tabled solutions reuse, which is the main purpose of TABDUAL.

NegABDUAL transforms programs and uses minimal meta-interpretation. Its transformations are intricate, especially on account of its constructive negation design goal, which is no concern of TABDUAL. In the past, we have used it extensively; but, for now, we have not yet compared it to TABDUAL. It will surely be heavier and not benefit from our abductive solutions tabling. However, because of its constructive negation ability, NegABDUAL can deal with problems that TABDUAL does not, and intends not, precisely because it aims at being lighter and more adaptable. 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 $Y\backslash=1$, as long as one assumes there are at least two constants in the Herbrand Universe. Indeed, the local variable Y in the dualizing transmutation of $p(X)$ produces a default negation. In TABDUAL implementation, there is no floundering. However, distinct from NegABDUAL, our TABDUAL answers 'no' to $not\ p(X)$, which is correct, even in the absence of a conditional answer afforded only by having constructive negation in place.

To the best of our knowledge, no other work on using tabling and dual programs for abduction exists, nor has the problem of tabled abduction that we currently addressed even been formulated by others.

We next introduce TABDUAL through a sequence of prototypical examples, which illustrates how, for easier understanding, it incrementally evolves to cope with the major issues of concern arising in tabled abduction. We begin with the key transformation employing the very idea of tabling and of reusing abductive solutions. We further successively argue why and how more constructs should be added to the transformation, to gradually deal with the issues involved, as illustrated in ever more complex and demanding examples.

In TABDUAL, we introduce a so-called *abductive context* (referred to "context" hereafter), as illustrated in the example at the beginning of this section, for every predicate defined by the transformation. The context indicates the ongoing tentative abductive solution for any given tabled goal. It allows to relay the abductive solution from the head to the body of a rule and back, and from one subgoal to subsequent subgoals. The context is effectively catered for and realized by adding two extra arguments to every predicate, as defined by the transformation: one for the input context and the other for the output context of any ongoing abductive solution for the predicate. Abducibles are 'parsed' as terminals in an ongoing derivation and tracked in these additional arguments.

```
1.  :- table p_ab/1, q_ab/1.
2.  p_ab(E) :- q([a],E).
3.  q_ab([b]).          q_ab([c]).
4.  p(I,O) :- p_ab(E),produce(O,I,E).
5.  q(I,O) :- q_ab(E),produce(O,I,E).
6.  p_st(I,O) :- p_st_1(I,O).
7.  q_st(I,O) :- q_st_1(I,T),q_st_2(T,O).
8.  p_st_1(I,O) :- not_a(I,O).
9.  p_st_1(I,O) :- not_q(I,O).
10. q_st_1(I,O) :- not_b(I,O).
11. q_st_2(I,O) :- not_c(I,O).
12. not_p(I,O) :- tnot(p_ab([])),p_st(I,O).
13. not_q(I,O) :- tnot(q_ab([])),q_st(I,O).
```

Figure 2. Main rules obtained from the transformation of P_1

We fix some notation. We use capital letters to denote variables appearing in a program and write P/N to denote that predicate P has arity N. In the examples, the set of abducible atoms are declared in the predicate $abds/1$. In the result of the transformation, we use $:-$ to separate the head and the body of a rule, instead of $\leftarrow$. As the implementation is done in XSB Prolog [18], [19], we borrow from it its syntax, e.g., for the tabled negation $tnot/1$ and for the compiler directive to declare tabled predicates.

B. Basic Idea

We start with the basic idea by means of the example below. Consider ground program P_1 below.

$$abds([a/0, b/0, c/0]). \quad p \leftarrow a, q. \quad q \leftarrow b. \quad q \leftarrow c.$$

The transformation produces, for every defined predicate in P_1 ($p/0$ and $q/0$), several sets of rules. The transformation of P_1 contains the main rules as shown in Figure 2.

The first set of rules defines the tabled predicates $p\_ab/1$ and $q\_ab/1$, declared in line 1. The tabled predicate essentially tables the abductive solution entry, always assuming, for facilitating reuse, empty input context. Since the input context calls for tabled predicates are always empty, only one extra argument (i.e., for solution entry context) is needed. The p_ab rule (line 2) is derived from the p rule of the original program, where p is defined by the subgoals q and by abducing a. The tabled entry E of p_ab is passed from the output context of q, i.e., the second argument of the subgoal q. Since q rules in the original program are defined solely by abducibles, they are transformed into facts q_ab with those abducibles as their abductive solution entries (line 3).

The second set of rules provides the definition of $p/2$ and $q/2$, now with the input and output contexts I and O (lines 4-5). Predicate $p/2$, for example, reuses the abductive solution entry E from tabled predicate p_ab and then, using it together with the input context I to produce its output context O. Similar reasoning applies equally to predicate $q/2$.

 ISBN: 978-1-61208-230-1

The context updating is performed externally by predicate $produce/3$. It concerns itself with: whether E is already contained in I and 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 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, eliminating any redundant abduction.

The third set of rules (lines 6-11) contains the *dual* rules, $p\_st/2$ and $q\_st/2$, where p_st is true iff p of the original program is false [6]; similar reasoning also applies to $q\_st/2$. Note that the st stands for '$*$', as in p^*, a notation often used in abduction to denote the negation of p. The set of dual rules is defined in two layers. The first layer (lines 6-7) captures the idea that, e.g., to make q_st (line 7) true, we need to (non-deterministically) fail each q rule: q_st is defined by q_st_1 and q_st_2 that correspond to both failing the first and the second q rules, respectively. Note that the abductive solution from subgoal q_st_1 is relayed to the subsequent subgoal q_st_2 via the intermediate context T. The second layer of q's dual rules (lines 10-11) defines how to fail each q rule, by alternatively and non-deterministically failing one subgoal in q's body at a time, i.e., by negating just one literal in q's body at a time, in order to avoid excessive abduction. The negated literal is renamed into its corresponding positive one, e.g., $not\ a$ into not_a, and also equipped with the input and output contexts. The two-layer dual definition for p (lines 6 and 8-9) follows similar reasoning. The first layer p_st (line 6) is simpler, because in the original program p has only one rule.

The fourth set of rules consists of the *negated* rules $not\_p/2$ and $not\_q/2$ (lines 12-13). These will be the ones that generate abductive solutions for the such 'positive' literal, say not_p. Indeed, every default literal is replaced with its corresponding positive. In line 12, predicate $not\_p(I,O)$ is defined by two subgoals: $tnot(p\_ab([]))$ and $p\_st(I,O)$. The first subgoal serves as an optimization, to immediately fail not_p without the need to launch the call to the more elaborate p_st rule that follows. The idea behind the 'quick-kill' $tnot(p\_ab([]))$ is to permit to see whether goal '$not\ p$' has no hitting set at all, pertaining to the set of abductive solutions of p_ab. This is done by inspecting whether p_ab has an empty abductive solution entry, i.e., p can be satisfied without abducing anything, in which case not_p can immediately fail. Indeed, an abductive solution of the negation of positive atom A is construable as a set that negates the members of a hitting set for the abductive solutions of A. If one of these latter sets is empty then no hitting set exists. Our approach consists in generating such hitting sets incrementally, by means of finding abductive solutions to the dual rules of A, without thus having to wait for the explicitly availability of all abductive solutions for A. Nevertheless, a quick-kill option is readily available, just in case there exists an empty abductive solution for A. An optimization consists in simply detecting if such an entry is already in the table for A, rather than generating solutions for A trying to produce the empty one.

In addition to the rules shown in Fig 2, for each predicate having no rule in the original program, a fact about its negation is added. For example, since there is no IC defined in P_1, fact $not\_false(I,I)$ is added. Note that the two contexts are the same. Having no body, the output context does not depend on the context of any other goals, but depends only on its corresponding input context. When ICs exist, they are transformed exactly like the dened predicates.

Finally, for each abducible, a pair of rules is created: a rule for the positive abducible, e.g., $a(I,O) :- insert(a,I,O)$, and another rule for its negation, e.g., $not\_a(I,O) :- insert(not\ a,I,O)$. Similar pairs of rules are also added for abducibles b and c, and their negations. Note that the rules are defined by the external predicate $insert/3$, which inserts the corresponding abducible with respect to the input context I and results in the output context O. It maintains a consistent context during the insertion, and avoids redundant abduction.

A query to a program, consequently, should be transformed too, in order to conform to the transformation: positive goals are augmented with the two extra arguments for the abductive context, whereas negative goals are made 'positive' in addition to the two extra context arguments. Moreover, a query should always be conjoined with $not\_false/2$ to ensure that all ICs are satisfied. For example, query $not\ p$ is transformed into $not\_p(I,O)$. Its complete call, as a top goal, becomes $not\_p([],T),\ not\_false(T,O)$, where O is an abductive solution to the query, given initially an empty input context. Note, how the abductive solution for not_p is further constrained by passing it to the subsequent subgoal not_false for confirmation, via the intermediate context T.

Note that at this point we are not concerned with incremental IC checking, as this is not a specific tabled abduction problem, but a general tabling problem that others are addressing and that tabled abduction does not preclude any reuse of.

C. Dealing with Loops

The next examples concern programs involving loops between predicates. Consider the ground program P_2 below.

$$abds([a/0]). \qquad p \leftarrow q, a. \qquad q \leftarrow p.$$

XSB with its tabling mechanism supports Well-Founded Semantics [5], which would detect direct positive loops and fail predicates involved in such loops. For P_2, query p fails, due to the direct positive loop between tabled predicates $p\_ab/1$ and $q\_ab/1$. On the other hand, query $not\ p$ should succeed with two abductive solutions $[]$ and $[not\ a]$. The call to the latter query, after transformation,

 ISBN: 978-1-61208-230-1

becomes $not\_p([], T), not\_false(T, O)$. Instead of succeeding, the first subgoal $not\_p([], T)$ will loop indefinitely! This loop occurs because of the mutual dependency between $not\_p/2$ and $not\_q/2$ through $p\_st\_1/2$ and $q\_st\_1/2$. 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. Indeed, since any source program's direct positive loops must fail, the loops between their corresponding transformed negations must succeed [6]. For example, whereas $r \leftarrow r$ fails query r, perforce $not\_r \leftarrow not\_r$ succeeds query not_r. The problem can be remedied by detecting such loops in a program. Since XSB's tabling mechanism already supports dealing with direct positive loops, we need only concern ourselves with positive and negative loops on negation in the transform.

*1) **Positive Loops on Negation (PLoN)**:* We detect PLoN 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 and serves as another extra argument for the dual and negated rules. The new transformation, with PLoN detection, of P_2 is shown below (without showing the usual transformation for not_false and the abducibles).

```
1.  :- table p_ab/2, q_ab/2.
2.  p_ab(E) :- q([a],E).     q_ab(E) :- p([],E).
3.  p(I,O) :- p_ab(E),produce(O,I,E).
4.  q(I,O) :- q_ab(E),produce(O,I,E).
5.  p_st(I,O,CWA) :- p_st_1(I,O,CWA).
6.  p_st_1(I,O,CWA) :- not_q(I,O,[not p|CWA]).
7.  p_st_1(I,O,_) :- not_a(I,O).
8.  q_st(I,O,CWA) :- q_st_1(I,O,CWA).
9.  q_st_1(I,O,CWA) :- not_p(I,O,[not q|CWA]).
10. not_p(I,I,CWA) :- member(not p,CWA), !.
11. not_p(I,O,CWA) :- tnot(p_ab([])),p_st(I,O,CWA).
12. not_q(I,I,CWA) :- member(not q,CWA), !.
13. not_q(I,O,CWA) :- tnot(q_ab([])),q_st(I,O,CWA).
```

The list is only updated in the second layer of dual rules, which are essentially the rules for negative goals (cf. lines 6 and 9). The update is done by adding the negative goal (without any context) into the CWA list of the negative subgoal in the body. For example, in case of p_st_1 (line 6), not_p is added into the CWA list of the subgoal not_q. Note that in line 10, another rule of not_p is added (similarly in line 12, for not_q) to detect PLoN, by membership testing, i.e., whether we are returning to the same call of not_p. In that case, the output context is equal to the input context. By placing this additional rule before the other not_p rule (line 11), we anticipate the loop by immediately succeeding it and, using cut, to prevent the call to the next not_p (which would lead to looping).

*2) **Negative Loops over Negation (NLoN)**:* XSB with its Well-Founded Semantics and tabling mechanism is aware of negative loops over negation (NLoN) and makes predicates involved in such loops undefined. Consider the ground program P_3 below:

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

where p and q are tabled predicates and, written in XSB, the tabled negation $tnot/1$ is used instead of $not/1$ to so indicate. In this example, p and q (also their default negations) are undefined. The CWA lists previously introduced are able to detect PLoN, but not NLoN. Query p, for example, with respect to the transformation (in the presence of CWA lists) will fail, instead of being undefined. It fails, because the tabled predicate p_ab is involved in a direct positive loop through the call of not_p and not_q. More precisely, whereas in the source program q is defined by the negative subgoal $not\ p$, in the resulting transformation q_ab is defined by the positive subgoal not_p. Hence, 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 so as to keep its truth value; thereby creating NLoN (instead of direct positive loops), but also preserving the semantics of the rule. Apart from other usual predicates produced by the transformation, the new definition of q_ab is as follows:

```
1. :- table q_ab/1, over/1, not_p/1, p_st/3.
2. q_ab(E) :- tnot p_ab([]),not_p_ab([],E).
3. not_p_ab(I,O) :- call_tv(tnot over(not_p(I)),V),
        (V = undefined, O = I, undefined;
         inspect(p_st(I,O,[]))).
4. not_p(I) :- p_st(I,O,[]).
```

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) :- tnot(G)`. The double-wrapping is called through an auxiliary predicate $not\_p\_ab/2$. The XSB system predicate $call\_tv/2$ calls the double-wrapping and unifies V with its truth value (*true* or *undefined*). The value of the output context O then depends on V's value: it is equal to the input context I when NLoN exists (i.e., V is undefined) or O's value is inspected from the tabled predicate p_st by means of predicate $inspect/1$, in case NLoN does not exist (i.e., V is true). The predicate $inspect/1$ can be defined, in XSB, using the combination of its table inspection predicates $get\_calls/3$ and $get\_returns/2$.

It is tempting to use the existing not_p(*I,O,CWA*) in the double-wrapping. Unfortunately, it would cause the call to $over/1$ to flounder, because the output context O is still uninstantiated; hence $not\_p/1$ is introduced instead, free from the output context. Note that in $not\_p/1$ we also omit the CWA context, because the call is made from q_ab, which actually has an empty CWA context (recall that the CWA context is only relevant for dual rules). Indeed, in the definition of $not\_p/1$ (line 4), where it is defined by the dual rule $p\_st/3$, the CWA context of p_st is empty. Its definition is similar to the $not\_p/3$ definition, except that the quick-

 ISBN: 978-1-61208-230-1

kill $tnot\ p\_ab([])$ is moved into the q_ab definition (line 2). That is, it prevents the quick-kill to be wrapped in the $tnot$. Moreover, p_st is tabled, so that its output context O, which is computed when $not\_p/1$ is evaluated in the double-wrapping, can later be reused, via $inspect/1$.

D. Programs with Variables

The problem gets more interesting when we have variables in the program and we consider non-ground queries. Consider program P_4.

$$abds([a/1]). \quad \leftarrow q(X), r(X). \quad q(1). \quad r(X) \leftarrow a(X).$$

We shall discuss how to adapt the construction of dual rules involving predicates with variables. Recall that the IC in P_4 is transformed like any other rule. Instead of only placing a negated literal in the body of a (second layer) dual rule, we are also going to keep all positive non-abducible literals of the original rule that appear before this negated literal. For example, the second dual rule $false\_st\_1$ of the IC in P_4 is now defined by the negated literal $not\_r/4$ and also by all positive non-abducible literals that appear before this negated literal in the original rule, in this case $q/3$, as shown below.

```
false_st(I,O,CWA) :- false_st_1(I,O,CWA).
false_st_1(I,O,CWA) :- not_q(X,I,O,CWA).
false_st_1(I,O,CWA) :- q(X,I,T),
       not_r(X,T,O,[not false|CWA])).
```

The idea to keep these literals before the negated literal in the body of dual rules is to provide an opportunity for the negated literal to be ground when it is called. For example, $not\_r/4$ in the transformation of P_4 can be made ground because X is instantiated when $q/3$ is called, i.e., $X = 1$, like the case in the original rule. This avoids floundering when $tnot(r\_ab(X, []))$ is called through $not\_r/4$ due to the uninstantiated X. Query $q(1)$, for example, will now correctly return the abductive solution $[not\ a(1)]$.

There are some points to remark on regarding this refinement. First, the newly introduced positive literals should be tabulated to avoid duplication of their derivations. Second, one could also introduce all other positive literals in the rule originating the 'not'. This may help produce additional grounding, though, in going against the left-to-right pragmatics coded by the programmer it may create inefficiencies of its own. Third, the semantics doesn't change because the conditions for failure of the positive 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 dual cases. Finally, knowledge of argument types ($+$, $-$, ?), and of shared variables in the body and whether they are local or not, would help refine the transformation to avoid introducing positive literals not contributing to further grounding.

In yet another grounding refinement, TABDUAL also allows us also to deal, in the quick-kill rules, with non-ground positive goals. For example, query $q(X)$ gives the abductive solution $[(not\ a(1)]$ for $X = 1$. But, non-ground negative goals, like $not\ q(X)$ flounders due to the uninstantiated X in the quick-kill $tnot(q\_ab(X, []))$. To resolve the problem, we may restrict the applicability of the quick-kill to ground negative goals only; non-ground negative goals may immediately call the $q\_st/4$ rule without calling the quick-kill. This leads to an improved transformation that allows us to have non-ground negative goals without a $tnot/1$ floundering error. The further refinement of the transformation with respect to the negated rules $not\_q/4$ of P_4 is shown below ($not\_r/4$ is also treated similarly):

```
not_q(X,I,O,CWA) :-
     (ground(not_q(X)),tnot(q_ab(X,[]));
      \+ground(not_q(X))), q_st(X,I,O,CWA).
```

The following example will take us to the actual final transformation. Consider program P_5.

$$abds([a/1]). \quad p(X) \leftarrow q(X). \quad p(1) \leftarrow a(1).$$
$$q(X) \leftarrow p(X). \quad q(2) \leftarrow a(2).$$

Query $p(X)$ to program P_5 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: the only solution returned is $[not\ a(1)]$ for a particular $X = 1$, instead of the expected $[not\ a(1), not\ a(2)]$ for any instantiation of X. The culprit is in the p_st definition used to answer the query:

```
p_st(Y,I,O,CWA) :- [Y] = [X], [Y] = [1],
                   p_st_1(Y,I,T,CWA),
                   p_st_2(Y,T,O,CWA).
```

Note that the p_st rule above is obtained by unifying the argument Y (from the call) with the argument of each p rule, i.e., X and 1 from the first and second p rule respectively, and then failing both rules by the subgoals $p\_st\_1/4$ and $p\_st\_2/4$. In general, four p_st rules from P_5 can be obtained to fail both p rules, by considering the unification of the call argument Y with the arguments X and 1 of the two p rule heads. But in practice it can be minimized, by removing unnecessary unifications, as explained in Section III-E.

When the goal $not\_p(X, [], O, [])$ is launched, the variable Y in both p_st_1 and p_st_2 is instantiated with 1. But recall that p_st_1 and p_st_2 are derived from two different p rules, hence failing p should be achieved by calling p_st_1 and p_st_2 independently. In other words, different variants of the calling argument Y should be used in the call of p_st_1 and p_st_2, as shown below:

```
p_st(Y,I,O,CWA) :-  variant([Y],[Y1]), [Y1] = [X],
                    variant([Y],[Y2]), [Y2] = [1],
                    p_st_1(Y1,I,T,CWA),
                    p_st_2(Y2,T,O,CWA).
```

Now the call of p_st_1 and p_st_2 are independent through the use of different variants $Y1$ and $Y2$, respectively. Note

 ISBN: 978-1-61208-230-1

we are just looking for abductive solutions for failing a calling goal, not for constraints on its free calling variables.

E. TABDUAL Implementation

We have made an implementation [20] on the basis of the TABDUAL specification [7] and have tested it under the current XSB Prolog version 3.3.6. Apart from the '$tnot$ quick-kill' optimization and the relying of the loop detection as much as possible on the tabling mechanism of XSB, we pushed some optimizations further in the implementation:

- We reorder literals in the body of the rules by prioritizing abducibles to come first in the body. This gives an advantage that the second layer dual rules are first defined by abducibles, if they exist. In this way, abductive solutions for queries with negative goals can be obtained faster, for it is easy to incorporate negated abducibles.
- We simplify the second layer dual rules by removing unnecessary unifications, i.e., those that succeed or fail independently of the instantiation of arguments from the calls. For example, in the refined definition of p_st, from the last paragraph of Section III-D, the unification $[Y1] = [X]$ can be removed because unifying with variables always succeeds. Moreover, other p_st alternative rules with the body containing $[Y] \neq [X]$ can also be removed (these rules always fail).
- We split the list of abductive solutions into positive and negative parts. Thus, checking consistency of abductive solutions, in predicates $insert/3$ and $produce/3$, can be done faster.

In the next section we discuss some ideas on how to migrate key ingredients of tabled abduction into an engine-level of logic programming systems, but they can also be appropriated by other systems.

IV. Discussion

Our high level specification design 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. We sketch some ideas on how to migrate key constructs of TABDUAL into an engine-level, say like XSB.

1) ***Hiding Data Structures****:* The CWA list (and attending operations), which is being deployed at the object language level, should migrate to the engine level, even disappearing from the generated code. New operations are needed concerning loop detection, in particular making positive loops on negation 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 object language level will help improve efficiency.

2) ***Tabling Abduction Entries****:* This is the core feature of our 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 improved reuse, because the input abduction table entries are not there. Reuse and consistency are done at the language level, above the tabling level one.

A new tabling mechanism could instead cater for 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 data structure representation consistent with the operations on them and the backtracking mechanism.

V. Conclusion and Future Work

We have addressed the issue of tabling abductive solutions, in a way that they can be reused in abductive contexts different from those in which they were produced. We do so by resorting to a program transformation approach, resulting in a tabled abduction implemented system, TABDUAL. It makes use of the dual program technique, whereby 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 techniques are employed to make the approach correct and more efficient, and to bring it closer to the engine level. Thence, the XSB System (and other Logic Programming systems affording tabled negation) can migrate to their inwards what is best done therein.

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

Future work will consist in perfecting this implementation approach to abduction, and liaising it seamlessly with the underlying engine. That in turn will permit the continued exploration of our applications of abduction and provide feedback for system improvement. For more applications of LP abduction consult our home pages, and references therein.

And a future conceptual application area in the context of abduction in logic, 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

 ISBN: 978-1-61208-230-1

explanations found for the primary observation. In other words, whether the secondary observations are plausible in the abductive context of the primary one, a quite common scientific reasoning task.

ACKNOWLEDGMENT

Ari Saptawijaya acknowledges the support from FCT-Portugal, grant SFRH/BD/72795/2010. We thank David Warren for his comments on prior drafts and corresponding implementation.

REFERENCES

[1] A. C. Kakas and A. Michael, "An abductive-based scheduler for air-crew assignment," *J. of Applied Artificial Intelligence*, vol. 15, no. 1-3, pp. 333–360, 2001.

[2] J. F. Castro and L. M. Pereira, "Abductive validation of a power-grid expert system diagnoser," in *Procs. 17th Intl. Conf. on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA-AIE'04)*, 2004.

[3] J. Gartner, T. Swift, A. Tien, C. V. Damásio, and L. M. Pereira, "Psychiatric diagnosis from the viewpoint of computational logic," in *7th Intl.Conf. on Principles of Knowledge Representation and Reasoning, NMR ws on Abductive Reasoning*, 2000.

[4] R. Kowalski and F. Sadri, "Abductive logic programming agents with destructive databases," *Annals of Mathematics and Artificial Intelligence*, vol. 62, no. 1, pp. 129–158, 2011.

[5] A. V. Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs." *J. of ACM*, vol. 38, no. 3, pp. 620–650, 1991.

[6] 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*, vol. 4, no. 4, pp. 383–428, 2004.

[7] L. M. Pereira and A. Saptawijaya, "Appendix: Abductive logic programming with tabled abduction," https://dl.dropbox.com/u/47496395/appendix_icsea12.pdf, retrieved: September, 2012.

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

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

[10] 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.

[11] T. Eiter, G. Gottlob, and N. Leone, "Abduction from logic programs: semantics and complexity," *Theoretical Computer Science*, vol. 189, no. 1-2, pp. 129–177, 1997.

[12] K. Inoue and C. Sakama, "A fixpoint characterization of abductive logic programs," *J. of Logic Programming*, vol. 27, no. 2, pp. 107–136, 1996.

[13] R. Kowalski, *Computational Logic and Human Thinking: How to be Artificially Intelligent*. Cambridge U. P., 2011.

[14] "ABDUAL System," http://www.cs.sunysb.edu/~tswift/interpreters.html, retrieved: September, 2012.

[15] 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, retrieved: September, 2012.

[16] "NegABDUAL System," http://centria.di.fct.unl.pt/~lmp/software/contrNeg.rar, retrieved: September, 2012.

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

[18] "XSB Prolog," http://xsb.sourceforge.net/, retrieved: September, 2012.

[19] T. Swift and D. S. Warren, "XSB: Extending Prolog with tabled logic programming," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 157–187, 2012.

[20] "TABDUAL System," https://dl.dropbox.com/u/47496395/tabdual_icsea12.zip, retrieved: September, 2012.

 ISBN: 978-1-61208-230-1