Implementing Tabled Abduction in Logic Programs

Ari Saptawijaya[⋆] and Luís Moniz Pereira (Supervisor)

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 it can be too difficult to implement, and costly to perform, in particular if abductive solutions are not tabled. If they become tabled, then abductive solutions can be reused, even from one abductive context to another. 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. We recently published a conception of tabled abduction with its prototype, TABDUAL, implemented in XSB Prolog. We detail here subsequent progress that has been made on the implementation aspect of TABDUAL, towards its more practical use.

Keywords: tabled abduction, abductive logic programming, XSB Prolog.
Dates: beginning October 2011; foreseen conclusion August 2014.

1 Introduction

Abduction has been well studied in the field of computational logic, and logic programming in particular, for a few decades by now [3, 5, 6, 11]. Abduction in logic programs offers a formalism to declaratively express problems in a variety of areas and it has many applications, e.g. in decision-making, diagnosis, planning, belief revision, and hypothetical reasoning [4, 7, 12–14]. On the other hand, many Prolog systems have become mature, and it thus makes sense to facilitate the use of abduction in them.

In abduction, finding some best explanations (i.e. 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 a context are relevant in a different context, and thus 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 is conceptually suitable for abduction, to deal with the reuse of abductive solutions. In practice, abductive solutions reuse is not immediately amenable to tabling, because now solutions go together with an abductive context. It also poses a new problem on how to reuse them in a different but compatible context, while catering as well to all varieties of loops in logic programs, now complicated by abduction.

We recently introduced a concept of tabled abduction in abductive normal logic programs, and its prototype TABDUAL, implemented using XSB-Prolog [2], to address the above issues [20]. It is realized via a program transformation, where abduction is subsequently enacted on the transformed program. The transformation relies on the theory

[⋆] Affiliated with Fakultas Ilmu Komputer at Universitas Indonesia, Depok, Indonesia.

of the dual transformation [3], which allows to more efficiently handle the problem of abduction under negative goals, by introducing their positive dual counterparts.

We report here on additional significant conceptual and technical progress of the TABDUAL development towards its more practical use, which might be taken up by other Prolog systems that afford tabling. First, we address the issue of heavy transformation load due to producing *complete* dual rules in advance of use. A natural solution is 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: tabling all dual rules as they are created for a predicate or, in contrast, lazily generating and storing (instead of tabling) those dual rules in a trie, as new alternatives are required. The former approach leads to an eager dual by-need transformation, whereas the latter permits a lazy one. Second, we enhance TABDUAL's flexibility by permitting modular mixes of abductive and non-abductive parts. The non-abductive part allows for simpler treatment of facts in programs, avoiding their superfluous transformation, which would hinder the use of large factual data. Third, we introduce a new TABDUAL system predicate that allows accessing ongoing abductive solutions for dynamic manipulations, e.g. exercising preferences on abducibles.

The paper is structured as follows. Section 2 reviews basic notions of logic programming and abduction in logic programs. The concept of tabled abduction is then discussed, in Section 3. Section 4 details the progress made in the implementation of TABDUAL. We broach related and future work, in Section 5, and conclude in Section 6.

2 Abduction in Logic Programs

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, or inference to the best explanation (a common designation of one of its uses in the philosophy of science [10,16]), is a reasoning method, whereby one chooses those hypotheses that would, if true, best explain the observed evidence – while meeting any attending ICs – or that would satisfy some query. In LPs, abductive hypotheses (or *abducibles*) are named literals of the program which have no rules, and whose truth value is not initially assumed. Abducibles may have arguments, but they must be ground on the occasion of their abduction. An *abductive normal logic program* is a normal logic program that allows for abducibles to appear in the body of rules. Note that the 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, in order to produce an abductive solution to a query, that is a consistent set of assumed hypotheses that support it. An *abductive solution* to a query is a consistent set of abducible instances or their negations 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 procedure is possible only when the underlying semantics is relevant, i.e. avoids computing a whole model 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) [8] enjoys the relevance property, and thus permits abduction to be performed by need, as induced by the top-down query-oriented procedure. That is, it finds solely the relevant abducibles needed by the query; the value of abducibles not mentioned in the abductive solution obtained is indifferent to the query, assuming the ICs are satisfied. Our prototype of tabled abduction TABDUAL, implemented in XSB Prolog, is underpinned by WFS with abduction theory [3]. Though WFS is three-valued, the abduction mechanism in TABDUAL enforces, by design, two-valued abductive solutions: needed abducibles are assumed either true or false.

3 Tabled Abduction in Logic Programs

We start with the motivation for tabled abduction and subsequently show how tabled abduction is conceptualized and realized in the TABDUAL transformation.

3.1 Motivation

Example 1. Consider program P_1: $q \leftarrow a.$ $s \leftarrow b, q.$ $t \leftarrow s, q.$
where a and b are abducibles. 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 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 an abductive context of q (i.e. $[b]$ in the second query, s)

to another 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, and (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 employing the very idea of tabling and of reusing abductive solutions, the dual transformation to deal with abduction under negative goals, and how abducibles and queries are transformed. Further constructs of the transformation, i.e. to deal with loops in a program and with programs containing variables, which are not the focus in the present paper, are fully detailed in [22].

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_1 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_1 is transformed. It is transformed into two rules:

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

Predicate $t_{ab}(E)$ is the tabled predicate which is introduced to table the abductive solution of t in its argument E. Its definition, in the left rule, 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 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 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 in a given (input) abductive context of t. This rules expresses that the output abductive solution O of t is obtained from the 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 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.

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. 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}.

Abduction under Negative Goals For abducing under negative goals, the program transformation employs the *dual transformation* [3], which makes negative goals 'positive', thus permitting to avoid the computation of all abductive solutions, and then negating them, under the otherwise regular negative goals. Using the dual transformation, we are able to obtain one abductive solution at a time, as when we treat abduction under positive goals. The dual transformation defines for each atom A and its set of rules R in a 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' one, not_A. Example 3 illustrates how the dual transformation is employed in the TABDUAL transformation.

Example 3. Consider program P_2: $\quad p \leftarrow a. \quad p \leftarrow q, not\ r. \quad r.$
where a is an abducible.
(A) 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:

$$not\_p(I, O) \leftarrow p^{*1}(I, T), p^{*2}(T, O).$$

In the TABDUAL transformation, this rule is known as the first layer of the dual transformation. Notice the addition of the input and output abductive context arguments, I and O, in the head, and similarly in each subgoal of the rule's body, where the intermediate context T is used to relay the 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 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 failing one subgoal in its body at a time, i.e. by negating q or, alternatively, by negating $not\ r$.

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

(B) 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. Being a fact, the content of the context I is just 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 only on the input context.
(C) With regard to r, since it is a fact, its dual $not\_r(\_, \_) \leftarrow fail$ may be introduced or simply omitted in the transformed program.

In TABDUAL, every abducible (and its negation) is transformed into a rule, which effectively updates the given abductive context with the abducible (or its negation, respectively). For instance, the abducible a from Example 3 is transformed into:

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

where $insert(A, I, O)$ is a TABDUAL system predicate which inserts abducible A into the input context I, resulting in the output context O. During the insertion, it maintains the consistency of the context. The negation $not\ a$ of the abducible $a/1$ is transformed similarly, where $not\ a$ is renamed into not_a in the head:

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

Finally, a query to a program, consequently, must be transformed too: (1) A positive goal G is simply augmented with the two extra arguments for the input and output abductive contexts, and (2) A negative goal $not\ G$ is made 'positive', not_G, in addition to the two extra input and output context arguments. A query should additionally ensure that all ICs are satisfied. With regard to ICs, when there is no IC defined in a program, then, following the dual transformation, fact $not\_false(I, I)$ is added to the transformed program. Otherwise, ICs, which essentially are rules with *false* in their heads, are transformed just like any other rules.

Example 4. Query $not\ p$ is first transformed into $not\_p(I, O)$. Then, to satisfy all ICs, it is conjoined with $not\_false/2$, resulting in its complete call, as a top goal:

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

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.

4 Further Implementing Tabled Abduction in Logic Programs

The core TABDUAL transformation has been implemented in XSB Prolog, and its improvement is continuing. Next, we discuss the further implementation aspect of TABDUAL, addressing progress that has been made to improve its earlier version in dealing with the dual transformation and handling programs with large factual data. Moreover, we enhance TABDUAL's flexibility with a feature to access ongoing abductive solutions for dynamic manipulation.

4.1 By-need Dual Transformation

The early version of TABDUAL performs a naive complete dual transformation, i.e. it produces *all* (second layer) dual rules, in advance, for every defined atom in an input program. This should be avoided in practice, as potentially massive dual rules and large sets of them 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, the transformation may suffer from a heavy load and may take ages. It hinders the subsequent abduction phase to take place next, not to mention the compile time of the thus produced large 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 first layer 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 definitions of the second layer.

Example 5. Recall Example 3. The by-need dual transformation contains the same first layer: $not\_p(I,O) \leftarrow p^{*1}(I,T), p^{*2}(T,O)$. The second layer now contains, for each $i \in \{1,2\}$: $p^{*i}(I,O) \leftarrow dual(i,p,I,O)$.

The newly introduced system predicate $dual/4$ facilitates the by-need construction of generic dual rules (i.e. without any context attached to them) from the i-th rule of $p/1$, during abduction. It will also instantiate the generic dual rules with the provided arguments and contexts, and subsequently invoke the instantiated dual rules.

Extra computation load that may occur during the abduction phase, due to the by-need construction of dual rules, 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 discuss two approaches for memoizing generic dual rules; each approach influences how generic dual rules are constructed.

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 (in case P is a non-nullary predicate) and the input context I and 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 dual rules 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 3, 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. This is a bit against the spirit of a full by-need dual rules construction, where only one alternative dual rule is constructed at a time, i.e. generic dual rules are to 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 [26]. 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 by-need dual rules construction, this means $dual\_rule/3$ will 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 via a new XSB installation, i.e. one XSB instance for one scheduling strategy, which is inconvenient for general purpose, we shall consider another approach to implement a lazy dual rules construction.

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 can be 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 literal used in constructing the latest 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):

$$\begin{array}{lll}
1. & dual(N, P, I, O) & \leftarrow trie\_property(T, alias(dual)), dual(T, N, P, I, O). \\
2a. & dual(T, N, P, I, O) & \leftarrow trie\_interned(d(N, P, Dual, \_), T), \\
 & & \quad call\_dual(P, I, O, Dual). \\
2b. & dual(T, N, P, I, O) & \leftarrow current\_pos(T, N, P, Pos), \\
 & & \quad dualize(Pos, Dual, NextPos), \\
 & & \quad store\_dual(T, N, P, Dual, NextPos), \\
 & & \quad call\_dual(P, I, O, Dual).
\end{array}$$

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 $2a$ and $2b$ give the definition of $dual/5$. In the first definition (line $2a$), 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 $d(N, P, Dual, Lits)$, 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 $2b$) 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 $dualize/3$. It 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 approach 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 overall local table scheduling.

4.2 Transforming Facts

TABDUAL by its specification transforms facts as any other rules in the program. For instance, fact p(1) will be completely transformed into:

$$p_{ab}(1, [\,]). \qquad\qquad p(X, I, O) \leftarrow p_{ab}(X, E), produce(O, I, E).$$
$$not\_p(X, I, O) \leftarrow p^*(X, I, O). \qquad p^*(X, I, I) \leftarrow X \neq 1.$$
Similar transformed rules are produced for all other facts on $p/1$, e.g. $p(2), p(3), \ldots$. 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 $p/1$, comprised of just facts, can be much more simply transformed. The transformed rules $p_{ab}/2$ and $p/3$ can be substituted by a single rule:
$$p(X, I, I) \leftarrow p(X).$$
and their negations, rather than using dual rules, can be transformed to a single rule:
$$not\_p(X, I, I) \leftarrow not\ p(X).$$
independently of the number of facts are there for $p/1$. Note that the input and output context arguments are added in the head, and the input context is just passed intact to the output one. All facts of predicate $p/1$ can be defined in the non-abductive part of the input program, thus allowing modular mixes of abductive and non-abductive parts. The non-abductive part is indicated by the $beginProlog$ and $endProlog$ identifiers and any program between them will not be transformed, i.e. it is treated as a usual Prolog program. For facts $p(1)$, $p(2)$, and $p(3)$, we simply list them as follows:
$$beginProlog. \qquad p(1). \qquad p(2). \qquad p(3). \qquad endProlog.$$
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 the abduction time itself.

4.3 Accessing 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 further, 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 a new system predicate $abdQ(P)$ that allows to access the ongoing abductive solution and to manipulate it using the rules of P. This system predicate is transformed by unwrapping it and adding an extra argument to P for the ongoing abductive solution.

Example 6. Suppose program P_3 contains the rules:
$$q \leftarrow r, abdQ(s), t. \qquad\qquad s(X) \leftarrow v(X).$$
along with some other rules. Note that, though predicate s within system predicate wrapper $abdQ/1$ has no argument, its rule definition has one extra argument for the ongoing abductive solution. The tabled predicate q_{ab} in the transformed program would be $q_{ab}(E) \leftarrow r([\,], T_1), \mathbf{s}(T_1, T_1, T_2), t(T_2, E)$. That is, $s/3$ now gets access to the ongoing

abductive solution T_1 from $r/2$, via its additional first argument. It still has the usual input and output contexts, T_1 and T_2, respectively, in its second and third arguments. Rule $s/1$ in P_3 is transformed like any other rules.

The new system predicate $abdQ/1$ also 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 $s(X) \leftarrow prolog(\mathit{preferred}(X))$, where *preferred*$(X)$ defines, in the non-abductive program part, some preference rule on a given solution X. The predicate wrapper $prolog/1$ is a general TABDUAL system predicate employed to execute Prolog calls to predicates, in its argument, that are not transformed by TABDUAL (e.g. Prolog built-in predicates, or those defined in the non-abductive program part, like *preferred*(X) in this example).

5 Discussion

TABDUAL deals additionally with a variety of loops in abductive normal logic programs as well as programs with variables; the details can be found in [20]. There have been a plethora of work on abduction in logic programming, cf. [5,11] for a survey on this line of work. But, with the exception of ABDUAL [3], we are not aware of any other efforts that have addressed the use of tabling in abduction for abductive normal logic programs, which may be complicated with loops. Like ABDUAL, we use the dual transformation and rely on the same theoretic underpinnings, but ABDUAL does not allow variables in rules. Tabling also has only been employed there limitedly, i.e. to table its meta-intepreter, which in turn allows abduction to be performed in the presence of loops in a program. It does not address at all the issues raised by the desirable reuse of tabled abductive solutions. Tabling has also been used in the context of statistical abduction [21,25], but it concerns itself with probabilistic logic programs, whereas TABDUAL with abductive normal logic programs.

We have evaluated TABDUAL in practice, detailed elsewhere [23], using examples from constraint satisfaction and declarative debugging problems [17, 18], to better understand its performance and scalability with respect to its main features. The result is promising, that TABDUAL indeed benefits from tabling ongoing abductive solutions: as constraints become more complex, its performance consistently surpasses that of its non-tabling counterpart. Other work (cf. [24]) details the preliminary use of TABDUAL in decision making.

Our preliminary experiment to compare the eager and the lazy by-need dual transformation yields that the lazy computation returns the first solution much faster than the eager one. When all solutions are aggregated (e.g. using predicate *findall*), the eager one performs slightly better than the lazy one. But 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, or is interesting enough, and may continue searching for more if needed, just like Prolog does. In that case, it seems reasonable that the lazy computation may be competitive with the eager one. Nevertheless, the two approaches may become options of TABDUAL customization. A more thorough comparison between the two approaches is earmarked for future work. We shall also evaluate TABDUAL in real-world applications, e.g. in modeling cell signaling and the effect of external drugs on them in the context of chemoprevention [15], in which we may compare TABDUAL with the new implementation of the A-system [1] used therein.

TABDUAL is part and parcel of our research plan: in the near future it will be refined, integrated with other features (e.g. program updates, side-effect inspection, uncertainty), and employed for moral reasoning; a field which has recently gained attention and a resurgence of interest from AI community, and on which we work [9, 19].

6 Conclusion

We have argued the need for tabled abduction, in a way that abductive solutions can be reused, even from one abductive context to another. We detailed how it is conceptualized and realized in TABDUAL using a program transformation. The core transformation employs the dual transformation for empowering abduction under negative goals, which allows obtaining one abductive solution at a time, instead of computing all alternative abductive solutions and negating their disjunction.

We also discussed some progress made on improving TABDUAL towards its more practical use. First, we fostered two approaches for constructing dual rules by need, the eager and the lazy ones. Second, the transformation for a predicate comprised of just facts is made much simpler, which also allows modular mixes between abductive and non-abductive parts. Third, accessing ongoing abductive solutions are made possible in order to dynamically manipulate them.

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. We hope this work will lead, in particular, to an XSB System that can provide its users with specifically tailored tabled abduction facilities.

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. A-system v2. `http://www-dse.doc.ic.ac.uk/cgi-bin/moin.cgi/abduction`.
2. XSB Prolog. `http://xsb.sourceforge.net/`.
3. 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.
4. 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)*, volume 3029 of *LNAI*, pages 838–847. Springer, 2004.
5. M. Denecker and A. C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*. Springer Verlag, 2002.

6. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.
7. 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.
8. 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.
9. 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.
10. J. R. Josephson and S. G. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge U. P., 1995.
11. 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.
12. 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.
13. R. Kowalski. *Computational Logic and Human Thinking: How to be Artificially Intelligent*. Cambridge U. P., 2011.
14. R. Kowalski and F. Sadri. Abductive logic programming agents with destructive databases. *Annals of Mathematics and Artificial Intelligence*, 62(1):129–158, 2011.
15. S. Lazarou, A. C. Kakas, C. Neophytou, and A. Constantinou. Logical modeling of cancer and chemoprevention. In *Procs. Workshop on Learning and Discovery in Symbolic Systems Biology (LDSSB)*, 2012.
16. P. Lipton. *Inference to the Best Explanation*. Routledge, 2001.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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.
22. A. Saptawijaya and L. M. Pereira. Tabled abduction in logic programs. Accepted as Technical Communication at ICLP 2013. Available at `http://centria.di.fct.unl.pt/~lmp/publications/online-papers/tabdual_lp.pdf`, 2013.
23. A. Saptawijaya and L. M. Pereira. Towards practical tabled abduction in logic programs. In *16th Portuguese Conference on Artificial Intelligence (EPIA)*, LNAI. Springer, 2013.
24. 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.
25. T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *J. of Artificial Intelligence Research (JAIR)*, 15:391–454, 2001.
26. 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.