Towards Practical 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. Despite its potential as a reasoning paradigm in AI applications, 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. 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. The concept of tabled abduction has been realized recently in an abductive logic programming system TABDUAL. Besides tabling abductive solutions, TABDUAL also relies on the dual transformation. In this paper, we emphasize two TABDUAL improvements: (1) the dual transformation *by need*, and (2) a new construct for accessing ongoing abductive solutions, that permits modular mixes between abductive and non-abductive program parts. We apply subsequently these improvements on two distinct problems, and evaluate the performance and the scalability of TABDUAL on several benchmarks on the basis of these problems, by examining four TABDUAL variants.

Keywords: tabled abduction, abductive logic programming, tabled logic programming, dual transformation.

1 Introduction

Abduction has already been well studied in the field of computational logic, and logic programming in particular, for a few decades by now [1, 3, 4, 8]. Abduction in logic programs offers a formalism to declaratively express problems in a variety of areas, e.g. in decision-making, diagnosis, planning, belief revision, and hypothetical reasoning (cf. [2, 5, 9–11]). On the other hand, many Prolog systems have become mature and practical, and thus it makes sense to facilitate the use of abduction into such systems.

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

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

tabling, because such solutions go together with an ongoing abductive context. It also poses a new problem on how to reuse them in a different but compatible context, while catering as well to loops in logic programs, i.e. positive loops and loops over negation, now complicated by abduction.

A concept of tabled abduction in abductive normal logic programs, and its prototype TABDUAL, to address the above issues, was recently introduced [16]. It is realized via a program transformation, where abduction is subsequently enacted on the transformed program. The transformation relies on the theory of the dual transformation [1], which allows one to more efficiently handle the problem of abduction under negative goals, by introducing their positive dual counterparts. We review TABDUAL in Section 2.

While TABDUAL successfully addresses abductive solution reuse, even in different contexts, and also correctly deals with loops in programs, it may suffer from a heavy transformation load due to performing the *complete* dual transformation of an input program in advance. Such heavy transformation clearly may hinder its practical use in real world problems. In the current work, we contribute by enacting a *by-need* dual transformation, i.e. dual rules are only created as they are needed during abduction, either eagerly or lazily – the two approaches we shall examine. We furthermore enhance TABDUAL's flexibility by introducing a new system predicate to access ongoing abductive solutions, thereby permitting modular mixes of abductive and non-abductive program parts. These and other improvements are detailed in Section 3.

Until now there has been no evaluation of TABDUAL, both in terms of performance and scalability, as to gauge its suitability for likely applications. In order to understand better the influence of TABDUAL's features on its performance, we separately factor out its important features, resulting in four TABDUAL variants of the same underlying implementation. One evaluation uses the well-known N-queens problem, as the problem size can be easily scaled up and in that we can additionally study how tabling of conflicts or nogoods of subproblems influences the performance and the scalability of these variants. The other evaluation is based on an example from declarative debugging, previously characterized as belief revision [13, 14]. The latter evaluation reveals the relative worth of the newly introduced dual transformation by need. We discuss all evaluation results in Section 4, and conclude in Section 5.

2 Tabled Abduction

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. In this work 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.

2.1 Abduction in Logic Programs

Let us recall that abduction, or inference to the best explanation (a designation common in the philosophy of science [7, 12]), is a reasoning method, whence one chooses hypotheses that would, if true, best explain observed evidence – whilst meeting any prescribed ICs – or that would satisfy some query. In LPs, abductive hypotheses (or *abducibles*) are named literals of the program having no rules, whose truth value is not assumed initially. Abducibles can have parameters, but must be ground when abduced. An *abductive normal logic program* is a normal logic program that allows for abducibles in the body of rules. Note that the negation '$not\ a$' of an abducible a refers not to its default negation, as abducibles by definition lack any rules, but rather to an assumed hypothetical negation of a.

The truth value of abucibles may be distinctly assumed *true* or *false*, through either their positive or negated form, as the case may be, as a means to produce an abductive solution to a goal query in the form of a consistent set of assumed hypotheses that lend support to it. An *abductive solution* to a query is thus a consistent set of abducible ground instances or their negations that, when replaced by their assigned truth value everywhere in the program P, provides a model of P (for the specific semantics used on P), satisfying both the query and the ICs – called an *abductive model*. Abduction in LPs can be accomplished naturally by a top-down query-oriented procedure to identify an (abductive) solution by need, i.e. as abducibles are encountered, where the abducibles in the solution are leaves in the procedural query-rooted call-graph, i.e. the graph recursively generated by the procedure calls from the literals in bodies of rules to the heads of rules, and subsequently to the literals in the rule's body.

2.2 Tabled Abduction in TABDUAL

Next, we recall the basics of tabled abduction in TABDUAL. Consider an abductive logic program, taken from [16]:

Example 1. 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 posed, 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 is realized in a prototype TABDUAL, implemented in the most advanced LP tabling system XSB Prolog [19], which involves a program transformation of abductive logic programs. Abduction can then be enacted on the transformed program directly, without the need of a meta-interpreter. Example 1 already 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.

Example 2. The rule $t \leftarrow s, q$ from Example 1 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 tables 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 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 TABDUAL system predicate $produce(O, I, E)$, that checks consistency. The other rules are transformed following the same idea.

An abducible is transformed into a rule that inserts it into the abductive context. For example, the abducible a of Example 1 is transformed into: $a(I, O) \leftarrow insert(a, I, O)$, where $insert(a, I, O)$ is a TABDUAL system predicate which inserts a into the input context I, resulting in the output context O, while also checking consistency. The negation $not\ a$ of the abducible a is transformed similarly, except that it is renamed into not_a in the head: $not\_a(I, O) \leftarrow insert(not\ a, I, O)$.

The TABDUAL program transformation employs the *dual transformation* [1], 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. Instead, 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 only the main idea of how the dual transformation is employed in TABDUAL and omits many details, e.g. checking loops in the input program, all of which are referred in [16].

Example 3. Consider the following program fragment, in which p is defined as:

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

where a is an abducible. The TABDUAL 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:

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

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 earlier. 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, alternatively, by negating $not\ r$.

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

Finally, TABDUAL transforms integrity constraints like any other rules, and top-goal queries are always launched by also satisfying integrity constraints.

3 Improvements on TABDUAL

The number of dual rules for atom A, produced by a naive dual transformation, is generally exponential in the number of A's rules, because all combinations of body literals from the positive rules need to be generated. For complexity result of the core TABDUAL transformation, the reader is referred to [17]. We propound here, for the first time, two approaches of dual transformation by need, as a means to avoid a heavy TABDUAL transformation load due to superfluous dual transformation. We also extend TABDUAL features here with a new system predicate that allows accessing ongoing abductive solutions for dynamic manipulation.

3.1 By-need Dual Transformation

By its very previous specification, TABDUAL performs a *complete* dual transformation for every defined atom in the program in advance, i.e. as an integral part of the whole TABDUAL transformation. This certainly has a drawback, as potentially massive dual rules are created in the transformation, though only a few of them might be invoked during abduction. That is unpractical, as real-world problems typically consist of a huge number of rules, and such a complete dual transformation may hinder abduction to start taking place, not to mention the compile time, and space requirements, of the large thus produced transformed program.

One pragmatic and practical 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. 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 $i \in \{1, 2\}$, rule $p^{*i}(I, O) \leftarrow dual(i, p, I, O)$. 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 thus their recomputation can be avoided. We discuss two approaches for memoizing generic dual rules; each approach influences how generic dual rules are constructed:

– *Tabling generic dual rules*, which results in an *eager* construction of dual rules, due to the local table scheduling 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 [19]. For instance, in Example 3, when $p^{*2}(I, O)$ is invoked, all two alternatives of generic 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 they are subsequently instantiated and invoked. XSB also provides other (in general less efficient) table scheduling alternative, i.e. batched scheduling, which allows constructing only one generic dual rule at a time before it is instantiated and invoked. But the choice between the two scheduling strategies is fixed in each XSB installation, i.e. at present one cannot switch from one to the other without a new installation. Hence, we propose the second approach.
– *Storing generic dual rules in a trie*, which results in a *lazy* construction of dual rules. The trie data structure in XSB allows facts to be stored in a tree representation, and is built from a prefix ordering of each fact; thus, in this case factors out the common prefix of the (stored) facts [20]. In our context, facts are used to represent generic dual rules, which can be directly stored and manipulated in a trie. This approach permits simulating batched scheduling within the local scheduling employed by default in XSB. It hence allows constructing one generic dual rule at a time, before it is instantiated and invoked, and memoizing it explicitly through storing it in the trie for later reuse. By constructing only one generic dual rule at a time, additional information to track through literal's position, used in constructing the latest dual rule, should then be maintained. All these steps are made possible through XSB's trie manipulation predicates.

3.2 Towards a More Flexible TABDUAL

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 abducibles using preferences. But since it is encapsulated in an abductive context, and such 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 4. Consider a program fragment: $q \leftarrow r, abdQ(s), t. \quad s(X) \leftarrow v(X).$ 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 $q_{ab}(E) \leftarrow r([\,], T_1), s(T_1, T_1, T_2), t(T_2, E)$. 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 $abdQ/1$ permits modular mixes of abductive and non-abductive program parts. For instance, the rule of $s/1$ in P_3 may naturally be defined by some non-abductive program parts. Suppose that the argument of s represents the ongoing board configuration for the N-queens problem (i.e. the ongoing abductive solution is some board configuration). Then, the rule of $s/1$ can be instead, $s(X) \leftarrow prolog(safe(X))$, where the $prolog/1$ wrapper is the existing TABDUAL predicate to execute normal Prolog predicates, i.e. those not transformed by TABDUAL. Predicate $safe(X)$ can then be defined in the non-abductive program part, to check whether the ongoing board configuration of queens is momentarily safe.

The treatment of facts in programs may also benefit from modular mixes of abductive and non-abductive parts. As facts do not induce any abduction, a predicate comprised of just facts can be much more simply transformed: only a bridge transformed rule for invoking Prolog facts, is needed, which keeps the output abductive context equal to the input one. The facts are listed, untransformed, in the non-abductive part.

4 Evaluation of TABDUAL

As our benchmarks do not involve loops in their representation, we employ in the evaluation the version of TABDUAL which does not handle loops in programs, such as positive loops (e.g. program $P_1 = \{p \leftarrow p.\}$), or loops over negation (e.g. program $P_2 = \{p \leftarrow not\ q.\ ;\ q \leftarrow p.\}$). Loops handling in TABDUAL and the evaluation pertaining to that are fully discussed in [17]. For better understanding of how TABDUAL's features influence its performance, we consider four distinct TABDUAL variants (of the same underlying implementation), obtained by separately factoring out its important features:

1. TABDUAL-need, i.e. without dual transformation by need,

2. TABDUAL+eager, i.e. with eager dual transformation by need,
3. TABDUAL+lazy, i.e. with lazy dual transformation by need, and
4. TABDUAL+lazy-tab, i.e. TABDUAL+lazy without tabling abductive solutions.

The last variant is accomplished by removing the table declarations of abductive predicates $*^{ab}$, where $*$ is the predicate name (cf. Example 2), in the transformed program.

4.1 Benchmarks

The first benchmark is 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 (for simplicity, we omit some syntactic details):

$$\begin{array}{l} q(0, N). \\ q(M, N) \leftarrow M > 0, q(M-1, N), d(Y), pos(M, Y), not\ abdQ(conflict). \\ conflict(BoardConf) \leftarrow prolog(conflictual(BoardConf)). \end{array}$$

with the query $q(N, N)$ for N queens. Here, $pos/2$ is the abducible predicate 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, i.e. tabling conflictual configurations of queens (essentially, tabling the ongoing abductive solutions), influences scalability of the four variants.

The second benchmark concerns an example of declarative debugging. Our aim with this benchmark is to evaluate the relative worth of the by-need dual transformation in TABDUAL, with respect to both the transformation and the abduction time. Declarative debugging of normal logic programs has been characterized before as belief revision [13, 14], and it has been shown recently that they can also be viewed as abduction [18]. There are two problems of declarative debugging: incorrect solutions and missing solutions. The problem of debugging incorrect solution S is characterized by an IC of the form $\leftarrow S$, whereas debugging missing solution S by an IC of the form $\leftarrow not\ S$. Since with this benchmark we aim at evaluating the by-need dual transformation, we consider only the problem of debugging incorrect solutions (its form of IC perforce constructs and invokes dual rules). The benchmark takes the following program to debug, where the size $n > 1$ can easily be increased:

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

with the IC: $\leftarrow q_m(0)$, for $0 \leq m \leq n$, to debug incorrect solution $q_m(0)$.

4.2 Results

The experiments were run under XSB-Prolog 3.3.6 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.

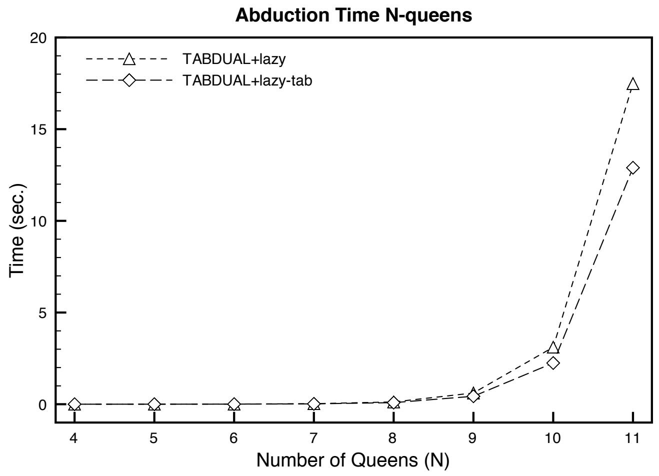

Fig. 1. The abduction time of different N queens.

N-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 1 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?

Figure 2 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.

Declarative Debugging Since our aim with this benchmark is to evaluate the relative worth of the by-need dual transformation, we focus on three variants: TABDUAL-need, TABDUAL+eager, and TABDUAL+lazy. We evaluate the benchmark for $n = 600$, i.e. debugging a program with 1202 rules. After applying the declarative debugging transformation (of incorrect solutions), which results in an abductive logic program, we ap-

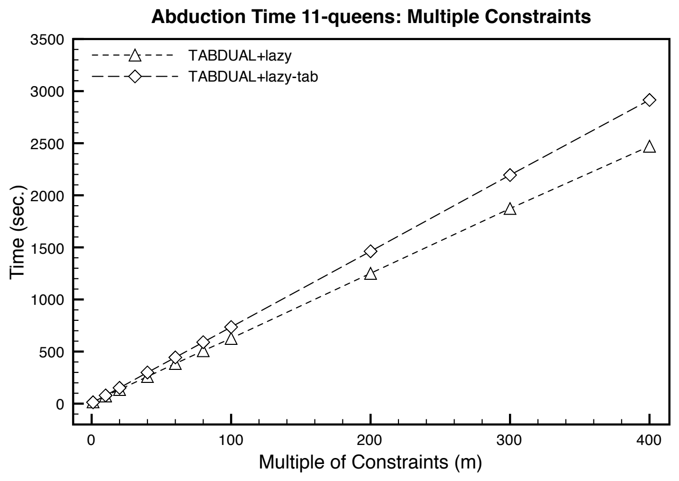

Fig. 2. The abduction time of 11 queens with increasing complexity of conflict constraints.

ply the TABDUAL transformation. The TABDUAL transformation of variants employing the dual transformation by need takes 0.4691 seconds (compiled in 3.3 secs with 1.5 MB of memory in use), whereas TABDUAL-need takes 0.7388 seconds (compiled in 2.6 secs with 2 MB of memory in use). Whereas TABDUAL-need creates 1802 second layer dual rules during the transformation, both TABDUAL+eager and TABDUAL+lazy creates only 601 second layer dual rules in the form similar to that of Example 4. And during abduction, the latter two variants construct only, by need, 60% of the complete second layer dual rules produced by the other variant.

Figure 3 shows how the by-need dual transformation influences the abduction time, where different values of m in the IC: $\leftarrow q_m(0)$ are evaluated consecutively, $100 \leq m \leq 600$; in this way, greater m may reuse generic dual rules constructed earlier by smaller m. We can 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 with the significantly less transformation time: the total (transformation plus abduction) time of TABDUAL-need is 0.8261, whereas TABDUAL+eager and TABDUAL+lazy need are 0.601 and 0.6511, respectively. That is, either dual by-need approach gives an overall better performance than TABDUAL-need.

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. But, TABDUAL+lazy returns the first abductive solution much faster than TABDUAL+eager, e.g. at $m = 600$ the lazy one needs 0.0003 seconds, whereas the eager one 0.0105 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.

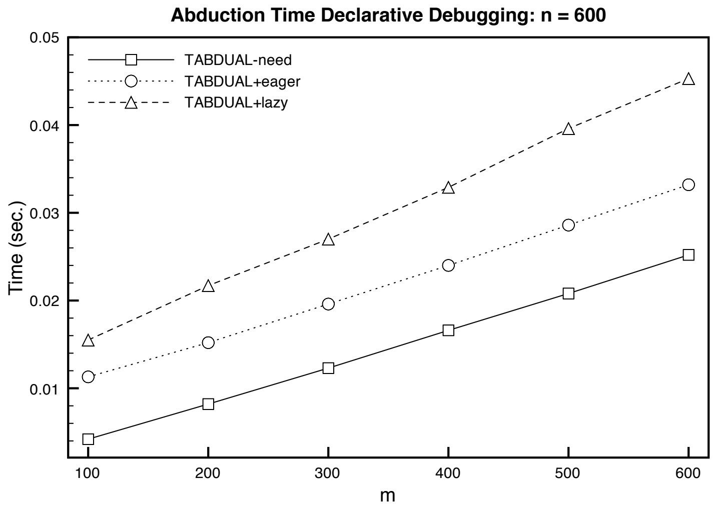

Fig. 3. The abduction time for debugging incorrect solution $q_m(0)$ (with $n = 600$).

5 Conclusion and Future Work

We introduced in TABDUAL two approaches for the dual transformation by need, and enhanced it with a system predicate to access ongoing abductive solutions for dynamic manipulation. Employing these improvements, we evaluated TABDUAL by factoring out their important features and studied its scalability and performance. An issue that we have touched upon the TABDUAL evaluation is tabling nogoods of subproblems (as ongoing abductive solutions) and how it may improve performance and scalability. With respect to our benchmarks, TABDUAL shows good scalability as complexity of constraints increases. The other evaluation result reveals that each approach of the dual transformation by need may be suitable for different situation, i.e. both approaches, lazy or eager, may become options for TABDUAL customization.

TABDUAL still has much room for improvement, which we shall explore in the future. We also look forward to applying TABDUAL, integrating it with other logic programming features (e.g. program updates, uncertainty), to moral reasoning [6, 15].

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. 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.
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)*, volume 3029 of *LNAI*, pages 838–847. Springer, 2004.
3. M. Denecker and A. C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond*. Springer Verlag, 2002.
4. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.
5. 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.
6. 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.
7. J. R. Josephson and S. G. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge U. P., 1995.
8. 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.
9. 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.
10. R. Kowalski. *Computational Logic and Human Thinking: How to be Artificially Intelligent*. Cambridge U. P., 2011.
11. R. Kowalski and F. Sadri. Abductive logic programming agents with destructive databases. *Annals of Mathematics and Artificial Intelligence*, 62(1):129–158, 2011.
12. P. Lipton. *Inference to the Best Explanation*. Routledge, 2001.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. T. Swift, D. S. Warren, K. Sagonas, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, R. F. Marques, D. Saha, S. Dawson, and M. Kifer. *The XSB System Version 3.3.x Volume 1: Programmer's Manual*, 2012.