Tabling Abduction

José Júlio Alferes
Dep. Matemática, Univ. Évora and
A.I. Centre, Univ. Nova de Lisboa,
2825 Monte da Caparica, Portugal
jja@dmat.uevora.pt

Luís Moniz Pereira
A.I. Centre, Univ. Nova de Lisboa
2825 Monte da Caparica, Portugal
lmp@di.fct.unl.pt

1 Introduction

One of the main issues in implementing an abductive procedure[1] is how to collect the literals abduced to prove a goal during its evaluation. In other words, to know, after resolving each non-abductive literal in a derivation for a goal, which set of consistent abducibles, if abduced, will prove the goal.

In XSB-Prolog, which relies on tabling, when a tabled goal G succeeds it may happen that the goal's table still contains some "residue", i.e. suspended goals in the table for G. In XSB-Prolog this "residue", explained below, can be easily accessed with the system predicate $get\_residual/2$, that gets the residue for a call instance.

Our first, naïve, idea was to table all abducible predicates, and also to force them to suspend every time, in order to produce an abductive solution as a residue of the top goal. This simple idea does not work in general. In this paper we explore and generalize it in order to implement an abductive procedure based on tabling in XSB-Prolog.

We begin by briefly presenting XSB-Prolog[2], in particular by explaning what the residual program is. Then, we explore the naïve idea for abduction in ground programs. We proceed by showing how to change the first naïve idea to deal with programs with default negation and with tabled predicates. This proposal requires computing all the abductive solutions for a goal. In section 6 we show how to avoid this, i.e. how to obtain one abductive

[1]For a survey on abduction in logic programming see [4] and references therein.

[2]For more information on XSB-Prolog, and to obtain the software, see http://www.cs.sunysb.edu/~ sbprolog/xsb-page.html

solution without necessarily having to compute all the others. Finally, in section 7 we present topics on extending this work.

2 XSB and the Residual Program

The XSB-Prolog system is based on SLG resolution [1], which uses a form of tabling in addition to SLD resolution. SLG evaluates definite programs by keeping tables of sub-goals and their associated answers, and by resolving repeated instances of subgoals against answer clauses from the tables, rather than against program clauses. This way, it avoids looping. In practice, not all subgoals need to be tabled for looping to be avoided. XSB-Prolog allows for a declaration of the tabled predicates. For these SLG resolution is used, whilst for non-tabled ones SLD resolution is used instead.

For programs with default negation, when a negative ground[3] goal *not* A is called (in XSB-prolog notation, `tnot(A)`), A has to be completely evaluated before the success of A can be determined. In [1] an algorithm for detecting tabled calls that are completely evaluated is defined. It is shown there that the algoritm thus defined computes answers under the well-founded semantics [3] of normal logic programs.

The basic ideia of the aforementioned algorithm is as follows. A tabled predicate induces a stack of tabled calls according to the order in which they are created. When call A is added to the table as a new call, its matching rule instances are processed recursively. When the processing of all rule instances finally returns, the set of calls from A to the top of the call stack is considered. If none of them depends on any call below A, or the negation of any call in the stack, then the set of calls from A to the top of the stack is completely evaluated. The calls can then be popped off the stack, and all nodes waiting on the negation of some of the completed calls are processed.

When in the presence of loops over negation (i.e. non-stratified programs) two tabled calls may depend on each other, neither of which can be completely evaluated. In such cases SLG resolution delays ground negative literals that are involved in loops, so that the remaining literals in the body of a rule instance can be solved. Delayed literals may turn out to be true or false, in which cases they should be simplified from answers to tabled calls. If, in the end, some calls cannot be completely evaluated, they'll have associated with it a list of delayed ground negative literals — its residue. These are literals that are involved in a loop over negation that cannot be

[3]Negative non-ground calls are delayed until its arguments are ground. If in the end that is not possible, those calls flounder.

resolved in another way, i.e. literals that are undefined according to the well-founded semantics.

For tabled literals, answers are rules with delayed literals in the body. When an answer is returned to a waiting node, the variable bindings accumulated in the answer head are propagated through unification.

Consider the XSB program:

```
:- table r/0, s/0, p/1.
p(X) :- tnot(r).                r :- tnot(s).
p(a).                           s :- tnot(r).
```

The evaluation of $p(X)$ returns as answers exactly all the rules of the program. Note that one answer for $p(X)$ is $p(a)$. For other answers, r must be completely evaluated, and then s must be completely evaluated. Since both are delayed, and cannot be completely evaluated, they remain delayed till the end, and are returned in the bodies of the remaining answer rules.

In order to avoid combinatorial explosion [2], the delayed literals in the answer body are not propagated. Instead, positive delayed literals are created as place holders for propagating truth values of delayed ground negative literals. For example, if in the program above the rule for $p(X)$ were instead `p(X) :- r`, though the delayed literals for the call $p(X)$ would be exactly the same as above, the answer for $p(X)$ would have r in its body.

The *residual program* of a tabled query consists of all its answers plus the rules of all head atoms upon which the answer of the query depends through delayed literals in answer bodies. In XSB-Prolog, the system predicate `get_residual(Goal,Res)` returns in `Res` all the bodies of rules in the residual program whose head unifies with `Goal`. Note that `get_residual/2` does not build the residual program itself. It simply accesses the already built residual program. Thus, a call to `get_residual/2` must be preceeded by other calls to it to guarantee that the residual is built and complete.

3 Naïve abduction for ground programs

Consider the simple logic program P:

$$q \leftarrow p \qquad p \leftarrow a, b \qquad p \leftarrow c$$

where only a, b, and c are abducible. Obviously, the abductive solutions for q are $\{a, b\}$ and $\{c\}$. How to obtain them using tabling and the residues of completed tables?

First, one has to make sure that all abductive literals are suspended during the evaluation of a goal, so that they will end up in the residue. This can easily be achieved by declaring abducible literals as tabled, and by adding a loop over negation for every abducible literal. Or, for simplicity, by adding a loop over negation for the special tabled predicate $abd/1$, and translating every abducible atom A in the body of a rule into $abd(A)$. Moreover the top goal has to be tabled so that it has a residue. This way, when its table is completed, the residues in it are exactly its abductive solutions. In XSB-Prolog:

```
:- table q/0, abd/1.
abd_sol(q,Ab) :- q, fail; get_residual(q,Ab).
q :- p.                                 p :- abd(a), abd(b).
abd(X) :- tnot(abd(X)).                 p :- abd(c).
```

The `q,fail` guarantees that the table for `q` is completed. The call to `get_residual(q,Ab)` gets the residues for `q`, one at a time. In this example, for the call `abd_sol(q,Ab)` we obtain, as desired, the two solutions:

`Ab = [abd(a),abd(b)]`, and `Ab = [abd(c)]`.

This simple solution works only if there are no tabled predicates other than the top-goal and `abd/1`, and the program has no default negation except for the rule for `abd/1` one.

Indeed, if other predicates are tabled then the residue of the top-goal can have other predicates instead of abducibles only. For example, suppose that in the program above `p/0` is also tabled. Then the query `abd_sol(q,Ab)` will have a single solution `Ab = [p]`. Note that tabling of other predicates might be necessary, especially if the program has loops over those predicates (tabling them is what makes it possible to avoid looping).

Using $abd(X) \leftarrow not\ abd(X)$ is not just a programming trick. Indeed, it prevents the semantics from assigning false to $abd(X)$. It would be inconsistent. Because we want to prove the top-goal, we interpret the abductive solutions for it conditional upon making the $abd(X)$ true, which is compatible with the rule.

If the program contains negation then, the negation as failure mechanism makes it impossible to pass to the top-goal the abductions performed for failing the negated atom (abducing for failure will be explained later).

If negation is tabled then the residue is given in terms of those negated goals, not in terms of the abducibles and, as in the case where predicates other than abducibles are tabled, the naïve solution does not work. Additionally, in the presence of negation, it might be necessary to abduce the necessary falsity of some abducible in order to succeed the top-goal. By abducing falsity (and also truth) of abducibles an extra issue is in order: the consistency of the abductive solution must be checked.

4 Abduction in the presence of negation

Let's first consider the case of programs with negation in the body of rules. Take the program $P = \{q \leftarrow a, not\ p \quad p \leftarrow a, b \quad p \leftarrow c\}$, where a, b, and c are the abducibles. The only abductive solution for q is $\{a, not\ b, not\ c\}$. Note that for q to be true, a has to succeed and p has to fail. For p to fail, c must be false and, either b or a have to be false. Since a must be true in order to prove q, b must be false.

If the naïve technique is used, and the *not* is a tabled *not* (i.e. `tnot`, in XSB-Prolog), then the residue of q would simply be `[abd(a),tnot(p)]`, giving no information on how to make p false in order to prove q.

For ground programs[4], this problem can be solved by generating rules for the negation of predicates and replacing the negative calls in bodies by calls to those rules. Such rules for a negated predicate $notA$ are a kind of dual of the rules for A[5]. More precisely, consider that predicate A has a single rule, possibly with disjunctions (if A has several rules, turn them into just one by using disjunction; if A has none add the rule $A \leftarrow fail$). The single rule for not_A is obtained by: first replacing every atom B in the body for the single rule for A by not_B, and every literal $not\ B$ by B; then replacing every conjunction in the single rule for A by a disjunction, and every disjunction by a conjunction; finally turning every call to an abducible A or its negation not_A into $abd(A)$, resp. $abd(not\_A)$ [6]. In the example above:

```
:- table q/0, abd/1.
abd_sol(q,Ab) :- q, fail; get_residual(q,Ab).
q :- abd(a),not_p.
p :- abd(a), abd(b); abd(c).
abd(X) :- tnot(abd(X)).
```

[4]See some remarks on how to generalize this for non-ground programs in the last section.

[5]We thank Terrance Swift for suggesting to us the use of rules for negative predicates to approach the problem of abduction over negation. This dual of rules was first used (though for other purposes) in [5].

[6]Note that for each abducible and its negation we have two rule, which could be written instead, without changing the well-founded model, as $abd(X) \leftarrow tnot(abd(not\_X))$ and $abd(not\_X) \leftarrow tnot(abd(X))$ forming an even loop over negation. Abductive solutions would then be seen as choosing a partial stable model. This semantics for abduction was first defined in [6].

```
not_q :- abd(not_a); p.
not_p :- (abd(not_a); abd(not_b)), abd(not_c).
```

`abd_sol(q,Ab)` has two solutions:

`Ab=[abd(a),abd(not_a),abd(not_c)]` and
`Ab=[abd(a),abd(not_b),abd(not_c)]`.

To avoid the first (inconsistent) solution, a call to `consistent/1` is needed after the call to `get_residual`. Predicate `consistent/1` simply checks whether its argument refers to some pair of atoms A and not_A.

It is worth noting how adding integrity constraints in the form of denials can be done. Simply write the denials as rules having as head the special predicate $false$, and treat them as any other rule. Abductive solutions respecting the constraints can now be obtained by conjoining the top-goal with not_false. For example, if one has denials the rule for `abd_sol(q,Ab)` would then be:

```
abd_sol(q,Ab):- q_cons, fail;
                get_residual(q_cons,Ab), consistent(Ab).
q_cons :- q, not_false.
```

Alternatively, the abductive solutions for $false$ may be computed beforehand, and predicate `consistent/1` then checks that they are not included in a candidate solution.

5 Dealing with other tabled predicates

The problem of dealing with negative calls was solved above without tabling them. In general, one may have/want to table predicates other than the top-goal and `abd/1`. One situation where that is desirable is when the original program has loops, including loops over negation (for non-stratified programs). Moreover, since the top-goal must be tabled, to allow for several possible top-goals to a program, the program must perforce contain several non-abducible tabled predicates.

Recall that the problem with there being other tabled predicates is that the residue of the top-goal may be given in terms of those predicates and not in terms of the abducibles, as desired. There is no problem in having those other predicates in the residue: they can always be eliminated when presenting the abductive solution. The problem is that the residue has no information on those abductions made "below" (in the call-graph of) such tabled predicates. To solve the problem, all we need is find a way of passing the information on abductions below a tabled predicate to the (residue of the) top-goal.

First of all, after calling a tabled predicate one has to get at its residue, in order to find out what was abduced to prove it. Having this information, it has to be passed to the top-goal. This is done by passing the information to the ancestor goal in an extra argument, which in turn passes it to its ancestor, ..., up till the top-goal.

In the example of page 1, with predicate `p/0` now tabled, the program would be:

```
:- table q/1, p/0, abd/1.
abd_sol(q,Ab) :- q(X), fail;
                 get_residual(q(R),Res), filter(R,Res,Ab).
q(R) :- p, get_residual(p,R).
p :- abd(a), abd(b).
p :- abd(c).
abd(X) :- tnot(abd(X)).
```

where `filter/3` takes the residue of the predicate with the extra argument containing the residues of any other predicates called by `q`, filters out all non-abducibles, and checks the consistency of the abductive solution.

Unfortunately, this solution doesn't work in general: something more is needed. Note that when `get_residual(p,R)` is called there is no guarantee that the table for `p` is complete. Consequently abductive solutions might be lost. In this example, the only abductive solution obtained for `q` is `[abd(a),abd(b)]`. The other is lost because the call to `get_residual(p,R)` is done before the other solution with residue `[abd(c)]` is added to the table.

Note that this is exactly the same problem we had with the top-goal where, to obtain all abductive solutions, one had to find all possible derivations for it (to complete the table) before getting the residue. The same method can be used now for completing the table of predicate `p`. In the example this is done replacing the rule for `q/1` by:

```
q(R) :- p, fail; get_residual(p,R).
```

These calls to `get_residual/2` in other places in the derivation (other than the top-goal) give us the opportunity to make consistency checks before the whole abductive solution is obtained. This can be done simply by calling `consistent/1` after the call to `filter` in the places where intermediate consistency checks are desired.

Note also that, when there is more than one residue producing predicate in a body, then the argument must contain them all, say in a list.

6 Obtaining one solution at a time

Finding all derivations of tabled predicates before getting at their residue can be quite inefficient. We are obtaining all abductive solutions before presenting the first. Here we sketch how to obtain an abductive solution before computing them all.

Consider the call `p, get_residual(p,R)` whose only solution is:

```
R=[abd(a), abd(b)].
```

The other solution (`R=[abd(c)]`) is not obtained because after getting the first residue, `p` is already proven and so no other solutions are tried. However in the modified:

```
p(1) :- abd(a), abd(b).              p(2) :- abd(c).
abd(X) :- tnot(abd(X)).
```

the call to `p(X), get_residual(p(X),R)` would have the two solutions.

This technique to obtain a solution avoids generating all derivations for `p` before accessing its residue. In fact, if one adds an extra argument to each (non-abducible) predicate, and numbers the rules of the predicate using that argument, then `get_residual/2` can be invoked before trying all possible derivations for the tabled predicate.

Note that some predicates do not need this extra argument. The `q` in the example does not need the argument because it already has a rule specific valued one for passing residues, and thus this argument already does the job.

This technique concludes our solution to abduction in ground programs. It should be noted that, by so doing, simplifications in the residue which are only possible after completing a table are not performed. But this only means that non-minimal abductive solutions are obtained. For minimality, this method requires computing all solutions first, though further techniques can improve on it.

7 Conclusion and further topics

We have fostered a new method to implement abduction in logic programming relying crucially on the tabling facilities of XSB. The techniques employed may be of a more general use when programming with tabling. Below we illustrate how yet other facilities may be conjoined into our abductive approach.

Constraints Abduction provides conditional answers to queries. So do programs with constraints. A query to a CLP program can be envisaged as having its answers conditional upon the successful evaluation of the collected constraints. It is thus expectable that the abduction mechanism might be used to collect constraints, if these are treated as abducibles. By employing such a uniform approach one can combine logical and numerical constraints. Consider the constraint abductive program, where constrained variables are atoms, and both `a` and the constraints are abducible:

```
p :- not q.                          r :- not a.
q :- r, not x>2.                     r :- not x+1=\=1.
```

Its abductive translation (where negating constraints is trivially performed):

```
p :- not_q.                 not_p :- q.
q :- r, abd( x=<2 ).        not_q :- not_r ; abd( x>2 ).
r :- not_a.                 not_r :- abd(a), abd( x+1=1 ).
r :- abd( x+1=\=1 ).        abd(X) :- tnot(abd(X)).
```

The two abductive solutions for `p` are: `[a,x+1=1]` and `[x>2]`. Evaluation of constraints during the abduction process can be treated like the evaluation of consistency of an ongoing abduction. In particular the constraints may fail, or be simplified, before they are passed on via the *get_residual* mechanism.

Constructive negation To deal with non-ground negated goals constructive negation is required. That is, we need to construct unification constraints. These can be handled by abduction again, similarly to the numerical constraints above. Take:

```
p(X) :- not q(X).     q(a) :- r(Y).         r(b).
                      q(X) :- s(X).         s(f(Z)).
```

Its negative counterpart is:

```
not_p(W) :- q(W).
not_q(W) :- (abd(W\=a) ; W=a,not_r(Y)),not_s(W).
not_r(V) :- abd(V\=b).
not_s(U) :- abd(U\=f(Z)).
```

The potential abductive solutions for `p(W)` are `[W\=a,W\=f(Z)]` and `[Y\=b, a\=f(Z)]` (the latter for `W=a`), but only the first subsists because `Y` is not constrained by calls to `p(W)`.

Explanations The argument introduced in the abduction transformation for passing residues can also be used to build explanations for the abductive solutions. These can be fine-tuned according to the desired level of detail. Lack of space prevents further elaboration.

Global consistency checking The consistency checking of an abductive solution found for some subgoal can be made to consider too the global abductive solution so far constructed for the topgoal. The information on the global solution can be collected and transmitted by extra arguments added to predicates by the abduction transformation. Lack of space prevents further elaboration.

Implementation and application We have implemented and successfully tested our approach using the XSB system. Its application to a medical domain project at Stony Brook is foreseen.

8 Acknowledgements

This work was supported by project MENTAL (PRAXIS XXI 2/2.1/TIT/-1593/95), project REAP (FLAD), and a NATO sabbatical scholarship to L.M.Pereira. It was partly carried out during a visit of the authors to the Computer Science Department at Stony Brook. We thank T. Swift for suggesting to us the rules for negative predicate, D. S. Warren and K. Sagonas for discussions on the use of XSB for abduction, and João Leite for helping us test the implementation.

References

[1] W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[2] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 1996.

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

[4] A. C. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[5] L. M. Pereira, J. N. Aparício, and J. J. Alferes. A derivation procedure for extended stable models. In *IJCAI'91*. Morgan Kaufmann, 1991.

[6] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Nonmonotonic reasoning with well founded semantics. In Koichi Furukawa, editor, *8th Int. Conf. on Logic Programming*, pages 475–489. MIT Press, 1991.