Inspecting Side-Effects of Abduction in Logic Programs

Luís Moniz Pereira and Alexandre Miguel Pinto
{lmp|amp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

Abstract. In the context of abduction in Logic Programs, when finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, that is without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. We show how this type of reasoning requires a new mechanism, not provided by others already available. To achieve it, we present the concept of Inspection Point in Abductive Logic Programs, and show, by means of examples, how one can employ it to investigate side-effects of interest (the *inspection points*) in order to help choose among abductive solutions. We show how to implement inspection points on top of already existing abduction solving systems — ABDUAL and XSB-XASP — in a way that can be adopted by other systems too.
Keywords: Logic Programs, Abduction, Side-Effects.

1 Introduction

Abductive logic programming offers a formalism to declaratively express and solve problems in areas such as decision-making, diagnosis, planning, belief revision and hypothetical reasoning.
When finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, i.e. without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. We show how this type of reasoning requires a new abduction mechanism, that of *Inspection Points* (IPs).
Electing a specific abducible occurrence as an inspection point can be afforded by using an intentional abduction device, for convenience dubbed “meta-abduction" or “conditional abduction”; that is, in lieu of abducing that occurrence, one instead (meta-) abduces just the *intent* to simply check that the abducible’s actual abduction occurs somewhere in the abductive solution, by virtue of some other occurrence of it. Consequently, as we shall see, inspecting the side-effects of abduction is achievable by using abduction itself.

We begin by presenting the motivation, plus some background notation and definitions follow. Then issues of reasoning with logic programs are addressed in section

2, in particular, we take a look at abductive reasoning and the nature of backward and forward chaining and their relationship to query answering in an abductive framework. In section 3 we introduce inspection points, illustrate their need and their use with examples, and provide a declarative semantics. In section 4 we describe in detail our implementation of inspection points and illustrate its workings with an example. We close with conclusions, comparisons, and future work.

1.1 Motivation

Often, besides needing to abductively discover which hypotheses to assume in order to satisfy some condition, we may also want to know some of the side-effects of those assumptions; in fact, this is rather a rational thing to do. But, most of the time, we do not wish to know *all* possible side-effects of our assumptions, as some of them may be irrelevant to our concern, e.g. in decision-making. Likewise, the side-effects inherent in abductive explanations might not all be of interest, e.g. in model-based fault-diagnosis. Another common application of abductive reasoning is that of finding which actions to perform, action names being coded as abducibles; again, only some of an action's side-effects may be of interest. A simple example will help bring out the abduction side-effect issue and our approach to it.

Example 1. **Relevant and irrelevant side-effects.** Consider this logic program where $drink\_water$ and $drink\_beer$ are abducibles. Suppose we want to satisfy the Integrity Constraint (IC), and also to check if we get drunk or not. However, we do not care about the glass becoming wet — that being completely irrelevant to our current concern. Thus, in general, computation of whole models can be a waste of time since we are normally only interested, as for side-effects, in some subset of the program's literals.

$\leftarrow thirsty, not\ drink.$	% this is an Integrity Constraint
$wet\_glass \leftarrow use\_glass.$	$use\_glass \leftarrow drink.$
$drink \leftarrow drink\_water.$	$drink \leftarrow drink\_beer.$
$thirsty.$	$drunk \leftarrow drink\_beer.$
$unsafe\_drive \leftarrow drunk.$	

Moreover, in this example, we may wish to decide a possible action (whether to drive or not) only **after** we know which side-effects are true. In such cases, we do not want simply to introduce an extra IC expressed as $\leftarrow not\ unsafe\_drive$, because that would always impose abducing $not\ drink\_beer$, irrespective of whether we are not even considering to drive. We want to allow all possible abductive solutions for the single IC $\leftarrow thirsty, not\ drink$ and only then check for the side-effects of each solution, in order to then decide the driving action.
What we need is an inspection mechanism that permits checking the truth value of given side-effect literals (like $drunk$) as a consequence of abductions made to satisfy a given query and the program's ICs, but without further abducing whilst checking. This is achieved simply via our $inspect/1$ meta-predicate, by introducing instead the extra IC $\leftarrow inspect(not\ unsafe\_drive)$, rather than just $\leftarrow not\ unsafe\_drive$. The so-formulated (passive) IC is not allowed to be met by actively introducing abductions to that effect, but only by consuming abductions introduced to satisfy the query and other (active) ICs, like $\leftarrow thirsty, not\ drink$.

1.2 Background Notation and Definitions

Definition 1. ***Logic Rule.*** *A Logic Rule has the general form*
$H \leftarrow B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$
where H is an atom, and the B_i and C_j are atoms.

H is the head of the rule, and $B_1, \ldots, B_n, not\ C_1, \ldots, not\ C_m$ is its body, where any rule variables are deemed universally quantified. Throughout, we use 'not ' to denote default negation. When the body is empty, we say its head is a fact and write the rule just as H. If the head is empty, the rule is said to be an Integrity Constraint (IC). The atoms $true$ and $false$ are by definition respectively true and false in every interpretation.

Definition 2. ***Logic Program.*** *A Logic Program (LP for short) P is a (possibly infinite) set of Logic Rules, where non-ground rules stand for all their ground instances.*

In this paper, we consider solely so-called Normal LPs (NLPs), those whose heads of rules are positive literals, i.e. positive atoms, or empty, as per the above definition of rules. We focus furthermore on abductive logic programs, i.e. NLPs allowing for *abducibles*—user-specified positive literals without rules, whose truth-value is **not** assumed initially. Abducible instances or their default negations may appear in bodies of rules, like any other literal. They stand for hypotheses, each of which may independently be assumed true, in positive literal or default negation form, as the case may be, in order to produce an abductive solution to a query.

Definition 3. ***Abductive Solution.*** *An abductive solution is a consistent set of abducible instances or their negations that, when replaced by $true$ everywhere in P, or equivalently simply omitted, affords a (Herbrand) model of P that satisfies the query and (of course) the ICs—a so-called abductive model, for the specific semantics being used on P.*

We replace abducibles or their negations into P—instead of the more standard adding of abducibles as facts to P—because we also may abduce negations of abducibles, since the latter are **not** assumed *false* by default to begin with.

2 Abductive Reasoning with Logic Programs

Logic Programs have been used for a few decades now in knowledge representation and reasoning. Amongst the most common kinds of reasoning performed using them, one can find deduction, induction and abduction. Abduction, or inference to the best explanation, is a reasoning method whereby one chooses those hypotheses that would, if true, best explain the observed evidence—by meeting the corresponding ICs—and satisfy some query. Within deduction, and its abduction counterpart, so-named "brave" and "cautious" reasoning varieties are distinguished. "Brave" reasoning consists in finding if there exists at least one consistent model of the program—according to some pre-established semantics—which entails the query. "Cautious" reasoning demands that every model of the program entail the query.
In LPs, abductive hypotheses (or *abducibles*) are named literals of the program which

have no rules. They can be considered *true* or *false* for the purpose of answering a query. Abduction in LPs ([1, 5, 6, 10, 11]) can naturally be used in top-down query-oriented proof-procedures to find an (abductive) solution to a query, where the abducibles in the solution are leaves in the procedure's query-rooted call-graph—that is the graph recursively engendered by the procedure calls from literals in bodies of rules to heads of rules, and thence from the literals in a rule's body.
When query-answering, wherein abduction is enjoined as needed, if we know the underlying semantics is *relevant*, i.e. guarantees it is enough to use only the rules relevant to the query (those in its call-graph) to assess its truthfulness, then we need not compute a whole model in order to find an answer to a query: it suffices just to use the call-graph or relevant part of the program and determine the truth of a subset of the program's literals, those in the query's call-graph. Thus, top-down finding a solution to a query, dubbed "backward chaining", is possible only when the underlying semantics is relevant, in the above sense, because then the extension of that subset to a full model is guaranteed.

When performing abductive reasoning, we typically wish to find by need only—via backward chaining—the abductive solutions to a query. However, sometimes we also want to know which are some of the consequences (or side-effects) of such abductive solutions. I.e., we desire to know the truth value of some other literals, not part of the query's call-graph, whose truth-value may be determined by a found abductive solution. In some cases, we might be interested in knowing every possible side-effect—the truth-value of every literal in a complete model satisfying the query. In other situations though, our focus is frequently just on some specific side-effects.
In our approach, the side-effects of interest are explicitly indicated by the user, by wrapping the corresponding goals within the reserved construct $inspect/1$.

2.1 Abductive Logic Program Procedures

Currently, the standard 2-valued semantics used by the logic programming community is Stable Model (SM) semantics [9]. Its properties are well known and there are efficient implementations (such as *DLV* and *SModels* [4, 12]). However, SM misses out on some important properties, both from the theoretical and practical perspectives: guarantee of model existence for every NLP, relevance and cumulativity—though the latter will not be of concern in the present context. Most importantly, since SM do not enjoy relevance they cannot just use backward chaining for query answering, irrespective of whether abduction is involved— indeed, odd-loops over default negation, outside the query's call-graph, may prevent model existence. This means SM implementations need to compute whole models, and so one will waste computational resources, because extra time and memory are required to compute parts of the model which are irrelevant to the query and ICs, i.e. outside their call-graph. The problem becomes compounded in abductive reasoning, because then the truth-value combinations of every abducible must be considered in order to provide complete models, even where abducibles are irrelevant to the query at hand. Moreover, such irrelevant abducibles and their combinations must subsequently be weeded out from the abductive models, at additional computational cost. On the other hand, because whole models are computed side-effects of abductive choices are computed too.

The Well-Founded Semantics (WFS) [8]—which enjoys model existence, relevance, and cumulativity—allows for top-down abductive query answering. Whole models need not to be computed, but then testing for side-effects involves extra querying about side-effected literals. One important issue we address with the introduction of inspection points is how to query the side-effects of a given abductive solution without performing additional abductions in the process. In so doing we avoid producing whole models still. We used WFS in the specific implementation described in section 4 based on ABDUAL [1]. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued.

Because they do not depend on any other literal in the program, abducibles can be modeled in a LP system without specific abduction mechanisms by automatically including for each *abducible* an even loop over default negation, e.g.,

$$abducible \leftarrow not\ neg\_abducible. \qquad neg\_abducible \leftarrow not\ abducible.$$

where $neg\_abducible$ is a new abducible atom, representing the (abducible) negation of the abducible. This way, under the SM semantics, a program may have models where some $abducible$ is *true* and another where it is *false*, i.e. $neg\_abducible$ is *true*. If there are n abducibles in the program, there will be 2^n models corresponding to all the possible combinations of *true* and *false* for each. Under the WFS without a specific abduction mechanism, both $abducible$ and $neg\_abducible$ remain *undefined* in the Well-Founded Model (WFM), but may hold (as alternatives) in Partial Stable Models. In ABDUAL, however, a specific distinct mechanism is employed: abducibles and their negations are explicitly collected during search.

Using the SM semantics, unless the program is stratified, abduction must be done by guessing the truth-value of each abducible, providing the whole model and testing it for stability; whereas using WFS, even for non-stratified programs, abduction can be performed *by need*, induced by the top-down query solving procedure, solely for the relevant abducibles—i.e., irrelevant abducibles are left unconsidered. Thus, top-down abductive query answering is a means of finding those abducible values one might commit to in order to satisfy a query.

A new additional procedural preoccupation, addressed in this paper, is when one wishes to only passively determine which abducibles would be sufficient to satisfy some goal but without actually abducing them, just consuming other goals' needed and produced abductions. The difference is subtle but of importance, and it requires a new construct. Its mechanism, of *inspecting without abducing*, can be conceived and implemented through *meta-abduction*, or conditional abduction, as discussed in detail in the sequel.

3 Inspection Points

When faced with some situation where several alternative courses of actions are available a rational agent must decide and choose which action to take. *A priori* preferences can be applied before choosing in order to reduce the number of considerable possible actions curtailing the explosion of irrelevant combinations of choices, but still several

(possibly exclusive) may remain available.
To make the best possible informed decision, and commit to a course of action, the agent must be enabled to foresee the consequences of its actions and then prefer on the basis of those consequences (with *a posteriori* preferences). Choosing which set of consequences is most preferred corresponds to an implicit choice on restricting which course of action to take. But only consequences relevant to the *a posteriori* preferences should be calculated: there are virtually infinitely many consequences of a given action, most of which are completely irrelevant to the preference-based decision making. Other consequences may be just predictions about the present state of the world, and observing whether they are verified can eliminate hypothetical scenarios where certain decisions would appear to make sense.
Not all consequences are experimentally observable though, hence Inspection Points (IPs) may serve to focus on the ones that are, and thus guide the experimentation required to decide among competing hypotheses. That is, IPs can be put to the service of sifting through competing explanations. In science, such decisive consequences are known as "crucial" side-effects, because they exclude untoward hypotheses. However, this is not the place to discuss the varied uses of abduction and its pragmatics. Instead, we direct the reader to Robert Kowalski's online book draft, available at his home page.

3.1 Backward and Forward Chaining

Abductive query-answering is intrinsically a backward-chaining process, a top-down dependency-graph oriented proof-procedure. Finding the side-effects of a set of abductive assumptions may be conceptually envisaged as forward-chaining, as it consists of progressively deriving consequences from the assumptions until the truth value of the chosen side-effect literals is determined.
The problem with full-fledged forward-chaining is that too many (often irrelevant) conclusions of a model are derived. Wasting time and resources deriving them only to be discarded afterwards is a flagrant setback. Even worse, in combinatorial problems, there may be many alternative solutions whose differences repose just on irrelevant conclusions. So, the unnecessary computation of irrelevant conclusions in full forward-chaining may be multiplied, leading to immense waste.

A more rational solution, when one is focused on some specific conclusions from a set of premises, is afforded by a selective top-down ersatz forward-chaining. In this setting, the user can specify the conclusions she is focused on, and only those are computed in a backward-chaining fashion, checking whether they are consequences of desired abductions, but without further abducing. Combining backward-chaining with such ersatz forward-chaining allows for a greater precision in specifying what we wish to know, and altogether improve efficient use of computational resources, because focusing on the points of interest.
Crucially, if abduction is enabled, the computation of side-effects should take place without further abduction, passively —but not destructively— just "consuming" abducibles "produced" elsewhere by abduction, for the top query.
In the sequel, we show how such ersatz forward-chaining from a set of hypotheses can be achieved by backward chaining from the consequences focused on—the inspection points—by virtue of a controlled form of abduction.

3.2 Meta-Abduction for Side-Effects Inspection

"Meta-abduction" is used in *abduction inhibited inspection*. Intuitively, when an abducible is considered under mere inspection, meta-abduction abduces only the intention to *a posteriori* check for its abduction elsewhere, i.e. it abduces the intention of verifying that the abducible is indeed adopted—that is, it abduces on condition. In practice, when we want to meta-abduce some abducible 'X', we abduce a literal '$consume(X)$' (or '$abduced(X)$'), which represents the intention that 'X' is eventually abduced elsewhere in the process of finding an abductive solution. The pairing check is performed after a complete abductive answer to the top query is found. Meta-abduction, by its very nature, can be supported by any abduction capable system.
In the examples below, we are not propounding a methodology for using abduction, but simply illustrating the concepts we have introduced.

Example 2. **Police and Tear Gas Issue.** Consider this NLP, where '$tear\_gas$', '$fire$', and '$water\_cannon$' are the only abducibles. Notice the two rules for '$smoke$'. The first states that one explanation for smoke is fire, when assuming the hypothesis '$fire$'. The second states '$tear\_gas$' is also a possible explanation for smoke. However, the presence of tear gas is a much more unlikely situation than the presence of fire; after all, tear gas is only used by police to contain riots and that is truly an exceptional situation. Fires are much more common and spontaneous than riots. For this reason, '$fire$' is a much more plausible explanation for '$smoke$' and, therefore, in order to let the explanation for '$smoke$' be '$tear\_gas$', there must be a plausible reason—imposed by some other likely phenomenon. This is represented by $inspect(tear\_gas)$ instead of simply '$tear\_gas$'.

$\leftarrow police, riot, not\ contain.$	% this is an Integrity Constraint
$contain \leftarrow tear\_gas.$	$contain \leftarrow water\_cannon.$
$smoke \leftarrow fire.$	$smoke \leftarrow inspect(tear\_gas).$
$police.$	$riot.$

The '$inspect$' construct disallows regular abduction—allowing only a conditional meta-abduction to be performed whilst trying to solve '$tear\_gas$'. I.e., if we take tear gas as an abductive solution for smoke, this rule imposes that the step where we abduce '$tear\_gas$' is performed elsewhere, not under the derivation tree for '$smoke$'. Thus, '$tear\_gas$' is an *inspection point*. The IC, because there is '$police$' and a '$riot$', forces '$contain$' to be *true*, and hence, '$tear\_gas$' or '$water\_cannon$' or both must be abduced. '$smoke$' is only explained if, at the end of the day, '$tear\_gas$' is abduced to enact containment. Abductive solutions should be plausible, and '$smoke$' is plausibly explained by '$tear\_gas$' if there is a reason, a best explanation, that makes the presence of tear gas plausible; in this case the riot and the police. Crucially, if the police were not around, or there was no riot, '$tear\_gas$' could not be abduced to explain '$smoke$'. Plausibility is an important concept in science, for lending credibility to hypotheses. Assigning plausibility measures to situations is an orthogonal issue though.

Example 3. **Nuclear Power Plant Decision Problem.** This example was extracted from [13] and adapted to our current designs, and its abducibles do not represent actions. In a nuclear power plant there is decision problem: cleaning staff will dust the

power plant on cleaning days, but only if there is no alarm sounding. The alarm sounds when the temperature in the main reactor rises above a certain threshold, or if the alarm itself is faulty. When the alarm sounds everybody must evacuate the power plant immediately! Abducible literals are $cleaning\_day$, $temperature\_rise$ and $faulty\_alarm$.

$$\begin{array}{ll} dust & \leftarrow cleaning\_day, inspect(not\ sound\_alarm) \\ sound\_alarm & \leftarrow temperature\_rise \\ sound\_alarm & \leftarrow faulty\_alarm \\ evacuate & \leftarrow sound\_alarm \\ & \leftarrow not\ cleaning\_day \end{array}$$

Satisfying the unique IC imposes $cleaning\_day$ *true* (we may not employ a fact as $cleaning\_day$ is an abducible and these may not have rules), and that gives us three minimal abductive solutions to what happens on a cleaning day:

$S_1 = \{dust, cleaning\_day\}$,
$S_2 = \{cleaning\_day, sound\_alarm, temperature\_rise, evacuate\}$, and
$S_3 = \{cleaning\_day, sound\_alarm, faulty\_alarm, evacuate\}$.

If we pose the query $? - not\ dust$ we want to know what could justify the cleaners dusting not to occur given that it is a cleaning day (enforced by the IC). However, we do not want to abduce the rise in temperature of the reactor nor to abduce the alarm to be faulty in order to prove $not\ dust$. Any of these justifying two abductions must result as a side-effect of the need to explain something else, for instance the observation of the sounding of the alarm, expressible by adding the IC $\leftarrow not\ sound\_alarm$, which would then abduce one or both of those two abducibles as plausible explanations. Hence S_2 and S_3 are not solutions to the query, as intended in [13]. They would be, however, if the query were $? - not\ dust, evacuate$. The $inspect/1$ in the body of the rule for $dust$ prevents any abduction below $sound\_alarm$ to be made just to make $not\ dust$ true. One other possibility would be for two observations, coded by ICs $\leftarrow not\ temperature\_rise$ or $\leftarrow not\ faulty\_alarm$, to be present in order for $not\ dust$ to be true as a side-effect. A similar argument can be made about evacuating: one thing is to explain why evacuation takes place, another altogether is to justify it as necessary side-effect of root explanations for the alarm to go off. These two pragmatic uses correspond to different queries: $? - evacuate$ and $? - inspect(evacuate)$, respectively.

3.3 Declarative Semantics of Inspection Points

A simple transformation Π maps any NLP P, with possibly nested inspection points—that is inspection points under the scope of other ones—into a NLP TP without them.

Definition 4. ***Abductive Models.*** *Abductive Models are those models obtained by the abductive solutions—according to the base semantics which is applied to the transformed program TP—*in which each $abduced(X)$ is required to be matched by the corresponding X. *Thus the transformation Π provides a definitional transformative declarative semantics for P, no matter what the base semantics chosen and its actual implementation*

Both the Stable Models or the Well-Founded Semantics are used in this paper, corresponding to different implementations naturally. For instance, the Abductive Stable Models of some TP, are the stable models for its abductive solutions, with respect to the source abducibles for P plus those abducibles introduced by the transformation. Likewise for the Abductive Well-Founded Models.

In essence, TP adds to P duplicates of its rules, wrapping each literal with $inspect/1$, except for the abducibles, which are treated differently. Mark, below, that the abductive Stable Models of the transform TP—in which, by definition, each $abduced(X)$ is required to be matched by the corresponding X—clearly correspond to the intended meaning ascribed to the inspection points of the original program, as the example illustrates.

Definition 5. ***Transforming Inspection Points.*** *Let P be a program containing rules whose body possibly contains inspection points. The program $\Pi(P)$ consists of:*

1. *all the rules obtained from the rules in P by systematically replacing:*
   - *$inspect(not\ L)$ with $not\ inspect(L)$;*
   - *$inspect(L)$ with $abduced(a)$ if L is an abducible a, and keeping $inspect(L)$ otherwise.*
2. *plus, for each rule $A \leftarrow L_1, \ldots, L_t$ in the replaced rules of P from step 1, the additional rule:*
   $inspect(A) \leftarrow L_1^{'}, \ldots, L_t^{'}$ where for every $1 \leq i \leq t$:

$$L_i^{'} = \begin{cases} abduced(L_i) & \text{if } L_i \text{ is an abducible} \\ inspect(X) & \text{if } L_i \text{ is } inspect(X) \\ inspect(L_i) & \text{otherwise} \end{cases}$$

The semantics of the $inspect/1$ predicate is exclusively given by the generated rules for $inspect/1$. Moreover, '$abduced/1$' is an abducible, joining the original abducibles.

Example 4. **Transforming a Program P with Nested Inspection Points.**

$$\begin{array}{ll} x \leftarrow a, inspect(y), b, c, not\ d & y \leftarrow inspect(not\ a) \\ z \leftarrow d & y \leftarrow b, inspect(not\ z), c \end{array}$$

where the abducibles are a, b, c, d. Then, $\Pi(P)$ is:

$$\begin{array}{ll} x & \leftarrow a, inspect(y), b, c, not\ d \\ inspect(x) & \leftarrow abduced(a), inspect(y), abduced(b), abduced(c), not\ abduced(d) \\ y & \leftarrow not\ inspect(a) \\ y & \leftarrow b, not\ inspect(z), c \\ inspect(y) & \leftarrow not\ abduced(a) \qquad \%\ by\ two\ rewrites \\ inspect(y) & \leftarrow abduced(b), not\ inspect(z), abduced(c) \\ z & \leftarrow d \\ inspect(z) & \leftarrow abduced(d) \end{array}$$

The single abductive stable model of $\Pi(P)$—that its stable model for its single abductive solution—respecting the meaning of the inspection points declarations in P is:

$\{x, a, b, c, abduced(a), abduced(b), abduced(c), inspect(y)\}$.

Note that indeed for each $abduced(X)$ the corresponding X is in the model.

4 Implementation

We based our practical work on a formally defined, XSB-implemented, true and tried abduction system—ABDUAL [1]. ABDUAL lays the foundations for efficiently computing queries over ground 3-valued abductive frameworks for extended logic programs with integrity constraints, on the well-founded semantics and its partial stable models.

The query processing technique in ABDUAL relies on an admixture of program transformation and tabled evaluation. A transformation removes default negative literals (by making them positive) from both the program and the integrity rules. Specifically, a dual transformation is used, that defines for each objective literal O (i.e. an atom or explicit negated atom) and its set of rules R, a dual set of rules whose conclusions $not\ (O)$ are true if and only if O is false in R. Tabled evaluation of the resulting program turns out to be much simpler than for the original program, whenever abduction over negation is needed. At the same time, termination and complexity properties of tabled evaluation of extended programs are preserved by the transformation, when abduction is not needed. Regarding tabled evaluation, ABDUAL is in line with SLG [15] evaluation, which computes queries to normal programs according to the well-founded semantics. To it, ABDUAL tabled evaluation adds mechanisms to handle abduction and deal with the dual programs.

ABDUAL is composed of two modules: the preprocessor which transforms the original program by adding its dual rules, plus specific abduction-enabling rules; and a meta-interpreter allowing for top-down abductive query solving. When solving a query, abducibles are dealt with by means of extra rules the preprocessor added to that effect. These rules just add the name of the abducible (or its negation) to an ongoing list of current abductions, unless the negation of the abducible was added before to the lists, then failing in order to ensure abduction consistency. Our conditional meta-abduction is implemented adroitly by means of a reserved predicate, '$inspect/1$' taking some literal L as its argument, which engages the abduction mechanism to try and discharge any conditional meta-abductions performed under L by matching with the corresponding abducibles, adopted elsewhere outside from under any '$inspect/1$' call. The approach taken can easily be adopted by other abductive systems, albeit in part—e.g. inspecting only abducibles directly, and so omitting inspection nesting too—as we had the occasion to check, namely with the authors of system [3]. We have also enacted an alternative implementation, relying on XSB-XASP and the declarative semantics transformation above, which is reported further below.

Procedurally, in the ABDUAL implementation, the checking of an inspection point corresponds to performing a top-down query-proof for the inspected literal, but with the specific proviso of disabling new abductions during that proof. The proof for the inspected literal will succeed only if the abducibles needed for it were already adopted, or will be adopted, in the present ongoing solution search for the top query. Consequently, this check is performed after a solution for the query has been found, except for "quick-kill" cases, as when the opposite abduction has already been collected in the ongoing solution. At "inspection-point-top-down-proof-mode", whenever an abducible is encountered, instead of adopting it, we simply adopt the intention to *a posteriori* check if the abducible is part of the answer to the query. That is, one conditionally (meta-) abduces the checking of some abducible A, and the check consists in confirm-

ing that A is part of the abductive solution by matching it with the object of the check. According to our method, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals, those to be subject to inspection mode, with the reserved construct '$inspect/1$'.

4.1 ABDUAL with Inspection Points—Details

Inspection points in ABDUAL function mainly by means of controlling the general abduction step, which involves very few changes, both in the pre-processor and the meta-interpreter, that might be imported into other abduction systems. Whenever an '$inspect(X)$' literal is found in the body of a rule, where 'X' is a goal, a meta-abduction-specific counter—the '$inspect\_counter$', initialized with zero—is increased by one, in order to keep track of the allowed character, active or passive, of ongoing abduction performing. The top-down evaluation of the query for 'X' then proceeds normally. Active abductions are only allowed if the counter is set to zero, otherwise only meta-abductions are permitted. After finding an abductive solution to query 'X', the counter is decreased by one, since that inspection execution of X has been completed. Backtracking over counter assignations is duly accounted for. Of course, this way of implementing the inspection points (with a single '$inspect\_counter$') presupposes the abductive query answering process is carried out "depth-first", guaranteeing that the order of the literals in the bodies of rules actually corresponds to the order they are processed in. For simplicity of description, we assume such a "depth-first" discipline in the implementation of inspection points, described in detail below. We then lift this restriction at the end of the subsection.

Changes to the pre-processor:

1. A new dynamic predicate was added: the '$inspect\_counter/1$'. This is initialized to zero ('$inspect\_counter(0)$') via an assert, before a top-level query is launched.
2. The original rules for the normal abduction step are now preceded by an additional condition checking that the '$inspect\_counter$' is indeed set to zero.
3. Extra rules for the "inspection" abduction step are added, preceded by a condition checking the '$inspect\_counter$' is set to greater than zero. When these rules are called, the corresponding abducible 'A' is not abduced as it would happen in the original rules; instead, '$consume(A)$' (or '$abduced(A)$') is abduced. This corresponds to the conditional meta-abduction: we abduce the need to abduce 'A', the need to 'consume' the abduction of 'A', which is finally checked when derivation for the very top goal is finished.

Changes to the meta-interpreter: The changes to the meta-interpreter include all the remaining processing needed to correctly implement inspection points, namely the matching of the abduction of '$consume(X)$' against the abduction of 'X'. If a conditional meta-abduction on 'X' (producing '$consume(X)$') is not matched by an actual

abduction on 'X' when the end of solving the top query is reached, the candidate abductive answer is considered invalid and the attempted query solving fails. On backtracking, an alternative abductive solution (possibly with other meta-abductions) will be sought.

In detail, the changes to the meta-interpreter include:

1. Two "quick-kill" rules for improved efficiency that detect and immediately solve trivial cases for conditional meta-abduction:
   - When literal 'X' about to be meta-abduced ('$consume(X)$' about to be added to the abductions list) has actually been abduced already ('X' is in the abductions list) the meta-abduction succeeds immediately and '$consume(X)$' is not added to the abductions list;
   - When the situation in the previous point occurs, but with '$not\ X$' already abduced instead, the meta-abduction immediately fails.
2. Two new rules for the general case of meta-abduction, that now specifically treat the '$inspect(not\ X)$' and '$inspect(X)$' literals. In either rule, first we increase the '$inspect\_counter$' mentioned before, then proceed with the usual meta-interpretation for '$not\ X$' ('X', respectively), and, when this evaluation succeeds, we then decrease '$inspect\_counter$'.
3. After an abductive solution is found to the top query, ensure that every meta-abduction, i.e. every '$consume(X)$' literal abduced, is indeed matched by a corresponding and consistent abduction, i.e. that it is matched by the abducible 'X' in the abductions list; otherwise the tentative solution found fails.

A counter—'$inspect\_counter$'—is employed instead of a simple toggle because several '$inspect(X)$' literals may appear at different graph-depth levels under one another, and resetting a toggle after solving a lower-level meta-abduction would enable producer abductions under the higher-level meta-abduction. An example clarifies this.

Example 5. **Nested Inspection Points.** Consider again the program of the previous example, where the abducibles are a, b, c, d:

$$x \leftarrow a, inspect(y), b, c, not\ d. \qquad y \leftarrow inspect(not\ a).$$
$$z \leftarrow d. \qquad\qquad\qquad\qquad y \leftarrow b, inspect(not\ z), c.$$

When we want to find an abductive solution for x—skipping over the low-level technical details—we proceed as follows:

1. a is an abducible and since the '$inspect\_counter$' is still set initially to 0 we can abduce a by adding it to the running abductions list;
2. y is not an abducible and so we cannot use any "quick-kill" rule on it. We increase the '$inspect\_counter$'—which now takes the value 1—and proceed to find an abductive solution to y;
3. Since the '$inspect\_counter$' is different from 0, only meta-abductions are allowed;
4. Using the first rule for y we need to '$inspect(not\ a)$', but since we have already abduced a, a "quick-kill" is applicable here: we already know that this '$inspect(not\ a)$' will fail. The value of the '$inspect\_counter$' will remain 1;

5. On backtracking, the second rule for y is selected, and now we meta-abduce b by adding '$consume(b)$' to the ongoing abductions list;
6. Increase the '$inspect\_counter$' again, making it take the value 2, and continue on searching for an abductive solution to $not\ z$;
7. The only solution to $not\ z$ is by abducing $not\ d$, but since the '$inspect\_counter$' is greater than 0, we can only meta-abduce $not\ d$, i.e.
   '$consume(not\ d)$' is added to the running abductions list;
8. Returning to y's rule: the meta-interpretation of '$inspect(not\ z)$' succeeds and so we decrease the '$inspect\_counter$' by one—it takes the value 1 again. Now we proceed and attempt to solve c;
9. c is an abducible, but since the $inspect\_counter$ is set to 1, we only meta-abduce c by adding '$consume(c)$' to the running abductions list;
10. Returning to x's rule: the meta-interpretation of '$inspect(y)$' succeeds and so we decrease the '$inspect\_counter$' once more, and it now takes the value 0. From this point onwards regular abductions will take place instead of meta-abductions;
11. We abduce b, c, and $not\ d$ by adding them to the abductions list;
12. A tentative abductive solution is found to the initial query. It consists of the abductions list: $[a, consume(b), consume(not\ d), consume(c), b, c, not\ d]$;
13. The abductive solution is now checked for matches between meta-abductions and producer abductions.
    In this case, for every '$consume(A)$' in the abduction list there is actually an A also in the abduction list, i.e. each abduction intention '$consume(A)$' is satisfied by a producer abduction A, where the A in $consume(A)$ is just any abducible literal a or its default negation $not\ a$. It is irrelevant in which order a '$consume(A)$' and the corresponding A appear or were placed in the abductions list. Because this final checking step succeeds, the abductive solution is actually accepted.

In this example, we can clearly see that the $inspect$ predicate can be used on any arbitrary literal, and not just on abducibles.

The correctness of this implementation against the declarative semantics provided before can be sketched by noticing that whenever the $inspect\_counter$ is set to 0 the meta-interpreter performs actual abduction, which corresponds to the use of the original program rules; whenever the $inspect\_counter$ is set to some value greater than 0, the meta-interpreter just abduces $consume(A)$—where A is the abducible being checked for its abduction being produced elsewhere—and that corresponds to the use of the transformed program rules for the $inspect$ predicate.

The implementation of ABDUAL with inspection points is available on request.

More general query solving In case the "depth-first" discipline is not followed, either because goal delaying is taking place, or multi-threading, or co-routining, or any other form of parallelism is being exploited, then each queried literal will need to carry its own list of ancestors with their individual '$inspect\_counters$'. This is necessary so as to have a means, in each literal, to know which and how many $inspect$s there are between the root node and the currently being processed literal, and which $inspect\_counter$ to update; otherwise there would be no way to know if abductions or meta-abductions should be performed.

4.2 Alternative Implementation Method

The method presented forthwith is an avenue for the implementation of the inspection points mechanism through a simple syntactic transformation that can readily be employed by any SMs system, like SModels or DLV. Using a SMs implementation alone, one can get the abductive SMs of some program P by computing the SMs of P', where P' is obtained from P by applying the program transformation we presented before for the declarative semantics of the inspection points, and then adding an even loop over negation for each declared abducible—as shown in section 2.1. When using XSB-Prolog's XSB-XASP interface, the process method is the same as for when using a SMs implementation alone, but instead of sending the whole P' to the SMs engine, only the residual or remainder program [2], the one that results from a query evaluated in XSB using tabling [14], relevant for the query at hand, is sent. This way, abductive reasoning may benefit from the relevance property enjoyed by the Well-Founded Semantics implemented in XSB-Prolog's SLG-WAM.

Given the top-down proof procedure for abduction, implementing inspection points for program P becomes just a matter of adapting the evaluation of derivation subtrees falling under '$inspect/1$' literals, at meta-interpreter level, subsequent to performing the transformation $\Pi(P)$ presented before, which actually defines the declarative semantics. Basically, any considered abducibles evaluated under '$inspect/1$' subtrees, say A, are codified as '$abduced(A)$', where, as in section 2.1:

$$abduced(A) \leftarrow not\ neg\_abduced(A)$$
$$neg\_abduced(A) \leftarrow not\ abduced(A)$$

All $abduced/1$ literals collected during computation of the residual program are later checked against the stable models themselves. Every '$abduced(A)$' in a model must pair with a corresponding abducible A for the model to be accepted.

5 Conclusions, Comparisons, and Future Work

In the context of abductive logic programs, we have presented a new mechanism of inspecting literals that can be used to check for side-effects, by relying on conditional meta-abduction. We have implemented the inspection mechanism within the Abdual [1] meta-interpreter, as well as in XSB-XASP. We have further checked that our approach can easily be adopted, in part, by other systems [3] with the help of these cited authors.

HyProlog [3] is an abduction/assumption system which allows for the user to specify if an abducible is to be consumed only once or many times. In HyProlog, as the query solving proceeds, when abducible/assumption consumptions take place, they are executed by storing the corresponding consumption intention in a store. After an abductive solution for a query is found, the actual abductions/assumptions are matched against the consumption intentions. Overall, there is not such a big gap between the operational semantics of HyProlog and the inspection points implementation we present; however, there is a major functional difference: in HyProlog we can only specify consumption directly on abducibles, whereas in our more general inspection points approach we can

declare inspection of any literal (not just abducibles)—meaning any abducible found below an inspect-wrapped literal call is automatically just inspected.

In [13], the authors detect a problem with the IFF abductive proof procedure [7] of Fung and Kowalski, in what concerns the treatment of negated abducibles in integrity constraints (e.g. in their examples 2 and 3). They then specialize IFF to avoid such problems, which arise only in ICs, and prove correctness of the new procedure. The detected problem refers to the active use of an IC comprising in its body some $notA$, where A is an abducible, whereas the intended use should be a passive one, simply checking whether some A is proved in the abductive solution found. To that effect, by means of an inference rule used during query evaluation, it's as if they replaced such occurrences of $notA$ by '$not\ provable(A)$', before moving each as a disjunct '$provable(A)$' to the IC head along with other disjuncts, so as to ensure that no new abductions are allowed during IC checking, by virtue of '$provable/1$'. For a detailed exposition the reader is referred to their section 4.2. Our own work generalizes the scope of the problem they solved, and solves the problems arising in this wider scope. For one, we abduce both positive and negative literals, and the latter are not true by default. Moreover, we allow for passive checking not just of negated abducibles but also of positive ones, as well as passive checking of any literal, whether or not abducible and whether in ICs or other rules. Furthermore, we allow to single out which specific occurrences are passive or active. Thus, we can cater for both passive and active ICs, depending on the desired usage. Our solution uses abduction itself to solve the problem, making it general for deployment in other abductive frameworks and procedures.

A future application of inspection points is planning in a multi-agent setting. An agent may have abduced a plan and, in the course of carrying out its abduced actions, it may find that another agent has undone some of its already executed actions. So, before executing an action, the agent should check all necessary preconditions still hold. Note that it should only *check*, thereby avoiding abducing again a plan for them: this way, should the preconditions hold, the agent can continue and execute the planned action. The agent should only take measures to enforce the preconditions again whenever the check fails. Clearly, an "inspection" of the preconditions is what is needed here.

More generally, inspection points afford us with the ability to avoid having to generate complete abductive models in order to glean the consequences of interest of abductive solutions. The developed techniques can be employed too for permitting passive ICs, which are not allowed to actively abduce but only to verify their satisfaction with regard to given abductions, in contrast to active ICs that can further abduce in order to be satisfied. Plus, of course, to enable ICs which contain a combination of both active and passive literals.

Another future use concerns the computation of inspected consequences of partially defined 2-valued models, obtained by top-down querying of NLPs, wherein the abducibles are the default nots themselves, plus appropriate ICs to enforce consistency. Once again, the computation of complete models can thus be avoided. A 2-valued semantics which enjoys relevance must then be used, or otherwise a guarantee that the NLP is stratified or does not contain loops over default negation via an odd number of nots.

6 Acknowledgements

We thank Robert Kowalski, Verónica Dahl and Henning Christiansen for discussions, Pierangelo Dell'Acqua for the declarative semantics, and Gonçalo Lopes for help with the XSB-XASP implementation. A special thanks to our helpful reviewer, Francesca Toni.

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, July 2004.
2. S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 1(5):497–538, 2001.
3. H. Christiansen and V. Dahl. HyProlog: A new logic programming language with assumptions and abduction. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *LNCS*, pages 159–173. Springer, 2005.
4. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and advanced frontends (system description). In *12th Workshop on Logic Programming*, 1997.
5. M. Denecker and D. De Schreye. SLDNFA: An abductive procedure for normal abductive programs. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700, Washington, USA, 1992. The MIT Press.
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. T. H. Fung and R. Kowalski. The IFF proof procedure for abductive logic programming. *J. Log. Prog.*, 33(2):151 – 165, 1997.
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. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
10. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.
11. A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in AI and LP*, volume 5, pages 235–324. Oxford University Press, 1998.
12. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Procs. 4th Intl. Conf. Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, July 1997.
13. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In E. Lamma and P. Mello, editors, *AI*IA*, volume 1792 of *LNCS*, pages 49–60. Springer, 1999.
14. T. Swift. Tabling for non-monotonic programming. *AMAI*, 25(3-4):201–240, 1999.
15. T. Swift and D. S. Warren. An abstract machine for slg resolution: Definite programs. In *Symp. on Logic Programming*, pages 633–652, 1994.