Incremental Tabling for Query-Driven Propagation of Logic Program Updates

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. We foster a novel implementation technique for logic program updates, which exploits incremental tabling in logic programming – using XSB Prolog to that effect. Propagation of updates of fluents is controlled by initially keeping any fluent updates pending in the database. And, on the initiative of queries, making active just those updates up to the timestamp of an actual query, by performing incremental assertions of the pending ones. These assertions, in turn, automatically trigger system-implemented incremental bottom-up tabling of other fluents (or their negated complements), with respect to a predefined overall upper time limit, in order to avoid runaway iteration. The frame problem can then be dealt with by inspecting a table for the latest time a fluent is known to be assuredly true, i.e., the latest time it is not supervened by its negated complement, relative to the given query time. To do so, we adopt the dual program transformation for defining and helping propagate, also incrementally and bottom-up, the negated complement of a fluent, in order to establish whether a fluent is still true at some time point, or rather if its complement is. The use of incremental tabling in this approach affords us a form of controlled, but automatic, system level truth-maintenance, up to some actual query time. Consequently, propagation of update side-effects need not employ top-down recursion or bottom-up iteration through a logically defined frame axiom, but can be dealt with by the mechanics of the underlying world. Our approach thus reconciles high-level top-down deliberative reasoning about a query, with autonomous low-level bottom-up world reactivity to ongoing updates, and it might be adopted elsewhere for reasoning in logic.

Keywords: logic program updates, updates propagation, incremental tabling, dual program transformation, XSB Prolog.

1 Introduction

The tabled logic programming paradigm, i.e., logic programming (LP) with tabling mechanisms, is supported by a number of Prolog systems, to different extent. Tabling affords solutions reuse, rather than recomputing them, by keeping in tables subgoals and their answers obtained by query evaluation. Incremental tabling, available in XSB Prolog [23], is an advanced recent tabling feature that ensures the consistency of answers in

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

a table with all dynamic clauses on which the table depends. It does so by incrementally maintaining the table, rather than by recomputing answers in the table from scratch to keep it updated. The applications of incremental tabling in LP have been demonstrated in pointer analyses of C programs in the context of incremental program analyses [18], data flow analyses [19], static analyses [6], incremental validation of XML documents and push down model checking [17]. This range of applications suggests that incremental tabling lends itself to dynamic environments and evolving systems, including notably logic program updates, as we proceed to show.

In [20], an approach to logic program updates, termed EVOLP/R, theoretically based on Evolving Logic Programs (EVOLP) [1], is proposed. It simplifies EVOLP by restricting updates to fluents only. Rule updates are nevertheless achieved by attaching to each rule, in its body, a name fluent that uniquely identifies that rule (cf. [16]). Updating such a rule name fluent, via its assertion or retraction, permits time-activation or deactivation of the corresponding rule, respectively. Its implementation preliminarily exploits incremental tabling, plus another tabling feature: answer subsumption [22]. Incremental tabling of fluents is employed to automatically maintain the consistency of program states due to assertion and retraction of fluents, whether obtained as updated facts or concluded by rules. On the other hand, answer subsumption of fluents allows to address the frame problem, by automatically keeping track of the latest assertion or retraction of fluents with respect to a given query time. The combined use of incremental tabling and answer subsumption is realized in the tabled predicate $fluent(F, Ht, Qt)$: given query time Qt, it looks for dynamic definitions of fluent F, and returns Ht, the latest time fluent F is true. Predicate $fluent/3$ depends on dynamic fluent definitions of F, and this dependency indicates that $fluent/3$ is tabled incrementally, to avoid abolishing the table each time a Prolog assertion is made and then recomputing from scratch. Moreover, since $fluent/3$ aims at returning only the latest time F is true (with respect to a given Qt), $fluent/3$ can be tabled using answer subsumption on its second argument. While answer subsumption is shown useful in this approach to avoid recursing through the frame axiom by allowing direct access to the latest time when a fluent is true, it requires $fluent/3$ to have query time Qt as its argument. Consequently, it may hinder reuse of tabled answers of $fluent/3$ by similar goals which differ only in their query time. In truth, the state of a fluent in time depends solely on the changes made to the world, and not on whether that world is being queried. For instance, suppose $fluent(a, 2, 4)$ is already tabled, and fluent a is inertially true till it is supervened by its negated complement, say at time $T = 7$. When a new goal $fluent(a, Ht, 5)$ is posed, it cannot reuse the tabled answer $fluent(a, 2, 4)$, as they differ in their query time. Instead, $fluent(a, Ht, 5)$ unnecessarily recomputes the same solution $Ht = 2$ (recall that fluent a is only retracted at $T = 7$), and subsequently tables $fluent(a, 2, 5)$ as a new answer. A similar situation occurs when $fluent(a, Ht, 6)$ is queried, where $fluent(a, 2, 6)$ is eventually added into the table. This is clearly superfluous, as existing tabled answers could actually be reused and such redundancies avoided, if the tabled answers are independent of query time. However, in XSB answer subsumption on argument Ht cannot be made to ignore argument Qt, by its very design.

In this paper we address the aforementioned issue by fostering further incremental tabling, but leaving out the problematic use of the answer subsumption feature by

reconceptualizing the issue at hand. The main idea, which was not captured in [20], is the perspective that knowledge updates (either self or world wrought changes) occur whether or not they are queried, i.e., the former take place independently of the latter. That is, when a fluent is true at Ht, its truth lingers on independently of query time: Qt no longer becomes an argument of the tabled $fluent$ predicate, i.e., we now have just $fluent(F, Ht)$. Being independent of query time Qt, $fluent/2$ consequently permits better and more general reuse of its tabled answers than that of [20].

In the present approach, fluent updates are initially kept pending in the database, and on the initiative of top-goal queries, i.e., by need only, incremental assertions make these pending updates active (if not already so), but only those with timestamps up to an actual query time. Such assertions automatically trigger system level incremental upwards propagation and tabling of fluent updates. In order to delimit answers in the table, which in some cases may lead to iterative non-termination, the propagation is bounded by a predefined upper global time limit. Though foregoing answer subsumption, recursion through the frame axiom can thus still be avoided, and a direct access to the latest time a fluent is true is made possible via system table inspection predicates. Benefiting from the automatic upwards propagation of fluent updates, the program transformation in the present approach becomes simpler than our previous one, in [20]. Moreover, it demonstrates how the dual program transformation, initially introduced in the context of abduction [3], is employed for helping propagate the dual negation complement of a fluent incrementally, to establish whether the fluent is still true at some time point or rather if its complement is. Keeping both a fluent and its complement tabled will permit in future to address paraconsistency and counterfactuals.

The paper is organized as follows. Section 2 recaps the EVOLP/R language, and reviews the dual transformation and incremental tabling. We detail the implementation technique in Section 3, discuss related work in Section 4, and conclude in Section 5.

2 Preliminaries

We begin by recapitulating the theoretical basis of our logic program updates.

2.1 The EVOLP/R Language

The syntax of EVOLP/R is simply adapted from that of EVOLP [1], by restricting updates to fluents only. Let $\mathcal{K}$ be an arbitrary set of propositional variables and $\tilde{\mathcal{K}}$ be the *extension* of $\mathcal{K}$, defined as $\tilde{\mathcal{K}} = \{A : A \in \mathcal{K}\} \cup \{\sim A : A \in \mathcal{K}\}$. Atoms $A \in \mathcal{K}$ and $\sim A$ are called *positive fluents* and *negative fluents*, respectively. As in EVOLP, program updates are enacted by having the reserved predicate $assert/1$ in the head of a rule.

Definition 1. *Let $\tilde{\mathcal{K}}$ be the extension of a set $\mathcal{K}$ of propositional variables. The* EVOLP/R language $\mathcal{L}$ *is defined inductively as follows:*

1. *All propositional atoms in $\tilde{\mathcal{K}}$ are propositional atoms in $\mathcal{L}$.*
2. *If A is a propositional atom in $\mathcal{L}$, then $assert(A)$ is a propositional atom in $\mathcal{L}$.*
3. *If A is a propositional atom in $\mathcal{L}$, then $\sim assert(A)$ is a propositional atom in $\mathcal{L}$.*
4. *Nothing else is a propositional atom in $\mathcal{L}$.*

5. *If A_0 is a propositional atom in $\mathcal{L}$ and $A_1, \ldots, A_n$, with $n \geq 0$, are literals in $\mathcal{L}$ (i.e. a propositional atom A, or its default negation $not\ A$), then $A_0 \leftarrow A_1, \ldots, A_n$ is a rule in $\mathcal{L}$.*
6. *Nothing else is a rule in $\mathcal{L}$.*

An EVOLP/R program *over a language $\mathcal{L}$ is a (possibly infinite) set of rules in $\mathcal{L}$.*

We extend the notion of positive and negative fluents in $\tilde{\mathcal{K}}$ to propositional atoms A and $\sim A$ in $\mathcal{L}$, respectively. They are said to be *complement* each other. When it is clear from the context, we refer both of them as fluents. Retraction of fluent A (or $\sim A$), making it false, is achieved by asserting its complement $\sim A$ (or A, respectively). I.e., no reserved predicate for retraction is needed. Non-monotonicity of a fluent can thus be admitted by asserting its complement, so as to let the latter supervene the former. Observe that the syntax permits embedded assertions of literals, e.g., $assert(assert(a))$, $\sim assert(assert(a))$; the latter being the complement of the former.

In [1], the semantics of EVOLP is given by a set of *evolution stable models*, each of which is a sequence of interpretations or states. Each evolution stable model describes some possible self-evolution of one initial program after a given number of evolution steps, where each self-evolution is represented by a sequence of generalized logic programs (i.e. programs that allow default negation in their heads). By Definition 1, EVOLP/R programs are not generalized logic programs, but they nevertheless permit negative fluents in the rules' heads. Indeed, one may view negative fluents as explicit negations, and due to the coherence principle [2], that states explicit negation entails default negation, negative fluents obey the principle. Therefore, the two forms of rules' heads, i.e. $assert(not\ A)$ in EVOLP and $assert(\sim A)$ in EVOLP/R, can be treated equivalently. This justification allows the semantics of EVOLP/R to be safely based on that of EVOLP, as long as the paraconsistency of simultaneously having A and $\sim A$ is duly detected and user-defined handled, say with integrity constraints or preferences.

In EVOLP, the most recent rule instances are put in force, and the previous rule instances are valid (by inertia) as far as possible, i.e., they are kept for as long as they do not conflict with more recent ones. Though EVOLP/R restricts updates to fluents only, rule updates (like in EVOLP) can nevertheless be achieved, via the mechanism of rule name fluents, placed in rules' bodies, allowing to turn rules on or off, through assertions or retractions of their corresponding unique name fluents. That said, the restriction amounts to saying that all rules are to be known at the start, so that their rule names can be manipulated. Conceivably however, new internally learnt or externally given rules could be associated at such time with corresponding new names, and the association recorded by an update.

We now review the semantics of EVOLP and adapt it for EVOLP/R, restricting updates to fluents only. In the following definitions, $\bigoplus \mathcal{P}$, where $\mathcal{P} = \{P_i \mid 1 \leq i \leq n\}$, denotes a sequence of EVOLP/R programs $P_1 \oplus \cdots \oplus P_n$; each program corresponding to a state $s \in S$.

Definition 2. *Let $\bigoplus\{P_i : i \in S\}$ be an EVOLP/R program over language $\mathcal{L}$, $s \in S$, and M be a set of propositional atoms of $\mathcal{L}$. Then:*

$Default_s(M) = \{not\ A \mid \nexists A \leftarrow Body \in P_i (1 \leq i \leq s) : M \models Body\}$
$Reject_s(M) = \{A \leftarrow Body \in P_i \mid \exists \sim A \leftarrow Body' \in P_j, i < j \leq s\ \wedge M \models Body'\}$

where $\sim A$ denotes the fluent complement of A, and both $Body$ and $Body'$ are conjunctions of literals.

Definition 3. *Let $P = \bigoplus\{P_i : i \in S\}$ be an EVOLP/R program over language $\mathcal{L}$. A set M of propositional atoms of $\mathcal{L}$ is a stable model of P at state $s \in S$ iff:*

$$M' = least\left(\left[\bigcup\nolimits_{i \leq s} P_i - Reject_s(M)\right] \cup Default_s(M)\right)$$

where $M' = M \cup \{not\_A \mid A \notin M\}$, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default negated literal $not\ A$ by a new atom not_A.

Definition 4. *An* evolution interpretation *of length n of an EVOLP/R program P over $\mathcal{L}$ is a finite sequence $\mathcal{I} = \langle I_1, \ldots, I_n\rangle$ of sets of propositional atoms of $\mathcal{L}$. The* evolution trace *associated with an evolution interpretation $\mathcal{I}$ is the sequence of programs $\langle P_1, \ldots, P_n\rangle$ where $P_1 = P$ and $P_i = \{A \mid assert(A) \in I_{i-1}\}$, for $2 \leq i \leq n$.*

Definition 5. *Let $M = \langle I_1, \ldots, I_n\rangle$ be an evolution interpretation of an EVOLP/R program P and $\langle P_1, \ldots, P_n\rangle$ be its evolution trace. M is an* evolution stable model *of P iff for every i ($1 \leq i \leq n$), I_i is a stable model of $\bigoplus\{P_1, \ldots, P_i\}$ at state i .*

Like EVOLP, besides the self-evolution of a program, EVOLP/R also allows influence from the outside, either as an observation of fluents that are perceived at some state, or assertion orders about fluents on the evolving program. Different from EVOLP, the outside influence in EVOLP/R, referred as *external updates*, persist by inertia as long as they do not conflict with the more recent values for them. Nevertheless, we may easily define external updates that do not persist by inertia, called *events* in EVOLP, by defining for every atomic event E the rule: $assert(\sim E) \leftarrow E$, i.e., if event E is imposed at some state i, then it is no longer assumed from the next state, i.e., $(i + 1)$, onwards. In other words, E holds momentarily at state i only.

Definition 6. *Let E_i, for $1 \leq i \leq k$, be a set of propositional atoms in $\mathcal{L}$. An evolution interpretation $\langle I_1, \ldots, I_n\rangle$, with evolution trace $\langle P_1, \ldots, P_n\rangle$, is an* evolution stable model of P given an external updates sequence $\langle E_1, \ldots, E_k\rangle$ *iff for every i ($1 \leq i \leq n$), I_i is a stable model at state i of $(P_1 \cup E_1) \oplus \cdots \oplus (P_i \cup E_i)$.*

The very idea of the paper is to show how an innovative use of tabling in LP, particularly of incremental tabling, may benefit program updates. Our implementation technique, as detailed Section 3, is realized on top of XSB Prolog, which is based on the well-founded semantics (WFS) [25]. Note that in principle, semantics (with a fixpoint definition) other than stable models can be employed in EVOLP/R. For example, it may alternatively be based on WFS, cf. [4]. Currently, EVOLP/R considers only stratified programs, i.e., programs with no loops over negation. The semantics of EVOLP/R for such programs therefore consists of only one evolution stable model, which is also the well-founded model. This is deliberately so, at this point, because we are concentrating rather on the incremental tabling aspects and usage. Indeed, incremental tabling in the current release of XSB Prolog also supports 3-valued WFS. Its use for non-stratified programs in EVOLP/R, i.e., for updating conditional answers and for reasoning with abduction, is a future line of work, as expressed in the Conclusion section.

2.2 The Dual Program Transformation

The dual program transformation is initially introduced in the context of abduction [3] to abduce explanations under negative goals. It is summarized here and adapted to the EVOLP/R language.

The dual program transformation defines for each atom A and its set of rules R in an EVOLP/R program P, a set of dual rules whose head $\sim A$ is true if and only if A is false by R in the employed semantics of P. It relies on the following definition.

Definition 7. *Let L be a literal in EVOLP/R (cf. Definition 1). The* conjugate $conj(L)$ *of L is defined as follows:*

$$conj(L) = \begin{cases} A & \text{, if } L = not\ A \text{ or } L = \sim A \\ \sim A & \text{, if } L = A \end{cases}$$

Example 1 illustrates the main idea of the dual transformation in EVOLP/R.

Example 1. Consider the following program: $a \leftarrow \sim b. \qquad a \leftarrow c, not\ d.$
The dual transformation creates a set of dual rules for fluent a which falsify a with respect to its two rules, i.e., by falsifying both the first rule *and* the second rule, expressed below by predicate a^{*1} and a^{*2}, respectively:

$$\sim a \leftarrow a^{*1},\ a^{*2}.$$

This single rule is named as the first layer of the dual transformation. The second layer contains the definitions of a^{*1} and a^{*2}, where a^{*1} and a^{*2} are defined by falsifying the body of a's first rule and second rule, respectively; i.e., by taking the conjugate of literals in the body. In case of a^{*1}, the only way the first rule of a can be falsified is by taking the conjugate of $\sim b$. Therefore, we have:

$$a^{*1} \leftarrow b.$$

In case of a^{*2}, the second rule of p is falsified by alternatively failing one subgoal in its body at a time, i.e., by taking the conjugate of c or alternatively, that of $not\ d$.

$$a^{*2} \leftarrow \sim c. \qquad a^{*2} \leftarrow d.$$

Note that, if there is only one definition of a, then the first layer dual rule is defined as $\sim a \leftarrow a^{*1}$. In this case, it is preferable to simply unfold a^{*1}'s definitions in the first layer. For instance, if a in Example 1 is defined only by the second rule, the dual rules $\sim a$ can be directly defined as:

$$\sim a \leftarrow \sim c. \qquad \sim a \leftarrow d.$$

Dual rules can be added to rules expressing falsity in their heads. This means the use of the dual is what actually enables us to incrementally propagate falsity, as well as truth. The reader is referred to [3] for theoretical details, and to [21] for our tabled implementation. Note that use of the dual program transformation does not preclude undefined fluents, and that incremental tabling is compatible with the WFS of XSB.

2.3 Incremental Tabling

Whenever a tabled predicate depends on dynamic predicates and the latter are updated (with Prolog's assert or retract predicates), these updates are not immediately reflected

in the table, i.e., the table becomes out of date. This problem is known as the view maintenance problem in databases and the truth maintenance problem in artificial intelligence. In "classical" tabling, a typical solution to this problem is to rely on the user to explicitly abolish the table whenever a dynamic predicate, on which the table depends, is updated. As several updates may take place on a dynamic predicate, such explicit table abolishment is rather inconvenient and also leads to inefficiency. To overcome this problem, XSB allows maintaining particular tables incrementally, known as *incremental tabling*, i.e., the answers in these tables are ensured to be consistent with all dynamic facts and rules upon which they depend. In XSB, this requires both tabled predicates and the dynamic predicates they depend on to be declared as incremental. For example, if the tabled predicate $r/2$ depends on the dynamic predicate $s/2$, then they are declared as `:- table r/2 as incremental` and `:- dynamic s/2 as incremental`, respectively. To update the table of $r/2$ incrementally by a single change to $s/2$, a call such as $incr\_assert(s(a,3))$ or $incr\_retract(s(a,3))$ can be issued, in which case the table of $r/2$ and other tables that depend on $r/2$ and $s/2$ are updated after such a call. Bulk changes are also supported. The reader is referred to [24] for the further options, examples, and details of incremental tabling.

3 Query-driven Updates Propagation with Incremental Tabling

Since changes by incremental assertions or retractions in incremental tabling update the tables that depend on them, and only those sought – possibly in a chain of dependencies between tabled predicates – this feature can be exploited for automatically propagating the appropriate fluent updates. The use of the frame axiom, with its recursive nature, is thereby avoided. The "world" manages its own consequences, so to speak, and the system provides its history only to the extent needed by queries.

3.1 The Idea

We start with a very simple example to illustrate the basic idea.

Example 2. Consider program P: $b \leftarrow a.$ $c \leftarrow b.$
Given the sequence of external updates $\langle E_1, E_2, E_3 \rangle$, where $E_1 = \{a\}$, $E_2 = \emptyset$, and $E_3 = \{\sim a\}$, the evolution of P in EVOLP/R (cf. definitions in Section 2) is as follows: $P_1 = P$ with $I_1 = \{a, b, c\}$, $P_2 = \emptyset$ with $I_2 = \{a, b, c\}$, and $P_3 = \emptyset$ with $I_3 = \emptyset$.

Observe that a is an external fluent update at state $i = 1$, which propagates to updates of fluents b (by the first rule) and c (by the second rule), making the three fluents true at state $i = 1$. Incremental tabling itself realizes such propagations. A tabled predicate, say $fluent(F, T)$, to record incremental updates of fluent F at state (or time) T is introduced. That is, it depends directly on fluent literals (treated as dynamic incremental predicates), whether extensional or intensional. The external update of fluent a at $i = 1$ is therefore accomplished by an incremental assertion, via $incr\_assert/1$ system predicate, i.e., $incr\_assert(a(1))$ to say that fluent a is incrementally asserted at $i = 1$. Such an incremental assertion results in having entry $fluent(a, 1)$ in the table. Furthermore,

due to the dependencies of the three fluents, as defined by the two rules in P, the incremental assertion of a propagates to fluents b and c, leading to tabling $fluent(b, 1)$ and $fluent(c, 1)$. We thus have $fluent(a, 1)$, $fluent(b, 1)$, and $fluent(c, 1)$, confirming that the three fluents are true at $i = 1$ (cf. I_1).

As there is no update in state $i = 2$, the truths of the three fluents persist by inertia at $i = 2$. From the tabling viewpoint, the previous entries $fluent(a, 1)$, $fluent(b, 1)$, and $fluent(c, 1)$ linger in the table, and a simple check can be performed to verify that the truths of these fluents are not supervened by their complements at $i = 2$. That is, whether there are no $fluent(\sim a, 2)$, $fluent(\sim b, 2)$, and $fluent(\sim c, 2)$ entries in the table, which is indeed the case, and consequently confirms that the three fluents (a, b, and c) are inertially true at $i = 2$ (cf. I_2).

A subsequent update of fluent $\sim a$ at $i = 3$ via $incr\_assert(\sim a(3))$ results in tabling $fluent(\sim a, 3)$. That means, we still have all previous tabled entries, viz., $fluent(a, 1)$, $fluent(b, 1)$, and $fluent(c, 1)$, plus now $fluent(\sim a, 3)$, and a simple state comparison (fluent a at $i = 1$ is supervened by its complement $\sim a$ at a later state $i = 3$) concludes that fluent a is no longer true. Different from before, there is no propagation to fluents $\sim b$ nor $\sim c$ by this incremental assertion, i.e., no $fluent(\sim b, 3)$ and $fluent(\sim c, 3)$ in the table. Indeed, there are no corresponding rules in P for $\sim b$ and $\sim c$; thus failing to conclude that both fluents are also false at $i = 3$ (cf. I_3). We adopt the dual transformation (cf. Section 2.2) to provide rules for $\sim b$ and $\sim c$ from definitions of b and c:

$$\sim b \leftarrow \sim a. \qquad \sim c \leftarrow \sim b.$$

The introduced dual rules now allow the propagations from $\sim a$ to $\sim b$ and then to $\sim c$, resulting in having $fluent(\sim b, 3)$ and $fluent(\sim c, 3)$ in the table. By having the latter two entries in the table, using the same previous reasoning, it can be concluded that fluents b and c are also false at $i = 3$, confirming I_3.

The automatic system level updates propagation, by means of incremental tabling, is driven by a query at a *particular state*, known as a *query time*. Such a query triggers incremental assertions up to the given query time. Indeed, any updates have been kept pending, and only those up to the query time are made actual, if not already so. This mechanism affords us a form of controlled but automatic system level truth-maintenance, up to the given query time. It can be viewed as reconciling a high-level top-down deliberative reasoning (about a query) with low-level bottom-up world reactivity to updates; the latter is relegated to the system enacted incremental tabling feature.

3.2 Implementation

The idea is implemented by a compiled program transformation plus a library of reserved predicates.

Transformation The transformation adds information and rules to program clauses:

1. *Timestamp* that corresponds to state and serves as the only extra argument of fluents. It denotes the time when a fluent is true (known as *holds time* in [20]). Compared to [20], there is no longer the need to carry the query time as an extra argument of fluents. Conceptually, the state of a fluent in time depends solely on the changes made to the world, and is independent of whether that world is being queried.

2. *Rule name* as a special fluent $\$rule(p/n, id_i)$, which identifies a rule of predicate p with arity n by its unique name identity id_i, and is introduced in its body, for checking that the rule is still active.
3. *Dual rules* that are obtained using the dual transformation for each atom with definitions in the input program.

The transformation technique is illustrated by Example 3, with the extra information and rules figuring in the transform ($\$r$ and as in the sequel stand for predicates $\$rule$ and $assert$, respectively). In EVOLP/R, the initial timestamp is set at 1, when a program is inserted. Fluent predicates can be defined as facts (extensional) or by rules (intensional) or both. In Example 3, both fluents b and $as(\sim a)$ are defined intensionally. For such rule regulated intensional fluent instances, unique rule name fluents, i.e., $\$r(b/0, id_1)$ and $\$r(as(\sim a/0), id_1)$ for the first and the second rules, respectively, are introduced. They are extensional fluent instances, and like any other extensional fluent instances, such a rule name fluent is translated (cf. line 1) by adding an extra argument (the third one) that corresponds to its holds time; in this case, each rule name fluent is true at the initial time 1, i.e., the time when its corresponding rule is inserted.

Line 2 shows the translation of rule $b \leftarrow a$ of the input program. The single extra argument in its head is its holds time, H. Call to the goal a in the body is translated into calls to the reserved predicate $fluent/2$ (defined later), that provides their holds time. The subgoal calls $fluent(\$r(b/0, id_1), H_r)$ and $fluent(a, H_a)$ reflect the propagation of the unique rule name fluent $\$r(b/0, id_1)$ and fluent a, respectively, from the body to the head (i.e., fluent b). The holds time H of fluent b in the head is thus determined by which inertial fluent in its body holds latest, via the $latest/2$ reserved predicate (detailed later), assuring that no fluents in the body were subsequently supervened by their complements at some time before H. Note the inclusion of the unique rule name fluent (i.e., the call $fluent(\$r(b/0, id_1), H_r)$) in the body, whose purpose is to switch the corresponding rule on or off.

The other rule of the input program, viz., $as(\sim a) \leftarrow b$, transforms into two rules: the transform in line 5 is similar to that of rule $b \leftarrow a$, whereas the one in line 8 is derived as the effect of asserting $\sim a$. That is, the truth of $\sim a$ is determined solely by the propagation of fluent $as(\sim a)$, indicated by the call $fluent(as(\sim a), H_{as})$. The holds time H of $\sim a$ is thus determined by $H_{as} + 1$ (rather than H_{as}, because $\sim a$ is actually asserted one time step after the time at which $as(\sim a)$ holds). This transform (line 8) is simpler compared to the one in [20] (cf. line 7 of Example 1 in [20]), because no extra reasoning with respect to query time is needed here (due to independence of the transform from query time). Such a simpler transformation consequently corresponds to less computation time: indeed, the extra reasoning with respect to query time Qt, in [20], requires recursively generating timestamps $T < Qt$, and checking via backtracking whether $assert(\sim a)$ holds at T.

Finally, lines 3 and 4 show the dual rules for b. Line 3 expresses how the conjugate $\sim\$r(b/0, id_1)$ of rule name fluent $\$r(b/0, id_1)$ propagates to fluent $\sim b$, whereas line 4 expresses the other alternative: how the conjugate $\sim a$ of a propagates to fluent $\sim b$. Observe that the dual rules are directly defined by unfolding b^{*1}, because b in the input program has only one definition (cf. the last paragraph of Section 2.2). With similar reasoning, lines 6 and 7 define the dual rules for $as(\sim a)$. Recall that dual rules are

defined for each atom with definitions in the input program. Therefore, rules in the transform derived from another rule with $assert/1$ in the head, e.g., rule $\sim a/1$ in line 8 with no definition in the input program, do not have dual rules. From the semantics viewpoint, once $\sim a$ is asserted, its truth remains intact by inertia till superseded, even if $assert(\sim a)$ is retracted at a later time.

Since every fluent occurring in the program is subject to updates, all fluents and their complements should be declared as dynamic and incremental (due to incremental tabling), e.g., `:- dynamic a/1, '~a'/1 as incremental` (the same for fluents b, $as(\sim a)$, $\$r(b/0, id_1, 1)$, $\$r(as(\sim a/0), id_1, 1)$, as well as their complements).

Example 3. Program: $b \leftarrow a.$ $as(\sim a) \leftarrow b.$ transforms into:

1. $\$r(b/0, id_1, 1).$		$\$r(as(\sim a/0), id_1, 1).$
2. $b(H)$	$\leftarrow$	$fluent(\$r(b/0, id_1), H_r),\ fluent(a, H_a),$ $latest([(\$r(b/0, id_1), H_r), (a, H_a)], H).$
3. $\sim b(H)$	$\leftarrow$	$fluent(\sim\$r(b/0, id_1), H).$
4. $\sim b(H)$	$\leftarrow$	$fluent(\sim a, H).$
5. $as(\sim a, H)$	$\leftarrow$	$fluent(\$r(as(\sim a/0), id_1), H_r),\ fluent(b, H_b),$ $latest([(\$r(as(\sim a/0), id_1), H_r), (b, H_b)], H).$
6. $\sim as(\sim a, H)$	$\leftarrow$	$fluent(\sim\$r(as(\sim a/0), id_1), H).$
7. $\sim as(\sim a, H)$	$\leftarrow$	$fluent(\sim b, H).$
8. $\sim a(H)$	$\leftarrow$	$fluent(as(\sim a), H_{as}),\ H$ is $H_{as} + 1.$

Example 4 focuses on the transformation of a rule with a default negation in its body. Apart from the usual rule name fluent in the body, the goal $not\ a$ with default negation translates into a call to reserved predicate $fluent\_not/2$ (defined later), i.e., $fluent\_not(a, H_a)$; cf. line 2. Lines 3 and 4 are the dual rules for fluent b.

Example 4. Program: $b \leftarrow not\ a.$ transforms into:

1. $\$r(b/0, id_1, 1).$		
2. $b(H)$	$\leftarrow$	$fluent(\$r(b/0, id_1), H_r),\ fluent\_not(a, H_a),$ $latest([(\$r(b/0, id_1), H_r), (a, H_a)], H).$
3. $\sim b(H)$	$\leftarrow$	$fluent(\sim\$r(b/0, id_1), H).$
4. $\sim b(H)$	$\leftarrow$	$fluent(a, H).$

Reserved Predicates Predicate $fluent/2$ used in the transformation is a tabled one, as described in Section 3.1. It depends on fluent definitions of F (which are dynamic incremental), and this dependency indicates that $fluent/2$ is tabled incrementally. It is declared as `:- table fluent/2 as incremental`, and defined as follows:

$$fluent(F, T) \leftarrow upper\_time(Lim),\ extend(F, [T], F'),\ call(F'),\ T \leq Lim.$$

where $extend(F, Args, F')$ extends the arguments of fluent F with those in list $Args$ to obtain F'. The definition requires a predefined upper time limit Lim, which is used to delimit updates propagation, i.e., to delimit answers in the $fluent/2$ table. The motivation for such an upper time limit was explained before, plus illustrated in the sequel.

For updates propagation to take place, initial calls $fluent(F, \_)$, for every fluent F, have to be made in order to initially create the table. Once created, the table is incrementally updated after every $incr\_assert/1$ call by propagating updates on which it depends. Updates propagation are controlled in two innovative ways:

1. *Activating pending updates till some query time.*
   In Section 3.1 we mentioned that updates propagation by incremental tabling is query-driven, within some query time of interest. This means we can use the given query time to control updates propagation by keeping the sequence of updates pending, say in the database, and then making active, through incremental assertions, only those with the states up to the actual query time (if they have not yet been so made already by queries of a later time stamp). For so doing, we may introduce a dynamic predicate $pending(F, T)$ to indicate that update of fluent F at state T is still pending, and use Prolog $assert/1$ predicate, i.e., $assert(pending(F, T))$ to assert such a pending fluent update into the Prolog database. Activating pending updates (up to the given query time Qt), as shown by the code below, can thus be done by calling all $pending(F, T)$ facts with $T \leq Qt$ from the database and actually asserting them incrementally using the system $incr\_assert/1$ predicate:

   $$activate\_pending(Qt) \leftarrow pending(F, T),\ T \leq Qt,\ extend(F, [T], F'),\ incr\_assert(F'),\ retract(pending(F, T)), fail.$$
   $$activate\_pending(\_).$$

   Note that a quasi forward-chaining approach [24] of incremental update through the use of $incr\_assert/1$ is employed, as opposed to the use of $incr\_assert\_inval/1$ system predicate of eager and lazy incremental update approaches [24]. Nevertheless, since pending updates are only made active on the initiative of top-goal queries, only those with timestamps up to an actual query time are actually asserted, i.e., by need only. Lazy evaluation by itself would not suffice to delimit actual updates to query time ceilings, and hence the need for pending updates.
2. *Limiting updates propagation to a predefined upper time limit.*
   Activating pending updates up to some query time does not guarantee termination of updates propagation, as Example 5 illustrates.

   Example 5. Consider program P: $as(\sim a) \leftarrow a.$ $as(a) \leftarrow \sim a.$
   Given an external update $\langle E_1 \rangle$, where $E_1 = \{a\}$, the evolution of P in EVOLP/R is as follows: $P_1 = P$ with $I_1 = \{a, assert(\sim a)\}$, $P_2 = \{\sim a\}$ with $I_2 = \{\sim a, assert(a)\}$, $P_3 = \{a\}$ with $I_3 = \{a, assert(\sim a)\}$, $P_4 = \{\sim a\}$ with $I_4 = \{\sim a, assert(a)\}$, ... etc. (the evolution continues indefinitely)

   In this example the external update of a at state $i = 1$ leads to non-terminating propagation. From the incremental tabling viewpoint, it indicates that a predefined upper time limit is required to limit updates propagation, thereby avoiding infinite number of answers in the $fluent/2$ table. This requirement is realistic, as our view into the future may be bounded by some time horizon, comparable to bounded rationality. For this purpose, a dynamic predicate $upper\_time(Lim)$ is introduced to indicate the predefined upper time limit Lim, and used in the above $fluent/2$

definition to time-delimit their tabled answers. In the case of Example 5, by setting, e.g., $upper\_time(4)$, the $fluent/2$ table contains a finite number of answers: $fluent(a, 1)$, $fluent(\sim a, 2)$, $fluent(a, 3)$, and $fluent(\sim a, 4)$.

We have seen predicate $latest([(F_1, H_1), \ldots, (F_n, H_n)], H)$ in the transformation, which appears in the body of a rule transform, say of fluent F. This reserved predicate is responsible for obtaining the latest holds time H of F amongst fluents $F_1, \ldots, F_n$ in the body, while also assuring that none of them were subsequently supervened by their complements at some time up to H. It is defined as:

$$latest(Fs, H) \leftarrow greatest(Fs, H),\ not\_supervened(Fs, H).$$

where $greatest(Fs, H)$ extracts from list Fs, of (F_i, H_i) pairs with $1 \leq i \leq n$, the greatest holds time H among the Hi's, and predicate $not\_supervened(Fs, H)$ subsequently checks, by means of table inspection, that there is no fluent complement F_i' (with holds time H_i') of F_i in Fs, such that $H_i < H_i' \leq H$.

Recall now Example 4. There, reserved predicate $fluent\_not/2$ is introduced. Its definition is given below:

$$\begin{array}{l}(1)\ fluent\_not(F, T) \leftarrow compl(F, F'),\ fluent(F', H).\\ (2)\ fluent\_not(F, T) \leftarrow nonvar(T),\ !,\ fail.\\ (3)\ fluent\_not(\_, 0).\end{array}$$

where $compl(F, F')$ obtains the fluent complement F' from F. Rule (1) captures the coherence principle [2], that states explicit negation entails default negation; in our case, negative fluents are treated as explicit negations, therefore they obey the principle. Rules (2) and (3) are the standard definition of default negation. Note that rule (3) artificially sets the timestamp to $T = 0$ for all fluents; for none are by then (before the "Big Bang" of the starting program update, which initially starts at $T = 1$) known to be true.

Given that an upper time limit has been set, and that the initial calls $fluent(F, \_)$ for every fluent F have been made, and that some pending updates may be available, the EVOLP/R system is ready for a top-goal query. The top-goal query $holds(F, Qt)$ verifies whether fluent F is true at query time Qt within the bounded time horizon (otherwise it is undefined). It does so by first activating pending updates up to Qt and then inspecting $fluent/2$ table to answer the query:

$$\begin{array}{ll}(1)\ holds(\_, Qt) & \leftarrow upper\_time(Lim),\ (Qt > Lim\ ;\ Qt \leq 0),\ !,\ undefined.\\ (2)\ holds(not\ F, Qt) & \leftarrow !,\ not\ holds(F, Qt).\\ (3)\ holds(F, Qt) & \leftarrow activate\_pending(Qt),\ compl(F, F'),\ inspect(F, H, Qt),\\ & \quad (H \neq 0 \rightarrow (inspect(F', H', Qt),\ H \geq H')\ ;\ fail).\end{array}$$

where $inspect(F, H, Qt)$ inspects the $fluent/2$ table and looks for entries of fluent F with the highest timestamp $H \leq Qt$. XSB provides various table inspection predicates, e.g., $get\_returns\_for\_call/2$ may be used. If there is no such fluent F in the table, $H = 0$ is returned, making $holds(F, Qt)$ fail, due to the last conditional subgoal in the body. Otherwise, this conditional goal exercises the table inspection of its complement fluent F' to obtain its highest timestamp H', and succeeds only if $H \geq H'$, i.e., checks

that fluent F is not supervened at a later time by its complement F'. Note that this allows for paraconsistency (case $H = H'$), to be dealt by the user as desired, e.g., by integrity constraints or preferences, but this matter is beyond the scope of the paper.

Example 6. Recall Example 3, which is loaded initially at time 1. Suppose that the upper time limit is set to $upper\_limit(5)$, and calls $fluent(F, \_)$ and $fluent(F', \_)$, where F' is the complement of F, have been made for every fluent F in the transform, i.e., $F = \{a, b, as(\sim a), \$r(b/0, id_1, 1), \$r(as(\sim a/0), id_1, 1)\}$. Note that, because rule name fluents are already inserted (as fluent facts) in the program (cf. line 1 of Example 3), these $fluent/2$ calls result in having entries $fluent(\$r(b/0, id_1), 1)$ and $fluent(\$r(as(\sim a/0), id_1), 1)$ in the table. Now, assume further that two pending external updates are available, viz., $pending(a, 1)$ and $pending(b, 4)$, that correspond to external updates of fluent a at $i = 1$ and fluent b at $i = 4$, respectively. In other words, $\langle E_1, E_2, E_3, E_4\rangle$ is the external updates sequence with $E_1 = \{a\}$, $E_2 = E_3 = \emptyset$, and $E_4 = \{b\}$. The following queries show that their answers conform to the evolution model of the program given the above external updates sequence:

1. When $holds(b, 1)$ is queried, it first activates pending updates up to $Qt = 1$, via subgoal $activate\_pending(1)$, thereby incrementally asserting $a(1)$ only, and keeping $pending(b, 4)$ still intact. The incremental assertion of $a(1)$ results in having $fluent(a, 1)$ in the table, and henceforth propagates to update fluents b (by rule 2), $as(\sim a)$ (by rule 5), $\sim a$ (by rule 8), $\sim b$ (by rule 4), and $\sim as(\sim a)$ (by rule 7). These make $fluent(b, 1)$, $fluent(as(\sim a), 1)$, $fluent(\sim a, 2)$, $fluent(\sim b, 2)$, and $fluent(\sim as(\sim a), 2)$ added into the table. When subgoal $inspect(b, H, 1)$ of $holds(b, 1)$ is called, it returns $H = 1$, and since $H \neq 0$, call $inspect(\sim b, H', 1)$ is subsequently made, in which case $H' = 0$ is returned (no $fluent(\sim b, H')$ with $H' \leq 1$ in the table). This eventually makes $holds(b, 1)$ succeed, because condition $H \geq H'$ in the definition of $holds/2$ is satisfied.
2. A similar reasoning applies when $holds(b, 2)$ is queried, but now no more pending updates up to $Qt = 2$ are available. The subgoal calls $inspect(b, H, 2)$ returns $H = 1$ and $inspect(\sim b, H', 2)$ returns $H' = 2$, in which case the condition $H \geq H'$ is unsatisfied, and therefore $holds(a, 2)$ fails, i.e., fluent a does not hold at state $i = 2$.
3. It is easy to confirm, that query $holds(b, 3)$ still fails. Indeed, it persists by inertia.
4. Finally, when $holds(b, 4)$ is queried, the only pending update $pending(b, 4)$ is made active by incrementally asserting $b(4)$ and tabling $fluent(b, 4)$. This propagates to adding several entries into the table: $fluent(as(\sim a), 4)$, $fluent(\sim a, 5)$, $fluent(\sim b, 5)$, and $fluent(\sim as(\sim a), 5)$. Therefore, subgoal call $inspect(b, H, 4)$ now returns $H = 4$, call $inspect(\sim b, H', 2)$ still returns $H' = 2$, and $H \geq H'$ is satisfied, making $holds(b, 4)$ succeed.
5. With the current entries in the $fluent/2$ table, one may verify that $holds(b, 5)$ fails.

4 Related Work

Many Prolog systems are nowadays adopting tabling, though none has gone as far as XSB Prolog, namely in allowing tabling over default negation, and providing together answer subsumption, incremental tabling, and threads with shared tables. Consequently,

there are also limited applications of these features, particularly of incremental tabling. Known applications are in pointer analyses of C programs in the context of incremental program analyses [18], data flow analyses [19], static analyses [6], incremental validation of XML documents and push down model checking [17]. But we are not aware of any work on employing incremental tabling for logic program updates as we do here.

Updates propagation has been well studied in the field of deductive databases, e.g., [5, 7, 13]. Similar to what we do here, updates propagation in these works aims at computing implicit changes of derived relations caused by explicit updates of extensional facts. Methods in updates propagation consist of bottom-up and top-down approaches. In [5], both approaches are combined, sharing the same basic idea with ours, i.e., to control bottom-up propagation with a top-down evaluation strategy. But different from ours, it does not use any Prolog tabling features, particularly incremental tabling, but employs instead the Magic Sets approach. Others, like [13], employ a purely top-down approach by querying the relevant portion of the database, whereas [7] focuses on bottom-up methods of updates propagation.

Logic-based Production System (LPS) with abduction [11] is a distinct but somewhat similar and complementary approach to ours. It aims at defining a new logic-based framework for knowledge representation and reasoning, relying on the fundamental role of state transition systems in computing, and involving fluent updates by destructive assignment. It is implemented in LPA Prolog [14] but no details are given about that. Their approach differs from ours in that it defines a new language and an operational semantics, rather than taking an existing one, and implements it on a commercial system (LPA Prolog) with no underlying tabling mechanisms. Moreover, in our work fluent updates are not managed by destructive database assignments, but rather tabled, thereby allowing to inspect their truths at a particular time, e.g., querying the past. Furthermore, full knowledge about each fluent in each state is not presupposed, so that only those fluents are updated for which changes are known about. Subsequent knowledge, say about updates on the world by another agent, or by yet unmeasured world processes, may change the picture of the world to a more complete one. In any case, the emergence and propagation of changes prepare the way for the wider topic of teleo-reactive systems [12, 15].

Regarding other related work, the use of incremental tabling in this paper is very strongly related to the (also incremental, and also tabling-based) algorithms employed in the compile-time analyses of logic programs (e.g., [10] and other connected papers). It could be interesting to compare the algorithms in incremental analysis and our work (which relies on the underlying incremental tabling algorithms of XSB), because some techniques used in incremental analyses might be useful in the context of incremental tabling (and vice-versa). Incremental analyses also table answers (like answers of $fluent/2$ in our work) and include algorithms to incrementally add, delete or modify a clause of a predicate. Furthermore, there exist several specific optimizations and techniques used in these incremental analysis algorithms which may be beneficial in the context of the tabling procedure proposed here, namely: (1) Being cautious about changes in the database that only affect a small subset of it (called local change in Section 5.1 of [10]); (2) Whereas we present in Example 5 a case in which propagation does not terminate, and solve such cases by delimiting propagation to a predefined

upper time limit, it may be opportune to consider operators similar to the widening operator of abstract interpretation, which lose precision on the tabling (possibly leading to answers being recomputed), but ensure termination.

Our approach to limit updates propagation, using a predefined time limit as a bound, has the same overall purpose as XSB's recent tabling feature: answer abstraction [8], i.e., to guarantee termination in tabling by ensuring that only a finite number of answers are generated by a query. In answer abstraction, this is achieved via a form of bounded rationality, viz., radial restraint, and is realized by bounding the depth of an answer.

5 Conclusion

We have propounded in detail an implementation technique to logic program updates by further exploiting incremental tabling in logic programming (available in XSB Prolog), which enriches the applicability of the incremental tabling feature to dynamic environments and evolving systems, and that might be adopted elsewhere for reasoning in logic. The implementation technique proposed much refines our previous approach by leaving out the answer subsumption feature that was heretofore employed to address the frame problem. Instead, we rely fully on incremental tabling by separating knowledge updates from queries on them; the former takes place independently from the latter. Incremental tabling allows updates propagation, which is controlled by initially keeping updates pending and making active only those with timestamp up to an actual query time, on the initiative of queries. Possible non-terminating updates propagation is avoided by setting a predefined upper time limit for queries, and the direct access to the latest time a fluent is true is achieved by table inspection predicates. Moreover, we adopt the dual transformation from abduction and adapt it for helping propagate also the complement of fluents incrementally. In summary, our approach affords us a form of controlled (i.e., query-driven) but automatic system level truth-maintenance (i.e., automatic updates propagation via incremental tabling), up to actual query time.

Our future work consists of integrating tabled abduction [21] with EVOLP/R, so as to jointly afford abduction and updating in one integrated XSB system. We intend to apply the system to abductive moral reasoning [9], with updating and argumentation, as a sequel to our ongoing approach to using logic for reasoning.

Acknowledgements Ari Saptawijaya acknowledges the support of Fundação para a Ciência e a Tecnologia (FCT/MEC) Portugal, grant SFRH/BD/72795/2010. We thank the anonymous reviewers for their helpful suggestions.

References

1. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In *JELIA 2002*, volume 2424 of *LNCS*, pages 50–61. Springer, 2002.
2. J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Springer, Berlin, 1996.
3. J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, 2004.

4. F. Banti, J. J. Alferes, and A. Brogi. Well founded semantics for logic program updates. In *IBERAMIA 2004*, volume 3315 of *LNCS*, pages 397–407, 2004.
5. A. Behrend and R. Manthey. Update propagation in deductive databases using soft stratification. In *ADBIS 2004*, volume 3255 of *LNCS*, pages 22–36. Springer, 2004.
6. M. Eichberg, M. Kahl, D. Saha, M. Mezini, and K. Ostermann. Automatic incrementalization of Prolog based static analyses. In *PADL 2007*, volume 4354 of *LNCS*, pages 109–123. Springer, 2007.
7. U. Griefahn. *Reactive Model Computation - A Uniform Approach to the Implementation of Deductive Databases*. PhD thesis, University of Bonn, 1997.
8. B. N. Grosof and T. Swift. Radial restraint: A semantically clean approach to bounded rationality for logic programs. In *AAAI 2013*. The AAAI Press, 2013.
9. T. A. Han, A. Saptawijaya, and L. M. Pereira. Moral reasoning under uncertainty. In *LPAR-18*, volume 7180 of *LNCS*, pages 212–227. Springer, 2012.
10. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, 2000.
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. R. Kowalski and F. Sadri. Teleo-reactive abductive logic programs. In *Logic Programs, Norms and Action (Festschrift of Marek Sergot)*, volume 7360 of *LNCS*, 2012.
13. V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In *DOOD 1991*, volume 566 of *LNCS*. Springer, 1991.
14. Logic Programming Associates Ltd. LPA prolog. `http://www.lpa.co.uk/`.
15. N. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.
16. D. L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.
17. D. Saha. *Incremental Evaluation of Tabled Logic Programs*. PhD thesis, SUNY Stony Brook, 2006.
18. D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *ACM PPDP 2005*, pages 117–128. ACM, 2005.
19. D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of tabled logic programs. In *ICLP 2006*, volume 4079 of *LNCS*, pages 56–71. Springer, 2006.
20. A. Saptawijaya and L. M. Pereira. Program updating by incremental and answer subsumption tabling. In *LPNMR 2013*, volume 8148 of *LNCS*. Springer, 2013.
21. A. Saptawijaya and L. M. Pereira. Tabled abduction in logic programs (technical communication of ICLP 2013). *Theory and Practice of Logic Programming, Online Supplement*, 13(4-5), 2013.
22. T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *JELIA 2010*, volume 6341 of *LNCS*, pages 300–312. Springer, 2010.
23. 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.
24. 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.
25. A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.