TLP **17** (5–6): 744–763, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000370 First published online 23 August 2017

*Ticker: A system for incremental ASP-based stream reasoning**

HARALD BECK, THOMAS EITER and CHRISTIAN FOLIE
Institute of Information Systems, Vienna University of Technology,
Favoritenstraße 9-11, A-1040 Vienna, Austria
(*e-mails:* beck@kr.tuwien.ac.at, eiter@kr.tuwien.ac.at, christian.folie@outlook.com)

submitted 17 July 2017; revised 20 July 2017; accepted 27 July 2017

Abstract

In complex reasoning tasks, as expressible by Answer Set Programming (ASP), problems often permit for multiple solutions. In dynamic environments, where knowledge is continuously changing, the question arises how a given model can be incrementally adjusted relative to new and outdated information. This paper introduces Ticker, a prototypical engine for well-defined logical reasoning over streaming data. Ticker builds on a practical fragment of the recent rule-based language LARS, which extends ASP for streams by providing flexible expiration control and temporal modalities. We discuss Ticker's reasoning strategies: first, the repeated one-shot solving mode calls Clingo on an ASP encoding. We show how this translation can be incrementally updated when new data is streaming in or time passes by. Based on this, we build on Doyle's classic justification-based truth-maintenance system to update models of non-stratified programs. Finally, we empirically compare the obtained evaluation mechanisms.

KEYWORDS: Stream reasoning, Answer set programming, Non-monotonic reasoning.

1 Introduction

Stream reasoning (Della Valle *et al.* 2009) as research field emerged from data processing (Babu and Widom 2001), i.e., the handling of continuous queries in a frequently changing database. Work in Knowledge Representation & Reasoning, e.g., Ren and Pan (2011), Gebser *et al.* (2015), shifts the focus from high throughput to high expressiveness of declarative queries and programs. In particular, the logic-based framework LARS (Beck *et al.* 2015) was defined as an extension of Answer Set Programming (ASP) with window operators for deliberately dropping data, e.g., based on time or counting atoms, and controlling the temporal modality in the resulting windows.

When dealing with complex reasoning tasks in stream settings, one may in general not afford to recompute models fromscratch every time new data comes in or when

* This research has been supported by the Austrian Science Fund (FWF) projects P26471 and W1255-N23.

older portions of data become outdated. Besides the pragmatic need for efficient computation, there is also a semantic issue: while aspects of a solution might have to change dynamically and potentially quickly, typically not everything should be reconstructed from scratch, but adapted to fit the current data.

Recently, many stream-processing tools and reasoning features have been proposed, e.g., Barbieri *et al.* (2010), Phuoc *et al.* (2011) and Gebser *et al.* (2014). However, an ASP-based stream reasoning engine that supports window operators and has an incremental model update mechanism is lacking to date. This may be explained by the fact that non-monotonic negation, beyond recursion, makes efficient incremental update non-trivial; combined with temporal reasoning modalities over data windows, this becomes even more challenging.

Contributions. We tackle this issue and make the following contributions.

(1) We present a notion of tick streams to formally represent the sequential steps of a fully incremental stream reasoning system.
(2) Based on this, we give an intuitive translation of a practical fragment of LARS programs, plain LARS, to ASP suitable for standard one-shot solving, and in particular, stratified programs.
(3) Next, we develop an ASP encoding that can be incrementally updated when time passes by or when new input arrives.
(4) We then present Ticker, our prototype reasoning engine that comes with two reasoning strategies. One utilizes Clingo (Gebser *et al.* 2014) with a static ASP encoding, the other truth maintenance techniques (Doyle 1979) to adjust models based on the incremental encoding.
(5) Finally, we experimentally compare the two reasoning modes in application scenarios. The results demonstrate the performance benefits that arise from incremental evaluation.

In summary, we provide a novel technique for adjusting an ASP-based stream reasoning program by time and data streaming in. In particular, the update technique of the program is independent of the model update technique used to process the program change. Supplementary material accompanying the paper can be found at the TPLP archive.

2 Stream reasoning in LARS

We will gradually introduce the central concepts of LARS (Beck *et al.* 2015) tailored to the considered fragment. If appropriate, we give only informal descriptions.

Throughout, we distinguish *extensional atoms* A^E for input data and *intensional atoms* A^I for derived information. By $A = A^E \cup A^I$, we denote the set of *atoms*.

Definition 1 (*Stream*)
A stream $S = (T, \upsilon)$ consists of a *timeline* T, which is a closed non-empty interval in $\mathbb{N}$, and an *evaluation function* $\upsilon : \mathbb{N} \mapsto 2^A$. The elements $t \in T$ are called *time points*.

Intuitively, a stream S associates with each time point a set of atoms. We call S a *data stream*, if it contains only extensional atoms. To cope with the amount of

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

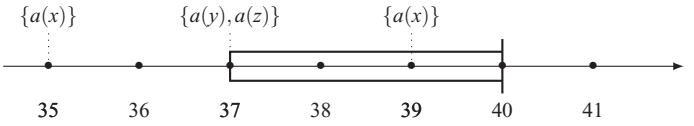


Fig. 1. Temporal extent of a sliding tuple-based window of size 3 (or 2) at $t = 40$.

data, one usually considers only recent atoms. Let $S = (T, \upsilon)$ and $S' = (T', \upsilon')$ be two streams such that $S' \subseteq S$, i.e., $T' \subseteq T$ and $\upsilon'(t') \subseteq \upsilon(t')$ for all $t' \in T'$. Then, S' is called a *window* of S.

Definition 2 (*Window function*)
Any (computable) function w that returns, given a stream $S = (T, \upsilon)$ and a time point $t \in \mathbb{N}$, a *window* S' of S is called a *window function.*

Time-based window functions, which select all atoms appearing in last n time points, and *tuple-based* window functions, which select a fixed number of latest tuples are widely used. To this end, we define the *tuple size* $|S|$ of a stream $S = (T, \upsilon)$ as $|\{(a, t) \mid t \in T, a \in \upsilon(t)\}|$.

Definition 3 (*Sliding time-based and tuple-based window*)
Let $S = (T, \upsilon)$ be a stream, $t \in T = [t_1, t_m]$ and let $n \in \mathbb{N} \cup \{\infty\}$. Then,

(i) the *sliding time-based window function* τ_n *(for size n)* is $\tau_n(S, t) = (T', \upsilon|_{T'})$, where $T' = [t', t]$ and $t' = \max\{t_1, t - n\}$;
(ii) the *sliding tuple-based window function* $\#_n$ *(for size n)* is

$$\#_n(S, t) = \begin{cases} \tau_{t-t'}(S, t) & \text{if } |\tau_{t-t'}(S, t)| \leqslant n, \\ S' & \text{else,} \end{cases}$$

where $t' = \max(\{u \in T \mid |\tau_{t-u}(S, t)| \geqslant n\} \cup \{t_1\})$ and $S' = ([t', t], \upsilon')$ has tuple size $|S'| = n$ such that $\upsilon'(u) = \upsilon(u)$ for all $u \in [t' + 1, t]$ and $\upsilon'(t') \subseteq \upsilon(t')$.

Note that, in general, multiple options exist for defining υ' at t' in the tuple-based window. However, we assume a deterministic choice as specified by the implementation of the function. In particular, we will later consider that atoms are streaming in an order, which leads to a natural, unique cut-off position based on counting.

Example 1
Figure 1 window depicts at partial stream $S = ([35, 41], \upsilon)$, where $\upsilon = \{35 \mapsto \{a(x)\}, 37 \mapsto \{a(y), a(z)\}, 39 \mapsto \{a(x)\}\}$, and a time window of length 3 at time $t = 40$, which corresponds to a tuple window of size 3 there. Notably, there are two options for a tuple window of size 2, both of which select timeline $[37, 40]$, but only one of the atoms at time 37, respectively.

We also use window functions with streams as single argument, applied implicitly at the end of the timeline, i.e., if $S = ([t_0, t], \upsilon)$, then $\tau_n(S)$ abbreviates $\tau_n(S, t)$ and $\#_n(S)$ stands for $\#_n(S, t)$.

Window operators $\boxplus^w$. A window function w can be accessed in rules by window operators. That is to say, an expression $\boxplus^w \alpha$ has the effect that α is evaluated on

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

the "snapshot" of the data stream delivered by its associated window function w. Within the selected snapshot, LARS allows for controlling the temporal semantics with further modalities.

Temporal modalities. Let $S = (T, \upsilon)$ be a stream, $a \in A$ and $\mathrm{B} \subseteq A$ static *background data*. Then, at time point $t \in T$,

- a holds, if $a \in \upsilon(t)$ or $a \in \mathrm{B}$;
- $\Diamond a$ holds, if a holds at some time point $t' \in T$;
- $\Box a$ holds, if a holds at all time points $t' \in T$; and
- $@_{t'} a$ holds, where $t' \in \mathbb{N}$, if $t' \in T$ and a holds at t'.

The set A^+ of *extended atoms* e is given by the grammar $e ::= a \mid @_t a \mid \boxplus^w @_t a \mid \boxplus^w \Diamond a \mid \boxplus^w \Box a$, where $a \in A$ and t is any time point. The expressions $@_t a$ are called *@-atoms*; $\boxplus^w \star a$, where $\star \in \{@_t, \Diamond, \Box\}$ are *window atoms*. We write $\boxplus^n$ for $\boxplus^{\tau_n}$, which is not to be confused with $\boxplus^{\#n}$.

Example 2 (*cont'd*)
At $t = 40$, $\boxplus^3 \Diamond a(x)$ and $\boxplus^3 @_{37} a(y)$ hold, as does $\boxplus^{\#1} \Box a(x)$ at $t = 35, 39$.

2.1 Plain LARS programs

We use a fragment of the formalism in Beck *et al.* (2015), called *plain LARS* programs.

Syntax. A *(ground plain LARS) program* P is a set of rules of the form

$$\alpha \leftarrow \beta_1, \ldots, \beta_j, \mathrm{not}\, \beta_{j+1}, \ldots, \mathrm{not}\, \beta_n, \tag{1}$$

where the *head* α is of form a or $@_t a$, $a \in A^I$, and in the *body* $\beta(r) = \beta_1, \ldots, \beta_j, \mathrm{not}\, \beta_{j+1}, \ldots, \mathrm{not}\, \beta_n$ each β_i is an extended atom. We let $H(r) = \alpha$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_j\}$ and $B^-(r) = \{\beta_{j+1}, \ldots, \beta_n\}$ are the the *positive*, resp. *negative body atoms* of r.

Semantics. For a data stream $D = (T_D, \upsilon_D)$, any stream $I = (T, \upsilon) \supseteq D$ that coincides with D on A^E, i.e., $a \in \upsilon(t) \cap A^E$ iff $a \in \upsilon_D(t)$, is an *interpretation stream* for D. A tuple $M = \langle I, W, \mathrm{B} \rangle$, where W is a set of window functions and B is the background knowledge, is then an *interpretation* for D. Throughout, we assume $W = \{\tau_k, \#_n \mid k, n \in \mathbb{N}\}$ and B are fixed and also omit them.

Satisfaction by M at $t \in T$ is as follows: $M, t \models \alpha$ for $\alpha \in A^+$, if α holds in (T, υ) at time t; $M, t \models r$ for rule r, if $M, t \models \beta(r)$ implies $M, t \models H(r)$, where $M, t \models \beta(r)$, if (i) $M, t \models \beta_i$ for all $i \in \{1, \ldots, j\}$ and (ii) $M, t \not\models \beta_i$ for all $i \in \{j{+}1, \ldots, n\}$; and $M, t \models P$ for program P, i.e., M is a *model* of P (for D) at t, if $M, t \models r$ for all $r \in P$. Moreover, M is *minimal*, if in addition no model $M' = \langle S', W, \mathrm{B} \rangle \neq M$ of P exists such that $S' = (T, \upsilon')$ and $\upsilon' \subseteq \upsilon$.

Definition 4 (*Answer Stream*)
An interpretation stream I is an answer stream of program P for the data stream $D \subseteq I$ at time t, if $M = \langle I, W, \mathrm{B} \rangle$ is a minimal model of the *reduct* $P^{M,t} = \{r \in P \mid M, t \models \beta(r)\}$. By $AS(P, D, t)$, we denote the set of all such answer streams I.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Example 3 (cont'd)
Consider D from Figure 1 and $P = \{b(x) \leftarrow \boxplus^3 \diamond a(x)\}$. Then, for all $t \in [35, 41]$, the answer stream I at t is unique and adds to D the mapping $t \mapsto \{b(x)\}$.

Non-ground programs. The semantics for LARS is formally defined for ground programs but extends naturally for the non-ground case by considering the respective ground instantiations.

Windows on intensional/extensional atoms. For practical reasons, we consider tuple windows only on extensional data. Their intended use is counting input data, not inferences; using them on intensional data is conceptually questionable.

Example 4
Consider the rule $r = b \leftarrow \boxplus^{\#1} \diamond a$ and the stream $S = ([0, 1], \{0 \mapsto \{a\}\})$, which is not a model for r, since the rule fires and we thus must have b at time 1. However, in this interpretation, $\boxplus^{\#1} \diamond a$ does not hold any more, if we also take into account the inference b. Thus, the interpretation would not be minimal. Moreover, further inferences would not be founded. Hence, program $\{r\}$ has no model.

In contrast to tuple windows, time windows are useful and allowed on arbitrary data, as long as no cyclic positive dependencies through time-based window atoms $\boxplus^n \Box a$ occur.

Example 5
Assume a range of values $V = 0, \ldots, 30$, among which $V \geqslant 18$ are considered "high." To test whether the predicate *alpha* always had a high value during the last n time points, we first abstract by $@_T high \leftarrow \boxplus^n @_T alpha(V), V \geqslant 18$ for and then test $yes \leftarrow \boxplus^n \Box high$.

3 Static ASP encoding

In this section, we will first give a translation of LARS programs P to an ASP program $\hat{P}$. Toward incremental evaluation of P, we will then show how $\hat{P}$ can be adjusted to accommodate new input signals and account for expiring information as specified by window operators.

Definition 5 (Tick)
A pair $k = (t, c)$, where $t, c \in \mathbb{N}$, is called a *tick*, with t the *(tick) time* and c the *(tick) count*; $(t + 1, c)$ is called the *time increment* and $(t, c + 1)$ the *count increment* of k. A sequence $K = \langle k_1, \ldots, k_m \rangle$, $m \geqslant 1$, of ticks is a *tick pattern*, if every tick k_{i+1} is either a time increment or a count increment of k_i.

Intuitively, a tick pattern captures the incremental development of a stream in terms of time and tuple count, where at each step exactly one dimension increases by 1. For a set of ticks, at most one linear ordering yields a tick pattern. Thus, we can view a tick pattern K also as set.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Definition 6 (*Tick stream*)
A *tick stream* is a pair $\dot{S} = (K, v)$ of a tick pattern K and an *evaluation function* v s.t. $v(k_{i+1}) = \{a\}$ for some $a \in A$, if k_{i+1} is a count increment of k_i, else $v(k_{i+1}) = \emptyset$.

We say that a tick stream $\dot{S} = (K, v)$ with $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$ is *at tick* (t_m, c_m). By default, we assume $(t_1, c_1) = (0, 0)$, and thus c_m is the total number of atoms. We also write $v(t, c)$ instead of $v((t, c))$. Naturally, a *(tick) substream* $\dot{S}' \subseteq \dot{S}$ is a tick stream (K', v'), where K' is a subsequence of K and v' is the restriction $v|_{K'}$ of v to K', i.e., $v'(t, c) = v(t, c)$ if $(t, c) \in K'$, else $v'(t, c) = \emptyset$.

Example 6
The sequence $K = \langle (0,0), (1,0), (2,0), (3,0), (3,1), (3,2), (4,2) \rangle$ is a "canonical" tick pattern starting at $(0, 0)$, where $(3, 1)$ and $(3, 2)$ are the only count increments. Employing an evaluation $v(3, 1) = \{a\}$ and $v(3, 2) = \{b\}$, we get a tick stream $\dot{S} = (K, v)$ which is at tick $(4, 2)$.

Definition 7 (*Ordering*)
Let $\dot{S} = (K, v)$ be a tick stream, where $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$, and let $S = (T, \upsilon)$ be a stream such that $T = [t_1, t_m]$ and $\upsilon(t) = \bigcup \{v(t, c) \mid (t, c) \in K\}$ for all $t \in T$. Then, we say $\dot{S}$ is an *ordering* of S, and S *underlies* $\dot{S}$.

Note that, in general, a stream S has multiple orderings, but every tick stream $\dot{S}$ has a unique underlying stream. All orderings of a stream have the same tick pattern.

Example 7 (*cont'd*)
Stream $S = ([0, 4], \upsilon)$, where $\upsilon = \{3 \mapsto \{a, b\}\}$, is the underlying stream of $\dot{S}$ of Example 6. A further ordering of S is $\dot{S}' = (K, v')$, where $v' = \{(3, 1) \mapsto \{b\}, (3, 2) \mapsto \{a\}\}$.

Sliding windows, as in Definition 3, carry over naturally for tick streams. There are two central differences. First, ticks replace time points as positions in a stream, and thus as second argument of the window functions. Second, tuple-based windows are now always unique.

Definition 8 (*Sliding windows over tick streams*)
Let $\dot{S} = (K, v)$ be a tick stream, where $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$ and $(t, c) \in K$. Then, the *time window function* τ_n, $n \geqslant 0$, is defined by $\tau_n(\dot{S}, (t, c)) = (K', v|_{K'})$, where $K' = \{(t', c') \in K \mid \max\{t_1, t - n\} \leqslant t' \leqslant t\}$, and the *tuple window function* $\#_n$, $n \geqslant 1$, by $\#_n(\dot{S}, (t, c)) = (K', v|_{K'})$, where $K' = \{(t', c') \in K \mid \max\{c_1, c - n + 1\} \leqslant c' \leqslant c\}$.

As for Definition 3, we consider windows over tick streams also implicitly at the end of the timeline.

Lemma 1
If stream S underlies tick stream $\dot{S}$, then $\tau_n(S)$ underlies $\tau_n(\dot{S})$.

Example 8 (*cont'd*)
Given $\dot{S}$ and S from Example 7, we have $\tau_1(\dot{S}, 4) = (\langle (3,0), (3,1), (3,2), (4,2) \rangle, v)$ with underlying stream $\tau_1(S, 4) = ([3, 4], \upsilon)$.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Correspondence for tuple windows is more subtle due to the different options to realize them.

Lemma 2
Let stream S underlie tick stream $\dot{S}$ and assume the tuple window $\#_n(S)$ is based on the order in which atoms appeared in S. Then, $\#_n(S)$ underlies $\#_n(\dot{S})$.

Example 9 (cont'd)
Stream S has two tuple windows of size 1: $S_a = ([3,4], \{3 \mapsto \{a\}\})$ and $S_b = ([3,4], \{3 \mapsto \{b\}\})$; the latter underlies $\#_1(\dot{S}) = (\langle(3,2),(4,2)\rangle, (3,2) \mapsto \{b\})$.

We can represent a stream $S = (T, \upsilon)$ alternatively by T and a set of *time-pinned* atoms, i.e., the set $\{a_@(\mathbf{x}, t) \mid a(\mathbf{x}) \in \upsilon(t), t \in T\}$. Similarly, tick streams can be modeled by *tick-pinned* atoms of form $a_\#(\mathbf{x}, t, c)$, where c increases by 1 for every incoming signal.

Example 10 (cont'd)
Given extra knowledge about the time $t = 4$, stream S is fully represented by $\{a_@(3), b_@(3)\}$, whereas tick stream $\dot{S}$ can be encoded by the set $\{a_\#(3,1), b_\#(3,2)\}$.

The notions of data/interpretation stream readily carry over to their tick analogs. Moreover, we say a tick interpretation stream I is an *answer stream* of program P (for tick data stream D at t), if the underlying stream I' of I is an answer stream of P (for the underlying data stream D' at t).

LARS to ASP (Algorithm 1). Plain LARS programs extend normal logic programs by allowing extended atoms in rule bodies, and also @-atoms in rule heads. Thus, if we restrict α and β_i in (1) to atoms, we obtain a normal rule. This observation is used for the translation of LARS to ASP as shown in Algorithm 1. The encoding has to take care of two central aspects. First, each extended atoms e is encoded by an (ordinary) atom a that holds iff e holds. Second, entailment in LARS is defined with respect to some data stream D and background data B at some time t. Stream signals and background data are encoded as facts, and temporal information by adding a time argument to atoms. The central ideas of the encoding are illustrated by the following example.

Example 11
Consider the LARS program P comprising the single rule $r = b(X) \leftarrow \boxplus^2 \Diamond a(X)$. Assume we are at time $t = 7$. We replace the window atom in the body by a fresh atom $\omega(X)$, which must hold if $a(X)$ holds at 7, 6 or 5. Thus, we can encode r in ASP by the following rules: $b(X) \leftarrow \omega(X); \omega(X) \leftarrow a_@(X,7); \omega(X) \leftarrow a_@(X,6); \omega(X) \leftarrow a_@(X,5)$. Assume an atom $a(y)$ was streaming in at time 5; modeled as time-pinned fact $a_@(y,5)$, we derive $\omega(y)$ and thus $b(y)$. That is, $b(y)$ holds at time 7, since signal $a(y)$ at 5 is still within the window.

Conceptually, the translation of a LARS program P to an ASP program $\hat{P}$ is such that if atom $a(\mathbf{x})$ (where $\mathbf{x} = x_1, \ldots, x_n$) is in an answer set A of $\hat{P}$, then $a(x)$ holds *now*. If the current time point is t, this is encoded in two ways, viz. by $a(\mathbf{x}) \in A$ and the time-pinned atom $a_@(\mathbf{x}, t) \in A$. This auxiliary atom corresponds to the LARS

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Algorithm 1: Plain LARS program to ASP $larsToAsp(P, t)$

Input: A (potentially non-ground) plain LARS program P, and the evaluation time point t

Output: ASP encoding $\hat{P}$, i.e., a set of normal logic rules

1 $Q := \{ a(\mathbf{X}) \leftarrow now(\dot{N}), a_{@}(\mathbf{X}, \dot{N});\ a_{@}(\mathbf{X}, \dot{N}) \leftarrow now(\dot{N}), a(\mathbf{X}) \mid a \text{ is a predicate in } P \}$
2 $R := \bigcup_{r \in P} larsToAspRules(r)$
3 **return** $Q \cup R \cup \{now(t)\}$

4 **defn** $larsToAspRules(r) = \{baseRule(r)\} \cup \bigcup_{i=1}^{m} windowRules(e_i)$

5 **defn** $baseRule(h \leftarrow e_1, \ldots, e_n, \text{not } e_{n+1}, \ldots, \text{not } e_m) =$
6 $\quad atm(h) \leftarrow atm(e_1), \ldots, atm(e_n), \text{not } atm(e_{n+1}), \ldots, \text{not } atm(e_m)$

7 **defn** $atm(e) =$ **match** e
8 $\quad$ **case** $a(\mathbf{X}) \Rightarrow a(\mathbf{X})$
9 $\quad$ **case** $@_T a(\mathbf{X}) \Rightarrow a_{@}(\mathbf{X}, T)$
10 $\quad$ **case** $\boxplus^w @_T a(\mathbf{X}) \Rightarrow \omega_e(\mathbf{X}, T)$ $\quad$ // ω_e is a fresh predicate associated with e
11 $\quad$ **case** $\boxplus^w \Diamond a(\mathbf{X}) \Rightarrow \omega_e(\mathbf{X})$
12 $\quad$ **case** $\boxplus^w \Box a(\mathbf{X}) \Rightarrow \omega_e(\mathbf{X})$

13 **defn** $windowRules(e) =$ **match** e
14 $\quad$ **case** $\boxplus^n @_T a(\mathbf{X}) \Rightarrow \{ \omega_e(\mathbf{X}, T) \leftarrow now(\dot{N}), a_{@}(\mathbf{X}, T), T = \dot{N} - i \mid i = 0, \ldots, n \}$
15 $\quad$ **case** $\boxplus^n \Diamond a(\mathbf{X}) \Rightarrow \{ \omega_e(\mathbf{X}) \leftarrow now(\dot{N}), a_{@}(\mathbf{X}, T), T = \dot{N} - i \mid i = 0, \ldots, n \}$
16 $\quad$ **case** $\boxplus^n \Box a(\mathbf{X}) \Rightarrow \{ \omega_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } spoil_e(\mathbf{X}) \} \cup$
17 $\quad\quad \{ spoil_e(\mathbf{X}) \leftarrow a(\mathbf{X}), now(\dot{N}), \text{not } a_{@}(\mathbf{X}, T), T = \dot{N} - i \mid i = 1, \ldots, n \}$
18 $\quad$ **case** $\boxplus^{\#n} @_T a(\mathbf{X}) \Rightarrow \{ \omega_e(\mathbf{X}, T) \leftarrow cnt(\dot{C}), a_{\#}(\mathbf{X}, T, D), D = \dot{C} - j \mid j = 0, \ldots, n-1 \}$
19 $\quad$ **case** $\boxplus^{\#n} \Diamond a(\mathbf{X}) \Rightarrow \{ \omega_e(\mathbf{X}) \leftarrow cnt(\dot{C}), a_{\#}(\mathbf{X}, T, D), D = \dot{C} - j \mid j = 0, \ldots, n-1 \}$
20 $\quad$ **case** $\boxplus^{\#n} \Box a(\mathbf{X}) \Rightarrow \{ \omega_e(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } spoil_e(\mathbf{X}) \} \cup$
21 $\quad\quad \{ spoil_e(\mathbf{X}) \leftarrow a(\mathbf{X}), cnt(\dot{C}), tick(T, D), \dot{C} - n + 1 \leqslant D \leqslant \dot{C}, \text{not } a_{@}(\mathbf{X}, T) \} \cup$
22 $\quad\quad \{ spoil_e(\mathbf{X}) \leftarrow a(\mathbf{X}), cnt(\dot{C}), tick(T, D), D = \dot{C} - n + 1, a_{\#}(\mathbf{X}, T, D'), D' < D \}$
23 $\quad$ **else** $\emptyset$

@-atom $@_t a(\mathbf{x})$, which then also holds now. In general for any $t' \in \mathbb{N}$, if $@_{t'} a(\mathbf{x})$ holds in an answer stream S now, then $a_{@}(\mathbf{x}, t')$ is in the corresponding answer set $\hat{S}$, but $a(\mathbf{x})$ is included only for $t' = t$. The resulting equivalence is stated by the rules Q in Algorithm 1, Line 1. To single out the current time point, we use an auxiliary predicate *now*.

The ASP encoding $\hat{P}$ for P at t is then obtained by Q, $\{now(t)\}$ and rule encodings R as computed by *larsToAspRules*. Given a LARS rule r of form (1), we replace every non-ordinary extended atom by a new auxiliary atom $atm(e)$ (Lines 8–12). Accordingly, for e of form $@_T a(\mathbf{X})$, we use $a_{@}(\mathbf{X}, T)$ (where T and $\mathbf{X}$ can be non-ground). For a window atom e, we use a new predicate ω_e for an *encoded window atom*. If e has the form $\boxplus^w \star a(\mathbf{X})$, $\star \in \{\Diamond, \Box\}$, we use a new atom $\omega_e(\mathbf{X})$, while for e of form $\boxplus^w @_T a(\mathbf{X})$, we use $\omega_e(\mathbf{X}, T)$ with a time argument.

Window encoding. Predicate ω_e has to hold in an answer set $\hat{S}$ of $\hat{P}$ iff e holds in a corresponding answer stream S of P at t. We use the function *windowRules*, which returns a set of rules to derive ω_e depending on the window (Lines 14–23). In case $e = \boxplus^n @_T a(\mathbf{X})$, we have to test whether $a_{@}(\mathbf{X}, T)$ holds for some time T

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

within the last n time points. For $\boxplus^n \Diamond a(\mathbf{X})$, we omit T in the rule head. Dually, if $\boxplus^n \Box a(\mathbf{X})$ holds for the same substitution $\mathbf{x}$ of $\mathbf{X}$ for all previous n time points, then in particular it holds now. So we derive $\omega_e(\mathbf{x})$ by the rule in Line 16 if $a(\mathbf{x})$ holds now and there is no *spoiler* i.e., a time point among $t-1, \mathit{dots}, t-n$ where $a(\mathbf{x})$ does not hold. This is established by the rule in Line 17. (We assume the window does not exceed the timeline and thus do not check $T - i \geqslant 0$.) Adding $a(\mathbf{X})$ to the body ensures safety of $\mathbf{X}$ in $a_@(\mathbf{X}, T)$.

For $\boxplus^{\#n} @_T a(\mathbf{X})$, we match every atom $a(\mathbf{x})$ with the time it occurs in the window of the last n tuples. Accordingly, we track the relation between arguments $\mathbf{x}$, the time t of occurrence in the stream, and the count c. To this end, we assume any input signal $a(\mathbf{x})$ is provided as $\{a_@(\mathbf{x}, t), a_\#(\mathbf{x}, t, c)\}$. Furthermore, the rules in Line 18 employ a predicate *cnt* that specifies the current tick count (as does *now* for the time tick). Based on this, the window is created analogously to a time-based window but counting back $n-1$ tuples instead of n time points. The case $\boxplus^{\#n} \Diamond a(\mathbf{X})$ is again analogous, but variable T is not included in the head.

For $\boxplus^{\#n} \Box a(\mathbf{X})$, Line 20 is as in the time-based analog (Line 16); $a(\mathbf{X})$ must hold now and there must not exist a spoiler. First, Line 21 ensures that $a(\mathbf{X})$ holds at every time point T in the window's range, determined by reaching back $n-1$ tick counts to count D. To do so, we add to the input stream an auxiliary atom of form $\mathit{tick}(t, c)$ for every tick (t, c) of the stream. Second, Line 22 accounts for the cut-off position within a time point, ensuring a is within the selected range of counts. Finally, $\mathit{windowRules}(e) = \emptyset$ if e is an atom or an @-atom, as they do not need extra rules for their derivation.

Example 12

Consider a stream $\dot{S}'$, which adds to $\dot{S}$ from Example 6 tick $(4, 3)$ with evaluation $v(4, 3) = \{a\}$. We evaluate $\boxplus^{\#2} \Box a$. The tick-pinned atoms are $a_\#(3, 1)$, $b_\#(3, 2)$ and $a_\#(4, 3)$; the window selects the last two, i.e., atoms with counts $D \geqslant 2$. It thus covers time points 3 and 4. While atom a occurs at time 3, it is not included in the window anymore, since its count is $1 < D$.

Stream encoding. Let $O = (K, v)$ be a tick stream at tick (t_m, c_m). We define its encoding $\hat{O}$ as $\{a_@(\mathbf{x}, t) \mid a(\mathbf{x}) \in v(t, c), (t, c) \in K\} \cup \{a_\#(\mathbf{x}, t, c) \mid a(\mathbf{x}) \in v(t, c), (t, c) \in K, a(\mathbf{x}) \in A^E\} \cup \{\mathit{cnt}(c_m)\} \cup \{\mathit{tick}(t, c) \mid (t, c) \in K\}$. We may assume that rules access background data B only by atoms (and not with @-atoms or window atoms). Viewing B as facts in the program, we skip further discussion. The following implicitly disregards auxiliary atoms in the encoding.

Proposition 1

Let P be a LARS program, $D = (K, v)$ be a tick data stream at tick (t, c) and let $\hat{P} = \mathit{LarsToAsp}(P, t)$. Then, S is an answer stream of P for D at t iff $\hat{S}$ is an answer set of $\hat{P} \cup \hat{D}$.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Example 13
We consider program P of Example 11, i.e., the rule $r = b(X) \leftarrow \boxplus^2 \diamond a(X)$. The translation $\hat{P} = \mathit{LarsToAsp}(P, 7)$ is given by the following rules, where $\omega = \omega_{\boxplus^2 \diamond a(X)}$:

$$
\begin{array}{lll}
r_0: & b(X) \leftarrow \omega(X) & \\
r_1: & \omega(X) \leftarrow \mathit{now}(\dot{N}),\ a_@(X,T),\ T = \dot{N} - 0 & \\
r_2: & \omega(X) \leftarrow \mathit{now}(\dot{N}),\ a_@(X,T),\ T = \dot{N} - 1 & \\
r_3: & \omega(X) \leftarrow \mathit{now}(\dot{N}),\ a_@(X,T),\ T = \dot{N} - 2 & \\
r_n: & \mathit{now}(7) \leftarrow & \\
q_1: & a(X) \leftarrow \mathit{now}(\dot{N}),\ a_@(X,\dot{N}) & \\
q_2: & a_@(X,\dot{N}) \leftarrow \mathit{now}(\dot{N}),\ a(X) & \\
q_3: & b(X) \leftarrow \mathit{now}(\dot{N}),\ b_@(X,\dot{N}) & \\
q_4: & b_@(X,\dot{N}) \leftarrow \mathit{now}(\dot{N}),\ b(X) &
\end{array}
$$

The single answer stream of P for D at 7 is $I = ([0,7], \{5 \mapsto \{a(y)\}, 7 \mapsto \{b(y)\}\})$, which corresponds to the set $\{a_@(y,5), b_@(y,7), b(y)\}$. In addition, the answer set $\hat{S}$ of $\hat{P} \cup \hat{D}$ contains auxiliary variables $\mathit{now}(7)$, $\mathit{cnt}(1)$, $a_\#(y,5,1)$ and $\omega(7)$ (and *tick* atoms).

4 Incremental ASP encoding

In this section, we present an incremental evaluation technique by adjusting an incremental variant of the given ASP encoding. We illustrate the central ideas in the following example.

Example 14 (*cont'd*)
Consider the following rules Π similar to $\hat{P}$ of Example 13, where predicate *now* is removed. Furthermore, we instantiate the *tick time variable* $\dot{N}$ with 7 to obtain so-called *pinned* rules. (Later, pinning also includes grounding the *tick count variable* $\dot{C}$ with the tick count.)

$$
\begin{array}{lll}
r'_0: & b(X) \leftarrow \omega(X) & \\
r'_1: & \omega(X) \leftarrow a_@(X,7) & \\
r'_2: & \omega(X) \leftarrow a_@(X,6) & \\
r'_3: & \omega(X) \leftarrow a_@(X,5) & \\
q'_1: & a(X) \leftarrow a_@(X,7) & \\
q'_2: & a_@(X,7) \leftarrow a(X) & \\
q'_3: & b(X) \leftarrow b_@(X,7) & \\
q'_4: & b_@(X,7) \leftarrow b(X) &
\end{array}
$$

Based on the stream, encoded by $\hat{D} = \{a_@(y,5), a_\#(y,5,1)\}$ (we omit tick atoms), we obtain a ground program $\hat{P}_{D,(7,1)}$ from Π by replacing X with y; the answer set is $\hat{D} \cup \{\omega(y), b(y), b_@(y,7)\}$.

Assume now that time moves on to $t' = 8$, i.e., a stream D' at tick $(8,1)$. We observe that rules $q'_1, \ldots, q'_4$ must be replaced by $q''_1, \ldots, q''_4$, which replace time pin 7 by 8. Rule r'_0 can be maintained since it does not contain values from ticks. The time window covers time points $6, 7, 8$. This is reflected by removing r'_3 and instead adding $\omega(X) \leftarrow a_@(X,8)$.

That is, based on the time increment from $(7,1)$ to $(8,1)$, rules $E^- = \{q'_1, \ldots, q'_4, r'_3\}$ and their groundings G^- (with $X \mapsto y$) *expire*, and new rules $E^+ = \{q''_1, \ldots, q''_4, \omega(X) \leftarrow a_@(X,8)\}$ have to be grounded based on the remaining rules (and the data stream), yielding new ground rules G^+. We thus incrementally obtain a ground program $\hat{P}_{D',(8,1)} = (\hat{P}_{D,(7,1)} \setminus G^-) \cup G^+$, which encodes the program P for evaluation at tick $(8,1)$.

Before we formalize the illustrated incremental evaluation, we present its ingredients.

Algorithm 2: Incremental rule generation. Algorithm 2 shows the procedure *IncrementalRules* that obtains incremental rules based on a tick time t, a tick

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Algorithm 2: Incremental rules $\mathit{IncrementalRules}(t, c, \mathit{Sig})$

Input: Tick time t, tick count c, signal set Sig with at most one input signal, which is empty iff (t, c) is a time increment. (The LARS program P is global.)
Output: Pinned incremental rules annotated with duration until expiration

$F := \{\langle(\infty, \infty), \mathit{tick}(t, c) \leftarrow\rangle\}$
foreach $a(\mathbf{x}) \in \mathit{Sig}$: $F := F \cup \{\langle(\infty, \infty), a_{@}(\mathbf{x}, t) \leftarrow\rangle,\ \langle(\infty, \infty), a_{\#}(\mathbf{x}, t, c) \leftarrow\rangle\}$
$Q := \{\ \langle(1, \infty), a(\mathbf{X}) \leftarrow a_{@}(\mathbf{X}, t)\rangle,\ \langle(1, \infty), a_{@}(\mathbf{X}, t) \leftarrow a(\mathbf{X})\rangle \mid a \text{ is a predicate in } P\}$
$R := \emptyset$
foreach $r \in P$
  $\hat{r} := \mathit{baseRule}(r)$ // as defined in Algorithm 1
  $I := \bigcup_{e \in B(r)} \mathit{incrementalWindowRules}(e, t, c)$
  $R := R \cup I \cup \{\langle(\infty, \infty), \hat{r}\rangle\}$
return $F \cup Q \cup R$

defn $\mathit{incrementalWindowRules}(e, t, c) =$ **match** e
  case $\boxplus^{n} @_{T}\, a(\mathbf{X}) \Rightarrow \{\ \langle(n+1, \infty), \omega_{e}(\mathbf{X}, t) \leftarrow a_{@}(\mathbf{X}, t)\rangle\ \}$
  case $\boxplus^{n} \Diamond a(\mathbf{X}) \Rightarrow \{\ \langle(n+1, \infty), \omega_{e}(\mathbf{X}) \leftarrow a_{@}(\mathbf{X}, t)\rangle\ \}$
  case $\boxplus^{n} \Box a(\mathbf{X}) \Rightarrow \{\ \langle(\infty, \infty), \omega_{e}(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } \mathit{spoil}_{e}(\mathbf{X})\rangle\ \} \cup$
    $\{\ \langle(n, \infty), \mathit{spoil}_{e}(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } a_{@}(\mathbf{X}, t-1)\rangle\ \}$ // only if $n \geqslant 1$
  case $\boxplus^{\#n} @_{T}\, a(\mathbf{X}) \Rightarrow \{\ \langle(\infty, n), \omega_{e}(\mathbf{X}, t) \leftarrow a_{\#}(\mathbf{X}, t, c)\rangle\ \}$
  case $\boxplus^{\#n} \Diamond a(\mathbf{X}) \Rightarrow \{\ \langle(\infty, n), \omega_{e}(\mathbf{X}) \leftarrow a_{\#}(\mathbf{X}, t, c)\rangle\ \}$
  case $\boxplus^{\#n} \Box a(\mathbf{X}) \Rightarrow \{\ \langle(\infty, \infty), \omega_{e}(\mathbf{X}) \leftarrow a(\mathbf{X}), \text{not } \mathit{spoil}_{e}(\mathbf{X})\rangle\ \} \cup$
    $\{\ \langle(\infty, n), \mathit{spoil}_{e}(\mathbf{X}) \leftarrow a(\mathbf{X}), \mathit{tick}(t, c), \mathit{covers}^{\tau}_{e}(t), \text{not } a_{@}(\mathbf{X}, t)\rangle\ \} \cup$
    $\{\ \langle(\infty, n), \mathit{spoil}_{e}(\mathbf{X}) \leftarrow a_{\#}(\mathbf{X}, t, c), \mathit{covers}^{\tau}_{e}(t), \text{not } \mathit{covers}^{\#}_{e}(c)\rangle\ \} \cup$
    $\{\ \langle(\infty, n), \mathit{covers}^{\tau}_{e}(t) \leftarrow \mathit{tick}(t, c)\rangle\ , \langle(\infty, n), \mathit{covers}^{\#}_{e}(c) \leftarrow \mathit{tick}(t, c)\rangle\ \}$
  else $\emptyset$

count c and the *signal set* $\mathit{Sig} = v(t, c)$, where $\mathit{Sig} = \emptyset$, if (t, c) is a time increment of k. The resulting rules of Algorithm 2 are annotated with a tick that indicates how long the ground instances of these rules are applicable before they expire.

Definition 9 (*Annotated rule*)
Let (t, c) be a tick, where $t, c \in \mathbb{N} \cup \{\infty\}$, and r be a rule. Then, the pair $\langle(t, c), r\rangle$ is called an *annotated rule*, and (t, c) the *annotation* of r.

Annotations serve two purposes. First, in Algorithm 2, they express a *duration* how long a generated rule is applicable. Then, in Algorithm 3 below, this duration will be added to the current tick to obtain the *expiration tick* (annotation) of a rule. If a rule *expires* at tick (t, c), i.e., if its expiration tick (t', c') fulfills $t' \geqslant t$ or $c' \geqslant c$, then it has to be deleted from the encoding.

Example 15 (*cont'd*)
Each rule q'_i, $1 \leqslant i \leqslant 4$, has duration $(1, \infty)$. That is, after 1 time point, the rule will expire, regardless of how many atoms appear at the current time point. Hence, the *time duration* is 1, and the *count duration* is infinite, since these rules cannot expire based on arrival of atoms. Similarly, rules r'_i, $1 \leqslant i \leqslant 3$, have duration $(2, \infty)$ due to the time window length 2.

We will discuss expiration ticks based on these durations below. Algorithm 2 is concerned with generating the incremental rules and their durations. In the first two

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Algorithm 3: Single tick increment $\mathit{IncrementTick}(\Pi, G, t, c, \mathit{Sig})$

Input: Set of annotated, cumulative incremental rules $\Pi \supseteq \hat{D}$ collected until previous tick; its annotated groundings $G = \bigcup_{\langle (t',c'),r\rangle \in \Pi} \mathit{ground}(\Pi, r)$, tick time t, tick count c and signal set Sig
Result: Updated Π and G

1 $I := \mathit{IncrementalRules}(t, c, \mathit{Sig})$
2 $E^+ := \{\langle (t + t_\Delta, c + c_\Delta), r\rangle \mid \langle (t_\Delta, c_\Delta), r\rangle \in I\}$ // determine expiration for new rules
3 $E^- := \{\langle (t', c'), r\rangle \in \Pi \mid t' \leqslant t \text{ or } c' \leqslant c\}$ // expired incremental rules
4 $\Pi' := (\Pi \setminus E^-) \cup E^+$
5 $G^+ := \{\langle (t', c'), r'\rangle \mid \langle (t', c'), r\rangle \in E^+, r' \in \mathit{ground}(\Pi', r)\}$ // new ground rules with expiration
6 $G^- := \{\langle (t', c'), r\rangle \in G \mid t' \leqslant t \text{ or } c' \leqslant c\}$ // expired ground rules with expiration annotation
7 $G' := (G \setminus G^-) \cup G^+$
8 **return** $\langle \Pi', G'\rangle$

lines, auxiliary facts, as discussed earlier, are added to a fresh set F. These facts expire neither based on time nor count, hence the duration annotation (∞, ∞). As illustrated in Example 15, we collect in set Q the incremental analogue of Q in Algorithm 1. These rules expire after 1 time point, hence the annotation $(1, \infty)$.

Within the loop, we collect for every LARS rule r a base rule $\hat{r}$ (as in Algorithm 1), together with incremental window rules, computed by *incrementalWindowRules* (Lines 10–21). We assign an infinite duration (∞, ∞) to the base rule $\hat{r}$ since it never needs to expire, i.e., it suffices to ensure that encoded window atoms ω_e expire correctly. An optimized version may expire also $\hat{r}$ due to the durations of atoms ω_e from the incremental windows that derive them.

Incremental window encoding. We already gave the intuition for atoms $\boxplus^n \Diamond a(\mathbf{X})$. The case of $\boxplus^n @_T a(\mathbf{X})$ is similar. Like in the static translation, we additionally have to use the time information in the head. Similarly, $\boxplus^{\#n} \Diamond a(\mathbf{X})$ and $\boxplus^{\#n} @_T a(\mathbf{X})$ expire after n new incoming atoms, instead of n time points. For $\boxplus^n \Box a(\mathbf{X})$, we add a spoiler rule for the previous time point $t-1$, which will be considered for the next n time points.

For $e = \boxplus^{\#n} \Box a(\mathbf{X})$, we maintain two spoiler rules as in the static case that ensure $a(\mathbf{X})$ occurs at all time points in the coverage of the window, and the occurrence of $a(\mathbf{X})$ at the leftmost time point is also covered by the tick count. At tick (t, c), we have a guarantee for the next n atoms that tick time t will be covered within the window. This is expressed by a rule $\mathit{covers}^\tau_e(t) \leftarrow \mathit{tick}(t, c)$ with duration (∞, n). Likewise, $\mathit{covers}^\#_e(c) \leftarrow \mathit{tick}(t, c)$ will select tick count c within duration (∞, n). Notably, coverage for time increments $(t+k, c)$ may extend the tuple window arbitrarily long if no atoms appear. As the spoiler rules are based on these cover atoms, their expiration is optional, i.e., keeping them does not yield incorrect inferences. However, we can also expire them when they become redundant, i.e., after n atoms. Finally, *IncrementalRules* returns the $F \cup Q \cup R$, where R contains all base rules and incremental window rules.

Algorithm 3: Incremental evaluation. Algorithm 3 gives the high-level procedure *IncrementTick* to incrementally adjust a program encoding. We assume the function $\mathit{ground}(\Pi, r)$ returns all possible ground instances of a rule $r \in \Pi$ (due to constants

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

in Π). In fact, *IncrementTick* maintains a program Π that contains the encoded data stream $\hat{D}$ and non-expired incremental rules as obtained by consecutive calls to *IncrementalRules*, tick by tick. Moreover, it maintains a grounding G of Π, i.e., the incremental encoding for the previous tick plus expiration annotations.

The procedure starts by generating the new incremental rules I based on Algorithm 2 described above. Next, we add for each rule the current tick (t, c) to its duration (t_Δ, c_Δ) (componentwise). This way, we obtain new incremental rules E^+ with expiration tick annotations. Dually, we collect in E^- previous incremental rules that expire now, i.e., when the current tick reaches the expiration tick time t' or count c'. The new cumulative program Π results by removing E^- from Π and adding E^+. Based on Π', we obtain in Line 5 the new (annotated) ground rules G^+ based on E^+. As in Line 3, we determine in Line 6 the set G^- of expired (annotated) ground rules. After assigning G' the updated annotated grounding in Line 7, we return the new incremental evaluation state $\langle \Pi', G' \rangle$, from which the current incremental program is derived as follows.

Definition 10 (*Incremental Program*)
Let P be a LARS program and $D = (K, v)$ be a tick stream, where $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$. The *incremental program* $\hat{P}_{D,k}$ *of* P *for* D *at tick* (t_k, c_k), $1 \leqslant k \leqslant m$, is defined by $\hat{P}_{D,k} = \{r \mid \langle (t', c'), r \rangle \in G_k\}$, where

$$\langle \Pi_k, G_k \rangle = \begin{cases} \mathit{IncrementTick}(\emptyset, \emptyset, t_1, c_1, \emptyset) & \text{if } k = 1, \\ \mathit{IncrementTick}(\Pi_{k-1}, G_{k-1}, t_k, c_k, v(t_k, c_k)) & \text{else.} \end{cases}$$

In the following, body occurrences of form $@_t a(\mathbf{X})$ are viewed as shortcuts for $\boxplus^\infty @_t a(\mathbf{X})$. The next proposition states that to faithfully compute an incremental program from scratch, it suffices to start iterating *IncrementalTick* from the oldest tick that is covered from any window in the considered program. In the subsequent results, we disregard auxiliary atoms like $\mathit{tick}(t, c), \mathit{covers}^\tau_e(t)$, etc. Let $AS^I(\hat{P})$ denote the answer sets of $\hat{P}$, projected to intensional atoms.

Proposition 2
Let $D = (K, v)$ and $D' = (K', v')$ be two data streams such that (i) $D' \subseteq D$, (ii) $K = \langle (t_1, c_1), \ldots, (t_m, c_m) \rangle$ and (iii) $K' = \langle (t_k, c_k), \ldots, (t_m, c_m) \rangle$, $1 \leqslant k \leqslant m$. Moreover, let P be a LARS program and n^τ (resp. $n^\#$) be the maximal window length for all time (resp. tuple) windows, or ∞ if none exists. If $t_k \leqslant t_m - n^\tau$ and $c_k \leqslant c_m - n^\# + 1$, then $AS^I(\hat{P}_{D,m}) = AS^I(\hat{P}_{D',m})$.

The result stems from the fact that in the incremental program $\hat{P}_{D,m}$ no rule can fire based on outdated information, i.e., atoms that are not covered by any window anymore. In order to obtain an equivalence between $\hat{P}_{D,m}$ and $\hat{P}_{D',m}$ on extensional atoms, we would have to drop all atoms of the stream encoding $\hat{D}$ during *IncrementalTick*, as soon as no window can access them anymore.

The following states the correspondence between the static and the incremental encoding.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

Proposition 3
Let P be a LARS program and D be a tick data stream at tick $m = (t, c)$. Furthermore, let $\hat{P} = LarsToAsp(P, t)$ and $\hat{P}_{D,m}$ be the incremental program at tick m. Then, $S \cup \{now(t), cnt(c)\}$ is an answer set of $\hat{P} \cup \hat{D}$ iff S is an answer set of $\hat{P}_{D,m}$ (modulo aux. atoms).

In conclusion, we obtain from Propositions 1 and 3 the desired correctness of the incremental encoding.

Theorem 1
Let P be a LARS program and $D = (K, v)$ be a tick data stream at tick $m = (t, c)$. Then, S is an answer stream of P for D at t iff $\hat{S}$ is an answer set of $\hat{P}_{D,m}$ (modulo aux. atoms).

5 Implementation

We now present *Ticker*, our stream reasoning engine which is written in *Scala* (source code available at `https://github.com/hbeck/ticker`). It has two high-level processing methods for a given time point: *append* is adding input signals, and *evaluate* returns the model. Two implementations of this interface are provided, based on two evaluation strategies discussed next.

One-shot solving by using Clingo. The ASP solver Clingo (Gebser *et al.* 2014) is a practical choice for stratified programs, where no ambiguity arises which model to compute. At every time point, respectively, at the arrival of a new atom, the static LARS encoding $\hat{P}$ (of Algorithm 1) is streamed to the solver and results are parsed as soon as Clingo reports a model. In case of multiple models, we take the first one. Apart from this so-called push-based mode, where a model is prepared after every *append* call, we also provide a pull-based mode, where only *evaluate* triggers model computation. Clingo's reactive features are not applicable (see supplementary material).

Incremental evaluation by TMS. In this strategy, the model is maintained continuously using our own implementation of the truth-maintenance system (TMS) by Doyle (1979). A TMS *network* can be seen as logic program P and data structures that reflect a so-called *admissible* model M for P. Given a rule r, the network is updated such that it represents an admissible model M' for $P \cup \{r\}$, thereby reconsidering the truth value of atoms in M only if they may change due to the network. Ticker analogously allows for rule removals, i.e., obtaining an admissible model M' for $P \setminus \{r\}$. We exploit the following correspondence of admissible models and answer sets.

Theorem 2 (*cf. Elkan (1990)*)
(i) A model M is admissible for program P iff it is an answer set of P. (ii) Deciding whether P has an admissible model is NP-complete.

Notably, this correspondence holds only in the absence of constraints, or more generally, odd loops (Elkan 1990). In case such programs are used, neither a correct

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

output nor termination are guaranteed. Elkan points out that also incremental reasoning is NP-complete, i.e., given an admissible model M for P, deciding for a rule r whether $P \cup \{r\}$ has an admissible model. No further knowledge about TMS is required for our purpose. A detailed, formal review can be found in Beck (2017), supplementing the textual presentation in Doyle (1979).

When new data is streaming in, we compute the incremental rules G^+ as defined in Algorithm 2, add them to the TMS network and remove expired ones G^-, which results in an immediate model update. The incremental TMS strategy is, due to its maintenance outset, more amenable to keep the latest model by inertia, which may be desirable in some applications.

Pre-grounding. In Algorithm 3, we assume a grounder that instantiates pinned rules from Algorithm 2. To provide according efficient techniques is a topic on its own; we restrict grounding to the pinning process in Algorithm 2. To this end, we add to each rule for every variable X in the scope of a window atom an additional *guard* atom that includes X. The guard is either background data or intensional. Based on this, the incremental rules in Algorithm 2 can be grounded upfront, apart from the tick variables $\dot{N}$ and $\dot{C}$ and time variables in @-atoms. We call such programs *pre-grounded*. A LARS program P is first translated into an encoding $\hat{P}$ with several data structures that differentiate Q, base rules R, and window rules W. During the initialization process, pre-groundings are prepared, where arithmetic expressions are represented by auxiliary atoms. During grounding, they are removed if they hold, otherwise the entire ground rule is removed.

Example 16
For rule $r = @_T high \leftarrow value(V), \boxplus^n @_T alpha(V), V \geqslant 18$ of Example 5, where $value(V)$ was added as guard, we get a base rule $\hat{r} = high_{@}(T) \leftarrow value(V), \omega_e(V, T), Geq(V, 18)$, where $e = \boxplus^n @_T alpha(V)$. Given facts $\{value(0), \ldots, value(30)\}$ (from background data or potential derivations), we obtain the pre-grounding $\{high_{@}(T) \leftarrow value(x), \omega_e(x, T) \mid x \in \{18, \ldots, 30\}\}$.

We then use pre-groundings in Algorithm 2 such that when Algorithm 3 receives its result I, all rules are already ground. Thus, the implementation has no further grounding in Algorithm 3 and only concerns handling durations and expirations, which is realized based on efficient lookups.

6 Evaluation

For an experimental evaluation, we consider two scenarios in the context of content-centric network management, where smart routers need to manage packages dynamically (Beck *et al.* 2017).

Scenario A: Caching Strategy. Figure 2 shows a program to dynamically select one of several strategies (*fifo*, *lfu*, *lru*, *random*) how to replace content items (video chunks) in a local cache. A user request parameter α, signaled as atom $alpha(V)$, is monitored and abstracted to a qualitative level (r_1–r_3) using tuple-based windows. At this level, time-based windows are used to decide among *fifo*, *lfu* and *lru* (r_4–r_6); the default policy is *random* (r_7–r_{10}).

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

$$
\begin{array}{lrcl}
r_1: & @_T\, high & \leftarrow & value(V), \boxplus^{\#n} @_T\, alpha(V),\ 18 \leq V \\
r_2: & @_T\, mid & \leftarrow & value(V), \boxplus^{\#n} @_T\, alpha(V),\ 12 \leq V < 18 \\
r_3: & @_T\, low & \leftarrow & value(V), \boxplus^{\#n} @_T\, alpha(V),\ V < 12 \\
r_4: & lfu & \leftarrow & \boxplus^{n} \Box\, high \\
r_5: & lru & \leftarrow & \boxplus^{n} \Box\, mid \\
r_6: & fifo & \leftarrow & \boxplus^{n} \Box\, low \\
r_7: & done & \leftarrow & lfu \\
r_8: & done & \leftarrow & lru \\
r_9: & done & \leftarrow & fifo \\
r_{10}: & random & \leftarrow & \text{not } done
\end{array}
$$

Fig. 2. Program for scenario A, setup *A2*. Setup *A1* uses $\boxplus^{n}$ in $r_1 - r_3$ instead of $\boxplus^{\#n}$.

$$
\begin{array}{lrcl}
r_1: & need(I,N) & \leftarrow & item(I), node(N), \boxplus^{n} \Diamond req(I,N) \\
r_2: & avail(I,N) & \leftarrow & item(I), node(N), \boxplus^{n} \Diamond cache(I,N) \\
r_3: & get(I,N,M) & \leftarrow & source(I,N,M), \text{not } nGet(I,N,M) \\
r_4: & nGet(I,N,M) & \leftarrow & node(M), get(I,N,M'), M \neq M' \\
r_5: & nGet(I,N,M) & \leftarrow & source(I,N,M), source(I,N,M'), M \neq M', qual(M,L), qual(M',L'), L < L' \\
r_6: & source(I,N,M) & \leftarrow & need(I,N), \text{not } avail(I,N), avail(I,M), reach(N,M) \\
r_7: & reach(N,M) & \leftarrow & conn(N,M) \\
r_8: & reach(N,M) & \leftarrow & reach(N,M'), conn(M',M), M' \neq M, N \neq M \\
r_9: & conn(N,M) & \leftarrow & edge(N,M), \text{not } \boxplus^{n} \Box\, down(M) \\
r_{10}: & qual(N,L) & \leftarrow & node(N), lev(L), lev(L'), L' < L, \boxplus^{n} \Diamond qLev(N,L), \text{not } \boxplus^{n} \Diamond qLev(N,L')
\end{array}
$$

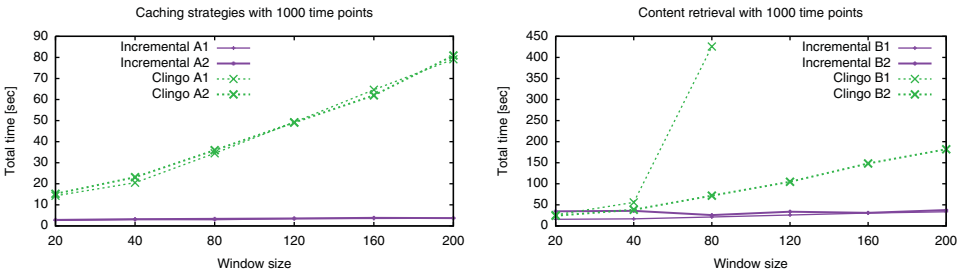
Fig. 3. Program for scenario B.

Setup *A1* replaces tuple windows in rules r_1–r_3 by time windows [as in Beck *et al.* (2017)]; setup *A2* uses the program as shown. The input signals $alpha(V)$ are generated such that a random mode *high*, *medium* or *low* is repeatedly chosen and kept for twice the window size.

Scenario B: Content Retrieval. Figure 3 depicts the second program, which, in contrast to the former, may have multiple models and includes recursive computation, instead of straightforward chaining. In a network, items can be cached and requested at every node. If a user recently requested item I at node N (rule r_1), it is either available at N (r_2) or has to be retrieved from some other node M (r_3, r_6). A single node is selected (r_3) that provides the best quality level (e.g., connection speed) among all reachable nodes having I (r_5). Connecting paths (r_7, r_8) work unless the end node of an edge was down during the last n time points (r_9). Finally, nodes repeatedly report their quality level, among which the best recent value is selected (r_{10}). We take the classic Abilene network (Spring *et al.* 2004), i.e., the set of edges $\{(x,y),(y,x) \mid (x,y) \in E\}$, where $E = \{(0,1),(1,2),\ldots,(9,10),(0,10),(1,10),(2,8),(3,7)\}$. We use three quality levels $\{0,1,2\}$ and two items. In setup *B1*, at every time point, with respective probability $p = 0.1$, each item is requested at a random node, one random item is cached at a random node and one random node is signaled as down. Further, the quality level of each node changes with $p = 3/n$, where n is the window size. Setup *B2* requests each item with $p = 0.5$ at 1–3 random nodes, always signals 1–3 random cache entries and a quality level for every node with $p = 0.25$, which is then with $p = 0.9$ the previous one. With $p = 1/n$, a random node will be down for $1.5 \cdot n$ time points.

Evaluations. For each scenario and setup, we ran two evaluation modes. The first one fixes the number tp of time points and increases the window size n stepwise; the second setup does vice versa.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press



(a) (b)

Fig. 4. Runtime evaluation for increasing window size. (a) Scenario A (caching strategy). (b) Scenario B (content retrieval).

In each evaluation mode, we measure

(i) the time t_{init} needed to initialize the engine before input signals are streamed (in case of the incremental mode, this includes pre-grounding),
(ii) the average time t_{tick} per tick, i.e., a time or count increment and
(iii) the total time t_{total} of a single run, resulting from t_{init} and t_{tick} for all timepoints and atoms.

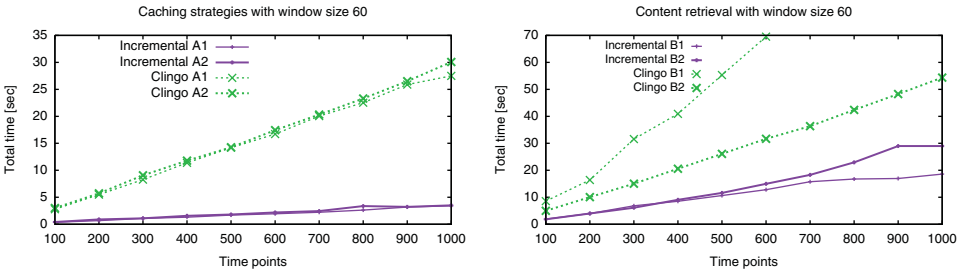
(Note that a tick increment may involve both adding and removing rules.) Each evaluation includes runtimes for both reasoning strategies, i.e., based on Clingo (Vers. 5.1.0) and based on the incremental approach with Doyle's TMS. For a fair comparison with TMS, we use Clingo in a push-based mode, i.e., a model is computed whenever a signal streams in. To obtain robust results, we first run each instance twice without recording time, and then build the average over the next five runs for t_{init}, t_{total} and t_{tick}, respectively. The first two runs serve as warm-up for the environment, ensuring that potential optimizations by the Java-Virtual-Machine do not distort the measurements. All evaluations were executed on a laptop with an Intel i7 CPU at 2.7 GHz and 16 GB RAM running the Java-Virtual-Machine version 1.8.0_112. They can be run via class `LarsEvaluation`.

Results. We report here on findings regarding the total execution times t_{total}, shown in Figures 4 and 5.

Detailed runtimes for t_{total}, t_{init} and t_{tick} can be found in Tables C1–C8 in the supplementary material.

Figure 4 shows the effect on the runtime when the window size is increased. We observe that for both scenarios the total execution time t_{total} is proportionally growing using Clingo, while for the incremental implementation (TMS), t_{total} remains nearly constant. For Clingo, this is explained by the full recomputation of the model with all previous input data, while TMS benefits from prior model computations and is thus significantly faster for larger window sizes. Dually, Figure 5 show the runtime evaluation for increasing number of timepoints. For both scenarios, the total run time t_{total} of both Clingo and TMS increases linearly, and incremental is significantly faster than repeated one-shot solving. For both evaluations (Figs. 4 and 5), using different windows (*A1* versus *A2*) has no influence on the execution

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press



(a) (b)

Fig. 5. Runtime evaluation for increasing timepoints. (a) Scenario A (caching strategy). (b) Scenario B (content retrieval).

time, for both Clingo and TMS, and different input patterns (*B1* versus *B2*) seem to influence TMS less than Clingo.

In conclusion, the experiments indicate that incremental model update may computationally pay off in comparison to repeated recomputing from scratch, in particular when using large windows. Furthermore, maintenance aims at keeping a model by inertia, which, however, we have not assessed in the experiments.

7 Related work and conclusion

In Beck *et al.* (2015), TMS techniques have been extended and applied for (plain) LARS, instead of reducing LARS to ASP. In contrast, the present approach does not primarily focus on model update, but incremental program update. Apart from work on Clingo mentioned earlier, alternatives to one-shot ASP were also considered by Alviano *et al.* (2014). The ASP approach of Do *et al.* (2011) for stream reasoning calls the dlvhex solver; it has no incremental reasoning and cannot handle heavy data load. ETALIS (Anicic *et al.* 2012) is a prominent rule formalism for complex event processing to reason about intervals for atomic events with a peculiar minimal model semantics. ETALIS is monotonic for a growing timeline (as such trivially incremental), and does not feature window mechanisms. StreamLog (Zaniolo 2012) extends Datalog for single-model stream reasoning, where rules concluding about the past are excluded; neither windows nor incremental evaluation were considered. The DRed algorithm (Gupta *et al.* 1993) for incremental Datalog update deletes all consequences of deleted facts and then adds all rederivable ones from the rest. It was adapted to RDF streams by Barbieri *et al.* (2010), where tuples are tagged with an expiration time. Ren and Pan (2011) explored TMS techniques for ontology streams. However, windows and time reference were not considered in their monotonic setting. Toward incremental grounding, techniques as in Lefèvre and Nicolas (2009), Palù *et al.* (2009) and Dao-Tran *et al.* (2012) might be considered.

Outlook. The algorithms we have presented center around the idea of incrementally adapting a model based on an incremental adjustment of a program. Our implementation indicates performance benefits arising from incremental evaluation. Developing techniques for full grounding on-the-fly in this context remains to be

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press

done. On the semantic side, notions of closeness between consecutive models and guarantees to obtain them are intriguing issues for future work.

Acknowledgements

We thank Roland Kaminski for providing guidance on the use of Clingo.

Supplementary materials

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068417000370

References

Alviano, M., Dodaro, C. and Ricca, F. 2014. Anytime computation of cautious consequences in answer set programming. *Theory and Practice of Logic Programming 14,* 4–5, 755–770.

Anicic, D., Rudolph, S., Fodor, P. and Stojanovic, N. 2012. Stream reasoning and complex event processing in ETALIS. *Semantic Web 3*, 4, 397–407.

Babu, S. and Widom, J. 2001. Continuous queries over data streams. *SIGMOD Record 3,* 30, 109–120.

Barbieri, D. F., Braga, D., Ceri, S., Valle, E. D. and Grossniklaus, M. 2010. Incremental reasoning on streams and rich background knowledge. In *Proc. of Semantic Web: Research and Applications, 7th Extended Semantic Web Conference, ESWC 2010, Part I,* Heraklion, Crete, Greece, May 30–June 3, 2010, L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds. Lecture Notes in Computer Science, vol. 6088. Springer, 1–15.

Beck, H. 2017. *Reviewing Justification-based Truth Maintenance Systems from a Logic Programming Perspective*. Tech. Rep. INFSYS RR-1843-17-02, Institute of Information Systems, TU Vienna.

Beck, H., Bierbaumer, B., Dao-Tran, M., Eiter, T., Hellwagner, H. and Schekotihin, K. 2017. Stream reasoning-Based control of caching strategies in CCN routers. In *Proc. of the IEEE International Conference on Communications*, May 21–25, 2017, Paris, France, 1–6.

Beck, H., Dao-Tran, M. and Eiter, T. 2015. Answer update for rule-based stream reasoning. In *Proc. of the 24th International Joint Conference on Artificial Intelligence (IJCAI-15),* July 25–31, 2015, Buenos Aires, Argentina, Q. Yang and M. Wooldridge, Eds. AAAI Press/IJCAI, 2741–2747.

Beck, H., Dao-Tran, M., Eiter, T. and Fink, M. 2015. LARS: A logic-based framework for analyzing reasoning over streams. In *Proc. of 29th Conference on Artificial Intelligence (AAAI'15),* January 25–30, 2015, Austin, Texas, USA, B. Bonet and S. Koenig, Eds. AAAI Press, 1431–1438.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G. and Weinzierl, A. 2012. Omiga: An open minded grounding on-the-fly answer set solver. In *Proc. of Logics in Artificial Intelligence - 13th European Conference, JELIA 2012,* Toulouse, France, September 26–28, 2012, L. F. del Cerro, A. Herzig, and J. Mengin, Eds. Lecture Notes in Computer Science, vol. 7519. Springer, 480–483.

Della Valle, E., Ceri, S., van Harmelen, F. and Fensel, D. 2009. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intelligent Systems 24*, 83–89.

Do, T. M., Loke, S. W. and Liu, F. 2011. Answer set programming for stream reasoning. In *Proc. of Advances in Artificial Intelligence - 24th Canadian Conference on Artificial Intelligence, Canadian AI 2011,* St. John's, Canada, May 25–27, 2011, C. J. Butz and P. Lingras, Eds. Lecture Notes in Computer Science, vol. 6657. Springer, 104–109.

Doyle, J. 1979. A truth maintenance system. *Artificial Intelligence 12,* 3, 231–272.

Elkan, C. 1990. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence 43* 2, 219–234.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2014. *Clingo* = ASP + control: Preliminary report. In *Proc. of Technical Communications of the 30th International Conference on Logic Programming (ICLP'14)*, M. Leuschel and T. Schrijvers, Eds. Theory and Practice of Logic Programming, Online Supplement.

Gebser, M., Kaminski, R., Obermeier, P. and Schaub, T. 2015. Ricochet robots reloaded: A case-study in multi-shot ASP solving. In *Proc. of Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation - Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, T. Eiter, H. Strass, M. Truszczynski and S. Woltran, Eds. Lecture Notes in Computer Science, vol. 9060. Springer, 17–32.

Gupta, A., Mumick, I. S. and Subrahmanian, V. S. 1993. Maintaining views incrementally. In *Proc. of ACM SIGMOD International Conference on Management of Data*, 157–166.

Lefèvre, C. and Nicolas, P. 2009. The first version of a new ASP solver: Asperix. In *Proc. of Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009,* Potsdam, Germany, September 14–18, 2009, E. Erdem, F. Lin, and T. Schaub, Eds. Lecture Notes in Computer Science, vol. 5753. Springer, 522–527.

Palù, A. D., Dovier, A., Pontelli, E. and Rossi, G. 2009. Answer set programming with constraints using lazy grounding. In *Proc. of Logic Programming, 25th International Conference, ICLP 2009,* Pasadena, CA, USA, July 14–17, 2009, P. M. Hill and D. S. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer, 115–129.

Phuoc, D. L., Dao-Tran, M., Parreira, J. X. and Hauswirth, M. 2011. A native and adaptive approach for unified processing of linked streams and linked data. In *Proc. of ISWC (1)*, 370–388.

Ren, Y. and Pan, J. Z. 2011. Optimising ontology stream reasoning with truth maintenance system. In *Proc. of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011,* Glasgow, United Kingdom, October 24–28, 2011, C. Macdonald, I. Ounis, and I. Ruthven, Eds. ACM, 831–836.

Spring, N. T., Mahajan, R., Wetherall, D. and Anderson, T. E. 2004. Measuring ISP topologies with rocketfuel. *IEEE/ACM Transaction Network 12* 1, 2–16.

Zaniolo, C. 2012. Logical foundations of continuous query languages for data streams. In *Proc. of Datalog*, 177–189.

https://doi.org/10.1017/S1471068417000370 Published online by Cambridge University Press