Rule-Based Stream Reasoning

Alessandro Ronca

Oriel College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2019

To my wife Nadezda.

Acknowledgements

I wish to express my gratitude to my supervisors Bernardo Cuenca Grau, Mark Kaminski, and Ian Horrocks for having me welcomed to the KR group and guided through my DPhil. I would also like to thank Boris Motik for his supervision. I am grateful to all my colleagues and friends of the Department of Computer Science for their support.

My DPhil was supported by the SIRIUS Centre for Scalable Data Access in the Oil and Gas Domain.

Abstract

In recent years, there has been an increasing interest in extending stream processing engines with rule-based temporal reasoning capabilities. To ensure correctness, such systems must be able to output results over the partial data received so far as if the entire (infinite) stream had been available; furthermore, these results must be streamed out as soon as the relevant data is received, thus incurring the minimum possible latency; finally, due to memory limitations, systems can only keep a limited history of previous facts in memory to perform further computations. These requirements pose significant theoretical and practical challenges since temporal rules can derive new information and propagate it both towards past and future time points; as a result, streamed answers can depend on data that has not yet been received, as well as on data that arrived far in the past. Towards developing a solid foundation for practical rule-based stream reasoning, we propose and study in this thesis a suite of decision problems that can be exploited by stream reasoning algorithms to tackle the aforementioned challenges, and provide tight complexity bounds for a core temporal extension of Datalog. All of the problems we consider can be solved at design time (under reasonable assumptions), prior to the processing of any data. Solving these problems enables the use of reasoning algorithms that process the input streams incrementally using a sliding window, while at the same time supporting an expressive rule-based knowledge representation language and minimising both latency and memory consumption.

Contents

Chapter 1

Introduction

Continuous processing of data streams is a key aspect of many applications. For instance, algorithmic trading relies on real-time analysis of stock tickers and financial news items [40], oil and gas companies continuously monitor and analyse data coming from their wellsites in order to detect equipment malfunction and predict maintenance needs [28], and network providers perform real-time analysis of network flow data to identify traffic anomalies and DoS attacks [39].

In stream processing, an input data stream is typically seen as an unbounded, append-only, sequence of timestamped tuples, where timestamps are either added by the external device that issued the tuple or by the stream management system receiving it [7, 6]. Data is only available for processing in a single pass and information stored by the system is thus inherently incomplete. Streaming jobs are long-running: standing queries are deployed once and continue to produce results as a stream until removed. Most applications of stream processing require *near real-time* analysis using limited resources, which poses significant challenges to stream management systems. On the one hand, to ensure correctness, systems must be able to compute query answers over the partial data received so far as if the entire (infinite) stream had been available; furthermore, they must stream query answers out as soon as the relevant data is received, thus incurring the minimum possible latency. On the other hand, due to memory limitations, systems can only keep a limited history of previously received input facts in memory to perform computations. These issues have been addressed at a language design level, by extending existing query languages with

window constructs, which declaratively specify the finite part of the input stream relevant to the answers at the current time [5, 20, 10, 43, 32, 4, 22, 41]. Window constructs allow for selecting portions of the input stream that are recent with respect to the current time; for instance, the data for the last n time points. Explicit window constructs limit the expressivity of a language, for example they do not allow one to refer to arbitrarily large portions of the stream.

Rule-based temporal languages have received significant attention in the context of stream processing [52, 15, 14, 18, 17, 31]. These languages allow us to naturally extend rule-based reasoners to the streaming setting; they can be formalised as temporal extensions of Datalog [30, 2]—a prominent language with a rich tradition in the database and knowledge representation communities, which is frequently being used in advanced applications that mix AI and data management techniques. Datalog programs can be used to represent in a succinct and declarative way domain knowledge as "if-then" rules where, if all atoms in the antecedent of a rule hold, then the atom in the consequent part of the rule must also hold. An important feature of Datalog rules is that they can be recursive: the application of a rule can (possibly indirectly) trigger an application of the same rule.

For rule-based temporal languages, the challenges stated above have only partially been addressed. In [52] Zaniolo acknowledges that some queries expressed in a temporal extension of Datalog show a blocking behaviour, i.e., they may require to read the entire stream in order to produce any answer tuple. To address that, he proposes a syntactic condition ensuring that the data received so far is always enough to answer the query for the current time point.

In this thesis, we investigate rule-based stream reasoning, focusing on a fragment of $\text{Datalog}_{1\text{S}}$ [26, 12], which we call *Temporal Datalog*. This is a core temporal rule-based language, which captures other prominent languages [1, 12] and forms the basis of more expressive formalisms for stream reasoning proposed in the literature [52, 19].[1] Temporal Datalog can be seen as an extension of negation-free Datalog in

[1]The reasoning problems we propose in this thesis can be formulated and studied for other temporal rule-based languages, but this is something that we leave for future work.

which predicates are equipped with a special time sort interpreted over the integers, and allowing for terms of the form $t + k$ with t a time variable and k an integer. In contrast to $\text{Datalog}_{1\text{S}}$, the language we consider imposes a *guardedness* condition on time arguments. Intuitively, guardedness ensures that application of a rule to a stream is localised, in the sense that facts matching the antecedent of the rule cannot be arbitrarily far apart in the stream; this makes the language naturally well-suited for incremental stream processing. The following examples illustrate the use of Temporal Datalog rules in streaming applications.

Example 1.1. *Consider the management of a wind farm in the North Sea. Each turbine is equipped with a sensor, which continuously records temperature levels of key devices within the turbine and sends those readings to a data centre monitoring the functioning of the turbines. Temperature levels are streamed by sensors using a ternary predicate Temperature, whose arguments identify the device, the temperature level, and the time of the reading. A monitoring task in the data centre is to track the activation of cooling measures in each turbine, record temperature-induced malfunctions and shutdowns, and identify parts at risk of future malfunction. This task is captured by the following set of rules:*

$$\mathit{Temperature}(x, \mathit{high}, t) \rightarrow \mathit{Flag}(x, t) \tag{1.1}$$

$$\mathit{Flag}(x, t) \wedge \mathit{Flag}(x, t+1) \rightarrow \mathit{Cooling}(x, t+1) \tag{1.2}$$

$$\mathit{Cooling}(x, t) \wedge \mathit{Flag}(x, t+1) \rightarrow \mathit{Shutdown}(x, t+1) \tag{1.3}$$

$$\mathit{Shutdown}(x, t) \rightarrow \mathit{Malfunction}(x, t-2) \tag{1.4}$$

$$\mathit{Shutdown}(x, t) \wedge \mathit{Near}(x, y, t) \rightarrow \mathit{AtRisk}(y, t) \tag{1.5}$$

$$\mathit{AtRisk}(x, t) \rightarrow \mathit{AtRisk}(x, t+1) \tag{1.6}$$

Rule (1.1) *'flags' a device whenever a high temperature reading is received. Rule* (1.2) *says that two consecutive flags on a device trigger cooling measures. Rule* (1.3) *says that an additional consecutive flag after activating cooling measures triggers a pre-emptive shutdown. By Rule* (1.4)*, a shutdown is due to a malfunction that occurred when the first flag leading to shutdown was detected. Finally, Rules* (1.5) *and* (1.6)

identify devices located near a shutdown device as being at risk and propagate risk recursively into the future.

Example 1.2. *Consider a computer network which is being monitored for external threats using an intrusion detection policy (IDP). Bursts (unusually high amounts of data) between any pair of nodes in the network are detected by specialised monitoring devices and streamed to the network's management centre as timestamped facts. A monitoring task in the centre is to identify nodes that may have been hacked according to a specific IDP, and add them to a blacklist of nodes. In this setting, one may want to know the contents of the blacklist at any given point in time in order to decide on further action. One IDP is specified by a Temporal Datalog program consisting of the rules given next:*

$$Burst(x, y, t) \wedge Burst(z, y, t + 1) \rightarrow Attack(x, y, t + 1) \quad (1.7)$$

$$Attack(x, y, t) \wedge Attack(x, y, t + 1) \wedge Attack(x, y, t + 2) \rightarrow Blacklist(x, t + 2) \quad (1.8)$$

$$Blacklist(x, t) \rightarrow Blacklist(x, t + 1) \quad (1.9)$$

Rule (1.7) *identifies two consecutive bursts from nodes x and z to a node y in the network as an attack on y originated from x. Rule* (1.8) *implements an IDP where three consecutive attacks from x on y result in x being added to the blacklist, where it remains indefinitely—see Rule* (1.9)*. A second IDP is desribed by the following program, which makes again use of the notion of attack defined by Rule* (1.7)*:*

$$Attack(x, y, t) \rightarrow Greylist(x, red, t) \quad (1.10)$$

$$Greylist(x, red, t) \rightarrow Greylist(x, orange, t + 1) \quad (1.11)$$

$$Greylist(x, orange, t) \rightarrow Greylist(x, yellow, t + 1) \quad (1.12)$$

$$Greylist(x, l, t) \wedge Burst(x, y, t) \rightarrow Blacklist(x, t) \quad (1.13)$$

Rule (1.10) *adds any attacker x to a 'greylist'. Such a list comes with a succession of three decreasing warning levels: red, orange, and yellow. As time goes by, the warning level decreases—Rules* (1.11) *and* (1.12)*. However, if at any point during*

this process node x generates another burst to any other node in the network, then it gets blacklisted, by Rule (1.13).

Example 1.3. *Consider a match of chess. At the end of each* half-move*—i.e., one player's move—a camera takes a picture of the board, and an image processing software detects which squares are unoccupied, which occupied by white pieces and which by black pieces. The information from the camera is made available as facts of the form $Unoccupied(e4, \tau)$ meaning that square $e4$ is occupied by no piece at time τ, and facts of the form $Occupied(e4, white, \tau)$ meaning that square $e4$ is occupied by a white piece at time τ. Time corresponds to half-moves, and we assume that the input contains exactly one fact $Start(\tau_0)$ to specify that τ_0 is the time point corresponding to the first half-move. The following program defines complex concepts starting from the information available.*

$$Start(t) \rightarrow White(t) \tag{1.14}$$

$$White(t) \rightarrow Black(t+1) \tag{1.15}$$

$$Black(t) \rightarrow White(t+1) \tag{1.16}$$

$$Occupied(x,y,t) \wedge Occupied(x,y,t-1) \rightarrow NotMoved(x,t) \tag{1.17}$$

$$Occupied(x,y,t) \wedge Unoccupied(x,t-1) \rightarrow Moved(x,t) \tag{1.18}$$

$$White(t) \wedge Occupied(x,black,t) \wedge Moved(x,t) \rightarrow Error(t) \tag{1.19}$$

$$Black(t) \wedge Occupied(x,white,t) \wedge Moved(x,t) \rightarrow Error(t) \tag{1.20}$$

$$Error(t) \rightarrow Error(t+1) \tag{1.21}$$

$$Occupied(x,black,t-1) \wedge Occupied(x,white,t) \rightarrow Captured(x,black,t) \tag{1.22}$$

$$Occupied(x,white,t-1) \wedge Occupied(x,black,t) \rightarrow Captured(x,white,t) \tag{1.23}$$

$$Moved(x,t) \wedge Captured(x,y,t+1) \rightarrow Sacrificed(y,t) \tag{1.24}$$

$$NotMoved(x,t) \wedge Captured(x,y,t+1) \rightarrow NotRescued(y,t) \tag{1.25}$$

Rules (1.14)–(1.16) *allow us to infer who moves in each half-move. Rule* (1.17) *specifies whether the piece in a certain square has not been moved in the current half-move, and Rule* (1.18) *specifies whether the piece in a certain square has been moved in the*

current half-move. Rules (1.19)–(1.21) *capture the fact that, when the input data says that a player moved a piece during the half-move of the other player, there is an error in the input data—either the information about the initial half-move is wrong, or there has been an error in interpreting the picture of the board. Rules* (1.22) *and* (1.23) *describe when a piece is captured. Rule* (1.24) *marks the half-moves in which a piece of a certain colour has been sacrificed, i.e., it has been moved to a square where it has been captured right at the next half-move. Rule* (1.25) *marks the half-moves in which a piece of a certain colour has not been rescued, i.e., it has been left in a square where it has been captured at the next half-move.*

As already mentioned, stream processing applications require near real-time response using limited resources. This becomes especially challenging in the context of rule-based stream reasoning since, as seen in our examples, rules may derive new information and recursively propagate it both towards past and future time points. As a result, the output of a Temporal Datalog program at a particular time point τ can depend on data that has not yet been received, as well as on data that arrived far back in the past. This can critically handicap the design of a near real-time stream reasoning system, due to the following reasons.

- The system may not be able to determine whether all facts for time point τ have already been derived, thus introducing a (potentially unbounded) delay on applications that rely on the availability of all such facts.
- The system may be forced to store a (potentially unbounded) input history to ensure correctness, thus precluding the use of efficient stream processing techniques based on a sliding window.

Towards developing a solid foundation for practical rule-based stream reasoning, we propose and study in this thesis a suite of decision problems that can be exploited by stream reasoning algorithms to tackle the aforementioned challenges.

1.1 Contributions

We study stream reasoning algorithms that process data incrementally with respect to time. Namely, an algorithm reads data and outputs logical consequences for time τ before moving on to time $\tau + 1$.

Rules have the capability (also through recursion) to propagate information over time, both towards the past and the future. The two kinds of propagation have a very different impact on stream reasoning, simply because the past is known and the future is unknown. Propagation towards the past means that the truth of a fact at a certain point may depend on the truth of some fact in the future; therefore, during the computation, we may not yet have enough data to determine whether a fact holds, and hence we must buffer more elements of the input stream before producing the next bit of the output stream. Thus, we introduce the notion of *valid delay*, which is the maximum time difference between a fact and any other explicit (i.e., in the input data) fact that is necessary to determine the truth of the former fact. Once a valid delay for a program is determined, it suffices to buffer data for a constant (and known) number of time points in order to compute the logical consequences at a certain time.

Symmetrically, propagation towards the future means that the truth of a fact at a certain point may depend on the truth of a fact in the past; making it challenging to safely discard past data. We observe that all the relevant information from the past can be summarised in its most recent consequences, i.e., in order to correctly compute the remaining consequences it suffices to maintain a full materialisation of the consequences over a sliding window of a limited number of most recent time points and safely discard older data. Thus, we introduce a notion of *valid window size*, which is the number of most recent time points for which to retain consequences in memory. Furthermore, our notion of valid window size enables to reason over a restricted timeline that begins from the first point in the window, as if the time before the window did not exist. This property allows for a cost of reasoning at each step that is independent of the number of time points processed so far.

We show that the notions of delay and window size validity enable stream reasoning, by presenting a generic stream reasoning algorithm that takes a delay and a window size as parameters. The delay parameter determines the logical delay with which the algorithm materialises and outputs consequences, and the window size parameter determines the size of the sliding window that the algorithm uses to determine when to discard stored facts. The algorithm, when parametrised with a valid delay and window size, computes all the consequences of an input Temporal Datalog program on an input stream. We analyse the suitability of the algorithm for near-real time stream processing, giving a bound on the time to compute all the consequences for a certain time point τ from the moment the input facts for τ are read, showing that such time grows at most logarithmically with the value of the considered time point τ; this implies that the performance of the algorithm remains mostly stable as it keeps processing the stream.

We show that programs admitting a valid delay always admit a valid window size that is linear in the size of the program and can be easily computed. Furthermore, such window size is optimal for programs that do not refer to the future and where no rule is redundant. We identify a class of programs where the temporal backward propagation (i.e., towards the past) is obviously bounded, as it can be established by a simple graph analysis. Such programs, that we call *backward-bounded*, always admit a valid delay that is linear in the size of the program and can be easily computed.

We propose and study a suite of decision problems that capture the task of finding the minimum valid delay and window size for a given program. Specifically, such a task subsumes each of the decision problems we consider, and can be solved worst-case optimally by combining worst-case optimal algorithms for the decision problems. The decision problems we consider are: valid delay existence, delay validity, and window size validity—we do not formulate valid window size existence as a decision problem, since it is trivial. We show that the three decision problems are undecidable in the general case, and we show decidability and tight complexity bounds under the assumption that the cardinality of the object domain of input streams is fixed. As

an intermediate result, we show the complexity of containment of Temporal Datalog programs under the same assumption on the object domain.

We believe that our results constitute a first step towards the development of robust and scalable stream reasoning engines with provable correctness guarantees. Although all of the problems we consider are computationally intractable, they can be solved 'offline' at design time prior to the processing of any data. Solving these problems enables the use of reasoning algorithms that process the input streams incrementally using a sliding window, while at the same time supporting an expressive rule-based knowledge representation language and minimising both latency and memory consumption.

This thesis builds on some of the results presented in conference publications [45, 44].

1.2 Outline

We give a brief description of the contents of the thesis.

The preliminaries are given in Chapter 2. There, we define the language considered in this thesis, Temporal Datalog; furthermore, we define two fundamental reasoning problems relevant to our results, namely fact entailment and program containment; finally, we define derivations, which are a useful tool to characterise entailment of a fact.

Chapters 3–5 present our original contributions. In Section 3.1 we formalise the stream reasoning setting. Specifically, we formalise the notions of stream, stream reasoning algorithm, and latency, which aims at capturing the near real-time capabilities of a stream reasoning algorithm. In Section 3.2 we present a stream reasoning algorithm that extends the traditional stream processing techniques based on a sliding window to the stream reasoning setting, while at the same time abstracting away from all implementation details. The algorithm is parametrised with a delay d and a window size w. It accepts as input a Temporal Datalog program Π and a stream S of timestamped facts, and outputs as a stream a subset of the logical consequences

of $\Pi \cup S$, which is determined by the values of the parameters d and w. As the algorithm receives and stores in memory the input stream at time point τ, it computes all implicit facts holding at $\tau - d$ using only the facts currently held in memory. The computed facts are first added to the memory and subsequently streamed as part of the output. Finally, the algorithm updates the memory by discarding all facts holding at $\tau - w$. We show that the latency of the algorithm while computing the consequences with timestamp up to τ depends only logarithmically on τ and, in addition to that, it depends on the delay d, the window size w, and the number of objects mentioned in the input stream. Having a latency up to τ that depends sublinearly on τ is crucial for a stream reasoning algorithm, as it means that its performance does not deteriorate as it keeps processing the stream. For the algorithm to be correct, the output stream must include all logical consequences of $\Pi \cup S$, for any input stream S. Such an assurance, however, can only be given for certain values of the delay and window size parameters. Hence, in Section 3.3, we formally define the notions of a valid delay d and a valid window size w as properties of the program Π, which are independent from the input stream S, and show that the algorithm parametrised with d and w is correct on an input program Π and each input stream S if and only if d and w are valid for Π.

In Chapter 4 we look for syntactic conditions under which it is easy to compute a (possibly minimum) valid delay and window size. We show in Section 4.2 that every program with a valid delay has a valid window size, which is linear in the size of the program and easily computable—note that a valid delay does not always exist though. We show in Section 4.2.1 that for forward-propagating programs (that propagate information only forward into the future) the syntactic condition above yields the minimum window size if the program contains no redundant rule—i.e., a rule that can be removed without affecting the meaning of the program. In Section 4.1 we show a class of programs which have a backward propagation (i.e., into the past) that is syntactically bounded; we call such programs *backward-bounded*.

In Chapter 5 we study the computation and decision problems associated with delay and window size validity. A program may not admit a valid delay, but it

will always admit a valid window size whenever it admits a valid delay. Thus, the computational task of interest in our setting is to first check whether a program Π admits a valid delay and, if it does, to then compute the corresponding minimum delay and window size values. We will focus on the following decision problems:

- Delay existence: Given program Π, decide whether there is a valid delay for Π.
- Delay validity: Given program Π and $d \geq 0$, decide whether d is a valid delay for Π.
- Window size validity: Given program Π, a valid delay d for Π, and $w \geq d$, decide whether w is a valid window size for Π and d.

The three decision problems above are clearly subsumed by (and hence not easier than) our overall task, and we will show that worst-case optimal algorithms for the three problems can be combined to obtain a worst-case optimal algorithm for the overall computation problem. In Section 5.2, we explore the limitations of our approach and establish a number of undecidability results. For this, we show that the decision problems considered in this thesis are closely related to *program containment*—a fundamental problem in static analysis and query optimisation, which is well-known to be undecidable already for non-temporal Datalog programs. To regain decidability, we consider the situation where the maximum number of domain objects that can occur in a stream is fixed. This assumption allows us to ground the object variables of the program to a bounded number of (representative) objects; such grounding is exponential and results into an object-ground program where all variables are time variables—which can also be restated as an object-free program. In Section 5.3, we use automata-based techniques to show that delay existence, delay validity, and window size validity are solvable in PSpace for object-free programs; this immediately yields an ExpSpace upper bound for Temporal Datalog programs under the assumption that there is a known bound on the cardinality of the object domain of input streams. Finally, in Section 5.4, we show that these results are tight, by providing matching lower bounds.

In Chapter 6 we discuss the related work, and in Chapter 7 we draw our conclusions, presenting interesting future work.

Chapter 2

Preliminaries

We introduce next the rule-based language considered in this thesis, two relative reasoning problems, and derivations with some related notions. Throughout the thesis we assume familiarity with the basic concepts of complexity theory, logic, and rule-based languages for databases and knowledge representation.

2.1 Temporal Datalog

We recapitulate the syntax and semantics of Datalog with a time argument, which we call *Temporal Datalog*. This language can be seen as a fragment of $\text{Datalog}_{1\text{S}}$ as defined by Chomicki and Imieliński [26].

We assume mutually disjoint and countably-infinite sets of *predicates* (written P, Q, R, etc.), *objects* (written a, b, c, etc.), *object variables* (written x, y, z, etc.), and *time variables* (written t, t_1, t_2, etc.). Each predicate comes with a non-negative arity; as usual, we assume an infinite supply of predicates of each arity.

A *constant* is either an object or a *time point* $\tau \in \mathbb{N}$ (where $\mathbb{N}$ includes zero). An *object term* is an object or an object variable. A *time term* is a time point or an expression of the form $t + k$ with t a time variable and k an integer (the *offset*); we write $t - k$ for $t + k'$ with k' the opposite of k, and we abbreviate $t + 0$ as t. We assume that time points and offsets are coded in unary unless stated otherwise; the rationale is that Temporal Datalog is meant to be a close syntactic variant of $\text{Datalog}_{1\text{S}}$ and, specifically, Temporal Datalog's time terms are supposed to be an easier-to-write

(and easier-to-read) variant of $\text{Datalog}_{1\text{S}}$'s time terms, which are implicitly coded in unary.[1] An *atom* α is an expression of the form $P(s_1, \ldots, s_n)$ where P is an n-ary predicate, each s_i for $1 \leq i \leq n-1$ is an object term, and s_n is a time term; each s_i is called an *argument* of α.

A *rule* r is a first-order sentence of the form (2.1), where the *body* φ of r is a conjunction of atoms and the *head* α is an atom:

$$\forall \mathbf{x}.\varphi \rightarrow \alpha. \tag{2.1}$$

As usual, we will omit universal quantifiers when writing rules and require that rules are *safe*, i.e., that each variable in a rule occurs in its body. A rule is *temporally-guarded* if there exists a time variable that occurs in every atom of the rule. The *forward radius* of a temporally-guarded rule r of the form (2.1) is the maximum between zero and $k-k'$ for k the offset in α and k' the minimum offset of an atom in φ; similarly, the *backward radius* of r is the absolute value of the minimum between zero and $k-k''$, with k'' the maximum offset of an atom in φ. A temporally-guarded rule is *forward-propagating* if its backward radius is zero. Intuitively, temporal guardedness ensures that an application of a rule to a stream is localised, in the sense that the time arguments of all facts in the stream matching the antecedent of the rule are close to each other and to the head that gets derived (with the maximum distance depending on the radius of the rule).

Example 2.1. *Consider rule* $r = P(t_1) \wedge Q(t_2) \rightarrow R(t_2)$. *This rule is not temporally-guarded as it contains two different time variables in the body, where only one of them is mentioned in the head. Intuitively,* r *derives a fact* $R(\tau)$ *for a time point* τ *if* Q *holds at the same time point and* P *holds* somewhere *in the stream. Thus, fact* $R(\tau)$ *may be justified by a fact about* P *that appears arbitrarily far away into the future, and hence a stream reasoning algorithm would need to wait indefinitely in order to be certain as to whether* $R(\tau)$ *holds or not.*

[1] In $\text{Datalog}_{1\text{S}}$ [26], time terms are built out of a unary function s, which intuitively stands for the successor function. Specifically, a $\text{Datalog}_{1\text{S}}$'s time term is the constant 0, a time variable, or the term $s(u)$ for u a time term. A time point τ is then represented by τ applications of the successor function to 0, i.e, $s(s(\ldots s(0)\ldots))$, and a time term $t+k$ is represented by k applications of the successor function to t, i.e, $s(s(\ldots s(t)\ldots))$.

By contrast, all rules in our Examples 1.1–1.3 are temporally-guarded.

A (Temporal Datalog) *program* is a finite set of temporally-guarded rules. Note that when a set of rules contains a rule that is not temporally-guarded, we refer to it simply as a *set of rules*, reserving the term 'program' for sets of temporally-guarded rules. As customary in the database literature, we distinguish between *extensional* (EDB) and *intensional* (IDB) predicates, where only the former can occur in the data and only the latter can occur in rule heads. Formally, a predicate is IDB in a program Π if it occurs in the head of a rule of Π, and it is EDB otherwise. An atom is IDB (respectively, EDB) in Π if so is its predicate; we will often not mention Π when referring to EDB or IDB predicates or atoms when it is clear from the context. A term, atom, rule, or set of rules is *ground* if it has no variables, *object-ground* if it has no object variables, and *object-free* if it has no object terms. A program is *forward-propagating* if so are all of its rules.

A *fact* is a ground atom. When a fact α has time argument τ, we often say that α *holds at* τ. Each fact α corresponds to a rule having empty body and α as the head, so we use α and its corresponding rule interchangeably. For F a set of facts and T an interval of the integers, the *restriction* of F to T, denoted by $F\restriction_T$, is the subset of facts in F whose time argument is in the interval T, that is, $F\restriction_T = \{P(\mathbf{a}, \tau) \in F \mid \tau \in T\}$; we write $F\restriction_\tau$ for $F\restriction_{[\tau,\tau]}$. For F a set of facts and n a non-negative integer, the *left* and the *right shifting* of F by n are defined as in (2.2) and (2.3), respectively.

$$F \ll n \;:=\; \{P(\mathbf{o}, \tau - n) \mid P(\mathbf{o}, \tau) \in F\restriction_{[n,\infty)}\} \tag{2.2}$$

$$F \gg n \;:=\; \{P(\mathbf{o}, \tau + n) \mid P(\mathbf{o}, \tau) \in F\} \tag{2.3}$$

A *substitution* σ is a finite partial mapping of object variables to object terms and of time variables to time variables and time points. For A a term, an atom, a rule, or a set thereof, $A\sigma$ denotes the result of replacing each variable x in A and in the domain of σ with $\sigma(x)$, and then replacing each expression of the form $\tau + k$ for τ a time point, with the time point τ' being the sum of τ and k. We sometimes write $\{v \mapsto u\}$ for the substitution of v with u. The expression $A\sigma$ is called an *instance* of A.

An *interpretation* is a set of facts—and hence we interpret constants as themselves. An interpretation $\mathcal{I}$ *satisfies* a fact α, written $\mathcal{I} \models \alpha$, if $\alpha \in \mathcal{I}$; interpretation $\mathcal{I}$ satisfies a conjunction of facts if it satisfies each of the facts, and it satisfies a rule of the form (2.1) if, for each ground instance $\varphi\sigma \to \alpha\sigma$ of the rule, we have that $\mathcal{I} \models \varphi\sigma$ implies $\mathcal{I} \models \alpha\sigma$; interpretation $\mathcal{I}$ satisfies a set of rules if it satisfies each of the rules. For A a fact, a rule, or a set of rules, an interpretation $\mathcal{I}$ is a *model* of A if $\mathcal{I}$ satisfies A; and a set of rules B *entails* A, written $B \models A$, if each model of B is a model of A. As customary in the treatment of database query languages, we often see a program Π as a transformation that maps each set F of facts to the set $\Pi(F)$ of all IDB facts that are entailed by $\Pi \cup F$.

How Temporal Datalog captures Datalog. Temporal Datalog is not a syntactic extension of Datalog, as it does not allow for atoms without a time argument; it is, however, straightforward to transform any Datalog program into an equivalent (forward-propagating) Temporal Datalog program. It suffices to 'temporalise' every atom by extending its list of arguments with a fresh time variable t, that amounts to a dummy time argument. Thus, using a fresh unary predicate T and a fresh n-ary predicate P^{T} for each $(n-1)$-ary predicate P, we transform an arbitrary Datalog program Π into a Temporal Datalog program Π^{T} by replacing each atom $P(\mathbf{s})$ with the atom $P^{\mathrm{T}}(\mathbf{s}, t)$ and then extending each rule body with the atom $T(t)$—to ensure rule safety with respect to t. The resulting program Π^{T} is equivalent to Π in the following sense:

- for each set D of *non-temporal facts*,[2] each non-temporal fact $G(\mathbf{a})$, and each time point τ,

$$G(\mathbf{a}) \in \Pi(D) \;\Rightarrow\; G^{\mathrm{T}}(\mathbf{a}, \tau) \in \Pi^{\mathrm{T}}(D^{\mathrm{T}})$$

  where D^{T} consists of $T(\tau)$ together with each $P^{\mathrm{T}}(\mathbf{o}, \tau)$ for $P(\mathbf{o}) \in D$;

- for each set D^{T} of facts and each fact $G^{\mathrm{T}}(\mathbf{a}, \tau)$,

$$G^{\mathrm{T}}(\mathbf{a}, \tau) \in \Pi^{\mathrm{T}}(D^{\mathrm{T}}) \;\Rightarrow\; G(\mathbf{a}) \in \Pi(D)$$

[2] A non-temporal atom/fact, is like an atom/fact but without the time argument.

where D consists of each $P(\mathbf{o})$ for $P^{\mathrm{T}}(\mathbf{o}, \tau) \in D^{\mathrm{T}}$.

Program Π^{T} does not feature time offsets—variable t is the only time argument in the program—and hence Π^{T} is clearly forward-propagating. We will make use of this fact in some of our undecidability results.

2.2 Standard Reasoning Problems

Fact entailment is the problem of checking $\Pi \cup D \models \alpha$ for a given program Π, a finite set of EDB facts D, and a fact α as input. The *data complexity* of fact entailment is the complexity when Π and α are considered fixed and only D is considered as the input. Fact entailment in $\mathrm{Datalog}_{1\mathrm{S}}$ is PSPACE-complete in data complexity [26], and the same bounds apply to Temporal Datalog as defined in this thesis. On the one hand, Temporal Datalog can be seen as a fragment of $\mathrm{Datalog}_{1\mathrm{S}}$ where a term $t + k$ corresponds to k applications of the successor function symbol to t; on the other hand, our temporal guardedness does not make standard reasoning easier: using essentially the same complexity lower bound proofs as for $\mathrm{Datalog}_{1\mathrm{S}}$ it is immediate to show that fact entailment over Temporal Datalog remains PSPACE-hard in data complexity.

In addition to fact entailment, we will also consider the *program containment problem* for input programs Π_1 and Π_2 with the same set of EDB predicates. We say that Π_1 is *contained* in Π_2, written $\Pi_1 \sqsubseteq \Pi_2$, if $\Pi_1(D) \subseteq \Pi_2(D)$ for each set D of EDB facts and each fact α; then, *program containment* is the problem of checking $\Pi_1 \sqsubseteq \Pi_2$ given Π_1 and Π_2 as input. Note that our definition of containment considers possibly infinite sets of facts, which is required to capture streams. This does not change the nature of the problem—due to the compactness and monotonicity properties of first-order logic, the well-known undecidability result for the containment of Datalog programs (with respect to finite sets of facts) transfers to our setting [33, 46].

2.3 Derivations

Entailment of a fact α by a set A of rules can be characterised in terms of existence of a *derivation* of α from A, where such derivation δ is a finite node-labelled tree

satisfying the following properties: (i) each node is labelled with a ground instance of a rule in A; (ii) fact α is the head of the rule labelling the root; and (iii) for each node v, the body of the rule labelling v contains an atom β if and only if β is the head of a rule labelling a child of v. A *β-subderivation* of δ is a subtree of δ that is itself a derivation of β. We say that the subderivation is *strict* if it is not δ itself, and it is an *immediate subderivation* of δ if it is rooted in a child of the root of δ. In some of our constructions later on in the thesis, it will be useful to consider *partial derivations*, where condition (iii) is weakened to only require that each head of a rule labelling a non-root node v must occur in the body of the rule labelling the parent of v. We then say that a fact α *depends* on a fact β in a set A of rules (via $n \geq 1$ steps) if there is a partial derivation of α from $A \cup \{\beta\}$ having height n; and that α has *rank* k in A if it depends on no fact in A via more than k steps.

Chapter 3

Stream Reasoning

In this chapter we first formalise the concepts of stream, stream reasoning algorithm, and latency of an algorithm. Then, we present a stream reasoning algorithm that is parametrised with a delay and a window size. Finally, we introduce the notions of delay and window size validity, and show that they characterise the correctness of our stream reasoning algorithm. Delay and window size validity are central concept of this thesis.

3.1 The Stream Reasoning Setting

We start by providing a formal definition of a *stream*. We see a stream as a possibly infinite set of timestamped facts with a finite restriction to each time point.

Definition 3.1. *A stream is a (possibly infinite) set of facts S such that $S\restriction_\tau$ is a finite set for each time point τ. The* object domain *of S is the set of objects occurring in S. An* EDB stream *(respectively,* IDB stream*) for a program Π is a stream where all facts are EDB (respectively, IDB) with respect to Π; we will often omit the reference to Π where it is clear from the context.*

Stream reasoning algorithms process streams incrementally over time, as described in the following definition, where we write a sequence $a_0, a_1, \ldots, a_n, \ldots$ as $(a_n)_{n\geq 0}$.

Definition 3.2. *A stream reasoning algorithm is an algorithm that: (i) takes as input a program Π and a stream S, (ii) reads S as the sequence $(S\restriction_\tau)_{\tau\geq 0}$ once and in order, and (iii) outputs $\Pi(S)$ as the sequence $(\Pi(S)\restriction_\tau)_{\tau\geq 0}$.*

Although facts may arrive to the system out of order, an ordered sequence can be generated by exploiting techniques (such as low watermarks [3]) to check whether all facts up to a given time point have been received.

In order to evaluate the near real-time capabilities of a stream reasoning algorithm we define the notion of latency. In the following definition, we mean time as computation time, e.g., steps of a Turing machine that implements the algorithm.

Definition 3.3. *The* latency up to τ *of a stream reasoning algorithm on an input program* Π *and stream* S *is the maximum time, for any* $\tau' \leq \tau$*, the algorithm takes to output all the facts with time argument* τ' *from the moment it reads* $S\restriction_{\tau'}$.

The latency is defined relatively to a time point τ in order to describe the trend of the performance as the value of τ increases. In particular, a weak dependency on τ implies that the performance of the algorithm remains mostly stable as it keeps processing the stream.

Time points in input streams are assumed to be coded in unary by default. However, we will show that the latency up to τ of the algorithm presented in the next section has a dependency on τ that provably drops from linear to logarithmic when we switch from unary coding to binary coding of time points in input streams.

3.2 A Generic Stream Reasoning Algorithm

Algorithm 1 is a generic algorithm that relies on the notion of a sliding window to process the input stream incrementally, while at the same time abstracting away from many details that are not fundamental to the overall approach. The algorithm is parametrised by a non-negative *delay* d and *window size* w. The delay parameter directly influences the latency of the algorithm, as it determines when the derived IDB facts are streamed out; in turn, the window size parameter w determines the stored facts at any moment, hence the memory consumption and the facts we have to reason about in each iteration—and hence it affects the latency indirectly. Both parameters have an impact on the correctness of the algorithm. The algorithm is

Algorithm 1: Stream reasoning algorithm based on a sliding window.

Parameters: integers d (the delay) and w (the window size) with $w \geq d \geq 0$.
Input: program Π, EDB stream S for Π.
Output: IDB stream for Π.

```
1  Initialise τ := 0, M := ∅;
2  loop
3      Receive S⌈τ and set M := M ∪ S⌈τ;
4      if τ < w then
5          M := M ∪ Π(M)⌈τ−d;
6      else
7          M := M ∪ (Π(M ≪ τ − w)⌈w−d ≫ τ − w);
8      Stream out all IDB facts in M⌈τ−d;
9      Remove from M all facts in M⌈τ−w;
10     τ := τ + 1;
```

initialised in Line 1, where the variable M is set empty and the "current time" τ is set to zero. The core of the the algorithm is an infinite loop, where each iteration of the loop processes a single time point τ in the input; the current time τ is incremented at the end of each iteration. More precisely, each iteration of the main loop consists of the following steps.

1. The set of all input stream facts holding at τ is received and loaded into memory (Line 3).

2. All the entailed IDB facts holding at $\tau - d$ are computed and stored in memory (Lines 4–7).

3. All IDB facts holding at $\tau - d$ are read from memory and streamed as part of the output (Line 8).

4. All facts (EDB or IDB) holding at $\tau - w$ are removed from memory (Line 9).

The first important feature of the algorithm is that "old" facts falling outside the sliding window are discarded in Line 9 and never reconsidered again; this has the key advantage of limiting the number of facts that the algorithm needs to store and reason about at any point in time, and hence favours low latency. The use of a sliding

window, however, carries the risk of missing IDB facts α entailed by $\Pi \cup S$ if the facts that α depends on are removed from memory too early.

The second feature of the algorithm is that it proceeds by computing and outputting consequences with a "logical" delay of d time points with respect to the current time τ, i.e, it outputs consequences for $\tau - d$ in the iteration where it reads input facts for τ. The delay is logical in the sense that it is expressed in terms of the time points in the data. Such delay can be translated into computation time—and hence a latency—by taking into account the time required to perform the relevant iterations. In particular, the latency up to τ of Algorithm 1 is given by the maximum time to perform d consecutive iterations among the first $\tau + 1$ iterations. Thus, a bound on the time to perform a single iteration immediately implies a bound on the latency. However, this approach carries the risk of missing IDB facts α with time argument τ entailed by $\Pi \cup S$ if the facts in S that α depends on have not been received yet by the time the algorithm generates the output for τ.

The third important feature of the algorithm is that it keeps in memory a complete materialisation of the past portion of the window—that is, all input EDB facts and entailed IDB facts for the relevant time points. Computing and incrementally maintaining a full materialisation is a common reasoning approach adopted by many rule-based systems [37, 36, 38, 35, 8]. Storing entailed IDB facts allows an algorithm to process programs that could not be processed otherwise by means of a sliding window, as shown in the following example.

Example 3.4. *The program consisting of rules* $A(t) \rightarrow B(t)$ *and* $B(t) \rightarrow B(t+1)$ *can be processed by Algorithm 1 with delay zero and window size one, since it can infer any* $B(\tau)$ *simply by checking whether* $A(\tau)$ *or* $B(\tau - 1)$ *are in memory. If the algorithm did not retain IDB consequences from one iteration to the next one, then it would not have* $B(\tau - 1)$ *in memory, and would need to check whether some* $A(\tau')$ *occurs in the stream for* $\tau' \leq \tau$*. A sliding window of constant size would not suffice for that check.*

The fourth feature of the algorithm is that, instead of computing directly the consequences at $\tau - d$ from the data stored in memory, it first left-shifts the stored facts by $k = \max(0, \tau - w)$, then computes the consequences at $\tau - d - k$—that is $\tau - d$ in the fist w iterations and $w - d$ in the following ones—and finally shifts back the computed consequences, i.e., it right-shifts them by k. Reasoning over left-shifted data amounts to restricting reasoning to an interval of the timeline, as stated in Theorem 3.6 below. We first define what it means to restrict reasoning to an interval of the timeline.

Definition 3.5. *For Π a program (seen as a transformation of sets of facts) and T an interval of the integers, the* restriction *of Π to T, written $\Pi{\restriction_T}$, is the transformation that maps each set F of facts to the set $\Pi{\restriction_T}(F)$ consisting of each IDB fact α for which there exists an instance Π' of Π where every time argument is a time point in T and such that $\alpha \in \Pi'(F{\restriction_T})$.*[1]

The facts in $\Pi{\restriction_T}(F)$ are the consequences that we obtain by restricting reasoning to the time interval T. In fact, taking $T = \mathbb{N}$ amounts to no restriction, i.e., $\Pi{\restriction_{\mathbb{N}}}(F) = \Pi(F)$. Now we can express reasoning over left-shited data in terms of reasoning over an interval of the timeline.

Theorem 3.6. *For Π a program, D a set of facts, and τ a time point, we have that $\Pi{\restriction_{[\tau,\infty)}}(D) = \Pi(D \ll \tau) \gg \tau$.*

The shifting technique allows one to reason over a set of facts where time points have always value at most w. In fact, if we were computing consequences at each iteration as in Line 5 (i.e., without the shifting operations), we would have to reason over data where the value of time points keeps growing with the number of iterations of the algorithm. Reasoning over facts with a bounded value of their time arguments allows us to achieve a latency up to τ that depends only linearly on the *size* of the largest time point we have read so far, hence that depends on the size $\|\tau\|$ of τ in

[1]The operator $\restriction$ is overloaded. Specifically, the expression $A{\restriction_T}$ yields a set of facts when A is a set of facts, and it yields a transformation from and to sets of facts when A is a program. In both cases it has the intuitive effect of restricting A to the time interval T.

the worst-case. Although we generally assume that time points are coded in unary (and hence $\|\tau\| = \tau$), here we additionally show a significantly better bound on the latency for the case where *binary coding* is assumed (hence $\|\tau\| = \log \tau$).

Theorem 3.7. *The latency up to τ of Algorithm 1 with parameters d and w on a fixed input program Π and input stream S is*

$$\mathcal{O}(2^{(w \cdot n)^c} \cdot \|\tau\|)$$

for some positive integer c, where n is the cardinality of the object domain of S, and $\|\tau\| = \tau$ if we assume unary coding of time points in S and $\|\tau\| = \log \tau$ if we assume binary coding.[2]

Proof. The latency up to τ of Algorithm 1 is bounded by the maximum time taken to perform any $d+1$ consecutive iterations of the main loop within the first $\tau + d + 1$ iterations; so the latency is at most $d + 1$ times the maximum time to perform any such iteration, which we bound next.

We first bound the time to materialise consequences, i.e., the time taken by Lines 4–7 if we exclude the shifting operations. The materialisation time is bounded by the number of entailment checks times the cost of each check. The number of relevant entailment checks corresponds to the number of facts that may be entailed at a single time point, which is $\mathcal{O}(n^a)$ for a the maximum arity of a predicate of Π. For the cost of a single fact entailment check, we first bound the size of an input to a fact entailment check (w.r.t. the fixed program Π), and then give the overall bound based on that. The maximum value of a time point in the input to a check is w (due to shifting), and hence the size of such input is $\mathcal{O}(w \cdot n^a \cdot (w + \log n))$. Since fact entailment is in PSPACE in data complexity, we have that a single fact entailment check takes time $\mathcal{O}(2^{(w \cdot n^a \cdot (w + \log n))^b})$ for some constant b.

The remaining operations in an iteration can be performed in time $\mathcal{O}(w \cdot n^a \cdot (\|\tau + d\| + \log n)$; in particular, the involved integers (including time points) have maximum size $\|\tau + d\|$, and the involved facts are over at most n objects and $w + 1$ time points.

[2] A function $f(x_1, \ldots, x_n)$ is $\mathcal{O}(g(x_1, \ldots, x_n))$ if there exist two positive integers a and b such that $f(x_1, \ldots, x_n) \leq a \cdot g(x_1, \ldots, x_n)$ for every $x_1, \ldots, x_n \geq b$.

Then, the overall bound is:

$$\mathcal{O}\Big(d \cdot \Big(n^a \cdot 2^{(w \cdot n^a \cdot (w+\log n))^b} + w \cdot n^a \cdot (\|\tau + d\| + \log n)\Big)\Big) =$$
$$\mathcal{O}\left(d^2 \cdot w \cdot n^{1+2 \cdot a} \cdot 2^{(w^2 \cdot n^{a+1})^b} \cdot \|\tau\|\right) =$$
$$\mathcal{O}\left(2^{(w \cdot n)^c} \cdot \|\tau\|\right)$$

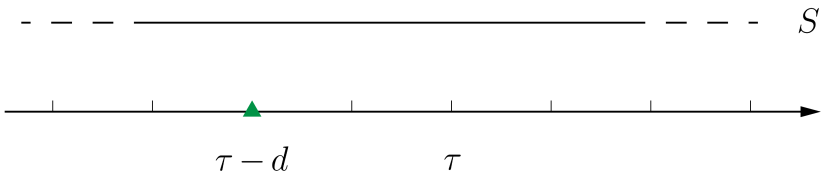
where the first equality holds by the identities $\mathcal{O}(x + y) = \mathcal{O}(x \cdot y)$ and $\log(x \cdot y) = \log x + \log y$, and the second equality holds because $w \geq d$ by the definition of Algorithm 1. □

3.3 Delay and Window Size Validity

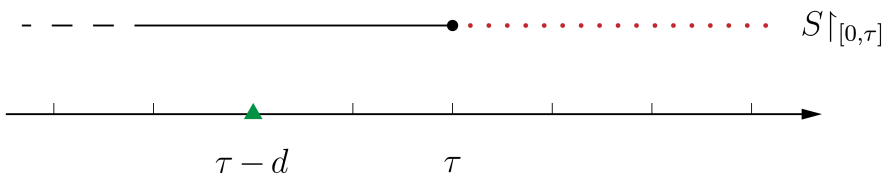
The choice of the algorithm's parameters determines its correctness, as shown in the following example.

Example 3.8. *Consider Algorithm 1 parametrised with delay zero and window size zero on input program* $\Pi = \{P(t) \to Q(t-5)\}$ *and stream* $S = \{P(10)\}$*. We have that* $\Pi \cup S \models Q(5)$*, so the algorithm is required to eventually output* $Q(5)$*. The algorithm, however, will output all consequences for* $\tau = 5$ *in the sixth iteration of the main loop (and will never attempt to compute them again later on); in this iteration, no input fact will have been received yet and the memory* M *will be empty as a result. Thus, fact* $Q(5)$ *will not be returned. By contrast, Algorithm 1 parametrised with delay five and window size five, on the same input, will output* $Q(5)$ *in the eleventh iteration of the main loop. Similarly, Algorithm 1 parametrised with delay zero and window size zero on input program* $\Pi' = \{P(t) \to Q(t+5)\}$ *and the same stream* S *will not output* $Q(15)$ *even though* $\Pi' \cup \{P(10)\} \models Q(15)$*; in order for the algorithm to return* $Q(15)$*, the window size parameter has to be at least five.*

We next define validity of a delay d for a program Π. Intuitively, d is a valid delay if, in order to compute the logical consequences of $\Pi \cup S$ up to time $\tau - d$, one does not need to consider any future facts in S with timestamp exceeding τ. As a result,



(a) The figure represents the elements involved in the left-hand side of the inclusion defining delay validity, which consists of the consequences entailed by the entire stream S at time $\tau - d$.



(b) The figure represents the elements involved in the right-hand side of the inclusion defining delay validity, which consists of the consequences entailed by the restricted stream $S\restriction_{[0,\tau]}$ at time $\tau - d$. The red dots denote the part of the stream that is left out by the restriction.

Figure 3.1: Graphical representation of the definition of *delay validity*—see Definition 3.9. In both Figure 3.1a and 3.1b we have the timeline showing an arbitrary time point τ together with the time point $\tau - d$, which is marked by a green triangle to show that this is the time point for which we are interested in checking that the logical consequences are preserved. Black solid lines (other than the timeline) stand for streams—specified on their right side—and their extension stands for the time points that a stream may span.

Algorithm 1 does not need to wait indefinitely before it can determine with certainty that all entailed IDB facts have already been computed for a given time point.

In reading the definition of valid delay given next, it may help to compare the definition with Figure 3.1, which gives a graphical representation of the definition.

Definition 3.9. *A non-negative integer d is a* valid delay *for a program Π if, for each EDB stream S and each time point $\tau \geq d$, we have that $\Pi(S)\restriction_{\tau-d} \subseteq \Pi(S\restriction_{[0,\tau]})$.*

Note that, by definition, if d is a valid delay for Π, then so is each $d' > d$; hence, we will be interested in determining whether Π admits a valid delay and, if so, in computing the smallest valid delay.

We next define the notion of a valid window size w as a property of a program Π having valid delay d. Intuitively, the definition ensures three properties which can be exploited by an algorithm in order to compute a logical consequence α of $\Pi \cup S$ holding at time $\tau - d$. First, one can derive α without considering any fact (EDB or IDB) holding at a time point smaller than $\tau - w$, and hence such old facts can be safely "forgotten" by a procedure such as Algorithm 1. Second, the IDB facts in the interval $(\tau - d, \tau]$ entailed by $\Pi \cup S$ are not required in order to derive α, which implies that an algorithm only needs to keep in memory a full materialisation in the interval $[\tau - w, \tau - d]$. Third, α can be derived by restricting reasoning over the time interval $[\tau - w, \infty)$, which implies that an algorithm can reason over shifted data as done by Algorithm 1.

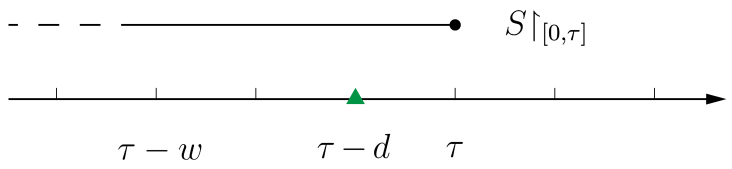
In reading the definition of valid window size given next, it may help to compare the definition with Figure 3.2, which gives a graphical representation of the definition.

Definition 3.10. *Let Π be a program and d a valid delay for Π. We say that $w \geq d$ is a* valid window size *for Π and d if, for each EDB stream S and each $\tau \geq d$, the following inclusion holds:*

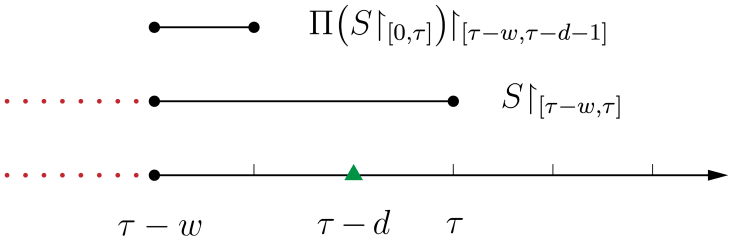
$$\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{\tau-d} \;\subseteq\; \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)$$

Note that, as in the case of delay, if w is a valid window size for Π and d, then so is each $w' > w$. Hence, in practice we will be interested in computing the minimum such w.

We next show that delay and window size validity as introduced in Definitions 3.9 and 3.10 characterise the correctness of Algorithm 1—that is, the algorithm will correctly output $\Pi(S)$ for each S if and only if d is a valid delay for Π and w is a valid window size for Π and d. Before establishing this result formally, we prove a lemma, which describes the contents M_τ of the memory at any point τ during the execution of the algorithm in terms of the logical consequences of $\Pi \cup S$.



(a) The figure represents the elements involved in the left-hand side of the inclusion defining window size validity, which consists of the consequences entailed by the restricted stream $S\restriction_{[0,\tau]}$ at time $\tau - d$.



(b) The figure represents the elements involved in the right-hand side of the inclusion defining window size validity, which consists of the consequences at time $\tau - d$ that are entailed by the restricted stream $S\restriction_{[\tau-w,\tau]}$ enriched with the IDB facts $\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]}$ over the timeline restricted so to start from $\tau - w$. The red dots denote the parts of the stream and of the timeline that are left out by the restrictions.

Figure 3.2: Graphical representation of the definition of *window size validity*—see Definition 3.10. In both Figure 3.2a and 3.2b we have the timeline showing an arbitrary time point τ together with the time points $\tau - w$ and $\tau - d$, where the latter is marked by a green triangle to show that this is the time point for which we are interested in checking that the logical consequences are preserved. Black solid lines (other than the timeline) stand for streams—specified on their right side—and their extension stands for the time points that a stream may span.

Lemma 3.11. *Consider the execution of Algorithm 1 parametrised with d and w on an input program Π and stream S. For τ a time point, let M_τ be the value of the memory variable M in the $(\tau+1)$-th iteration of the main loop of the algorithm right after executing Line 4. Then, the following inclusion holds:*

$$M_\tau \subseteq \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi\restriction_{[\tau-w-1,\infty)}\big(S\restriction_{[0,\tau-1]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)\restriction_{[\tau-w,\tau-d]} \cup S\restriction_{[\tau-w,\tau]}.$$

Proof. Note that, by definition of the algorithm, M_τ satisfies the following recursive equation for each $\tau \geq 0$, where $M_{-1} = \emptyset$ and $k = \max(0, \tau - w)$:

$$M_\tau = M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_\tau \cup \left(\Pi\Big(\big(M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_\tau\big) \ll k\Big)\restriction_{\tau-d-k} \gg k\right).$$

Then, by Theorem 3.6, we have:

$$M_\tau = M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_\tau \cup \Pi\restriction_{[\tau-w,\infty)}\big(M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_\tau\big)\restriction_{\tau-d}.^3 \tag{3.1}$$

Furthermore, we define $N_{\tau'} = \Pi\restriction_{[\tau'-w,\infty)}(S\restriction_{[0,\tau']})$ for each time point τ'—intuitively, the IDB facts entailed by the stream up to τ' if we consider time to start at $\tau'-w$—and rewrite the inclusion to show as

$$M_\tau \subseteq \Pi\restriction_{[\tau-w,\infty)}\Big(N_{\tau-1}\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)\restriction_{[\tau-w,\tau-d]} \cup S\restriction_{[\tau-w,\tau]}. \tag{3.2}$$

We show the claim of the lemma by induction on τ. In the base case, we have $\tau = 0$. Then, $M_0 = \Pi\restriction_{[\tau-w,\infty)}(S\restriction_0)\restriction_{\tau-d} \cup S\restriction_0$, which implies the claim since $S\restriction_0 = S\restriction_{[-w,0]}$.

In the inductive case, we have $\tau > 0$, and we assume that the claim holds for $\tau - 1$. It suffices to show that the r.h.s. of (3.1) is contained in the r.h.s. of the overall inclusion to show; we denote the latter r.h.s. as U in the following. Clearly, $S\restriction_\tau \subseteq U$, and thus it remains to be shown that $M_{\tau-1}\restriction_{[\tau-w,\infty)} \subseteq U$ and $\Pi\restriction_{[\tau-w,\infty)}\big(M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_\tau\big)\restriction_{\tau-d} \subseteq U$. For the former inclusion, note that the inductive hypothesis—in terms of Equation (3.2)—yields the following inclusion:

$$M_{\tau-1}\restriction_{[\tau-w,\infty)} \subseteq$$
$$\Pi\restriction_{[\tau-w-1,\infty)}\big(N_{\tau-2}\restriction_{[\tau-w-1,\tau-d-2]} \cup S\restriction_{[\tau-w-1,\tau-1]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau-1]}.$$

[3]Note that we make use of the fact that the intervals $[k,\infty)$ and $[\tau-w,\infty)$ are interchangeable when used as parameters of a restriction.

Furthermore, we have $\Pi\restriction_{[\tau-w-1,\infty)}\big(N_{\tau-2}\restriction_{[\tau-w-1,\tau-d-2]} \cup S\restriction_{[\tau-w-1,\tau-1]}\big) \subseteq N_{\tau-1}$, and hence the above implies

$$M_{\tau-1}\restriction_{[\tau-w,\infty)} \subseteq N_{\tau-1}\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau-1]} \subseteq U \tag{3.3}$$

as required. For the remaining inclusion, note that the first inclusion in (3.3) immediately yields

$$\Pi\restriction_{[\tau-w,\infty)}\big(M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_{\tau}\big)\restriction_{\tau-d} \subseteq \Pi\restriction_{[\tau-w,\infty)}\big(N_{\tau-1}\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\big)\restriction_{\tau-d} \subseteq U$$

which concludes the proof. □

With Lemma 3.11 at hand, we are ready to characterise correctness of Algorithm 1 in terms of delay and window size validity.

Theorem 3.12. *Algorithm 1 parametrised with d and w outputs $\Pi(S)$ on input program Π and on each input stream S if and only if d is a valid delay for Π and w is a valid window size for Π and d.*

Proof. ($\Leftarrow$) Assume that d is a valid delay for Π and that $w \geq d$ is a valid window size for Π and d, and let S be an EDB stream. We show that the algorithm outputs $\Pi(S)$. Let τ be a time point with $\tau \geq d$, and let M_τ be defined as in Lemma 3.11. Note that the algorithm outputs an IDB fact α with time argument $\tau - d$ if and only if $\alpha \in M_\tau$; thus, it suffices to show that $\Pi(S)\restriction_{\tau-d} \subseteq M_\tau$. Furthermore, since d is assumed to be a valid delay for Π, it suffices to show that

$$S\restriction_{[\tau-w,\tau]} \cup \Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d]} \subseteq M_\tau$$

We show this claim by induction on τ. In the base case, we have $\tau = 0$, and hence either $d = 0$ or $\tau - d < 0$. In the latter case, we have $\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d]} = \emptyset$, while in the former case, we have $\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d]} = \Pi(S\restriction_0)\restriction_0 \subseteq M_0$ by (3.1). Furthermore, in both cases we have $S\restriction_{[\tau-w,\tau]} = S\restriction_0 \subseteq M_0$ by (3.1). In the inductive case, we have $\tau > 0$ and we assume that the claim holds for $\tau - 1$. First, $S\restriction_{[\tau-w,\tau]} \subseteq M_\tau$ by (3.1) and the inductive hypothesis. Second, note that

$$\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \subseteq \Pi\big(S\restriction_{[0,\tau-1]}\big)\restriction_{[\tau-w,\tau-d-1]} \subseteq M_{\tau-1}\restriction_{[\tau-w,\infty)} \subseteq M_\tau$$

where the first inclusion holds since d is a valid delay, the second inclusion holds since $\Pi\big(S\restriction_{[0,\tau-1]}\big)\restriction_{[\tau-w-1,\tau-d-1]} \subseteq M_{\tau-1}$ by the inductive hypothesis, and the last inclusion holds by (3.1). Hence, it remains to show that

$$\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{\tau-d} \subseteq M_\tau.$$

To this end, note that, since w is a valid window size, we have

$$\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{\tau-d} \subseteq \Pi\restriction_{[\tau-w,\infty)}\big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\big)\restriction_{\tau-d} \tag{3.4}$$

where the r.h.s. can be rewritten as

$$\Pi\restriction_{[\tau-w,\infty)}\Big(\big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w-1,\tau-d-1]} \cup S\restriction_{[\tau-w-1,\tau-1]}\big)\restriction_{[\tau-w,\infty)} \cup S\restriction_{\tau}\Big)\restriction_{\tau-d}.$$

Furthermore, since d is a valid delay for Π, we have that $\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w-1,\tau-d-1]} = \Pi\big(S\restriction_{[0,\tau-1]}\big)\restriction_{[\tau-w-1,\tau-d-1]}$, and hence $\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w-1,\tau-d-1]} \subseteq M_{\tau-1}$ by the inductive hypothesis. Since also $S\restriction_{[\tau-w-1,\tau-1]} \subseteq M_{\tau-1}$ by the inductive hypothesis, then, by monotonicity of entailment, inclusion (3.4) implies

$$\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{\tau-d} \subseteq \Pi\restriction_{[\tau-w,\infty)}\big(M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_{\tau}\big)\restriction_{\tau-d}$$

and the claim follows since $\Pi\restriction_{[\tau-w,\infty)}\big(M_{\tau-1}\restriction_{[\tau-w,\infty)} \cup S\restriction_{\tau}\big)\restriction_{\tau-d} \subseteq M_\tau$ by (3.1).

($\Rightarrow$) We show that there is an input stream S on which Algorithm 1 does not output $\Pi(S)$ if either (i) d is not a valid delay for Π or (ii) d is a valid delay for Π but w is not a valid window size for Π and d. In either case it suffices to show an input stream S and a time point τ such that $\Pi(S)\restriction_{\tau-d} \not\subseteq M_\tau$, since we observed before that the former implies that the algorithm does not output $\Pi(S)$.

In the first case, let S be an input stream, let $\tau \geq d$ be a time point, and let α be a fact in $\Pi(S)\restriction_{\tau-d} \setminus \Pi\big(S\restriction_{[0,\tau]}\big)$—such S, τ, and α exist because d is not a valid delay for Π. We have $(M_\tau \setminus S\restriction_{[0,\tau]}) \subseteq \Pi\big(S\restriction_{[0,\tau]}\big)$ by Lemma 3.11, and hence $\alpha \notin M_\tau$, as required.

In the second case, let S be a stream, let $\tau \geq d$ be a time point, and let $\alpha \in \Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{\tau-d} \setminus \Pi\restriction_{[\tau-w,\infty)}\big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\big)$, which exist since w is not a valid window size for Π and d. By Lemma 3.11, $\alpha \notin M_\tau$, and the claim follows since $\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{\tau-d} \subseteq \Pi(S)\restriction_{\tau-d}$. □

Unfortunately there are programs for which a valid delay does not exist, and hence to which our algorithm is not applicable, as shown in the following example.

Example 3.13. *Consider the task of determining whether an observed potential cause is a likely cause of some effect observed later on. Specifically, a potential cause observed at time τ is a likely cause of some effect observed later on if an effect was observed at time τ' and a connector event was observed at each time point of the interval $(\tau, \tau']$. This is captured by the following rules.*

$$\mathit{Effect}(t) \rightarrow \mathit{ConnectedEffect}(t) \tag{3.5}$$

$$\mathit{ConnectedEffect}(t) \wedge \mathit{Connector}(t) \rightarrow \mathit{ConnectedEffect}(t-1) \tag{3.6}$$

$$\mathit{ConnectedEffect}(t) \wedge \mathit{PotentialCause}(t) \rightarrow \mathit{LikelyCause}(t) \tag{3.7}$$

Any $d \geq 0$ is not a valid delay for the rules above. In fact, given the stream

$$S = \{\mathit{PotentialCause}(0), \mathit{Effect}(d+1)\} \cup \{\mathit{Connector}(i) \mid 0 < i \leq d+1\}$$

we can derive $\mathit{LikelyCause}(0)$ from S but not from $S\restriction_{[0,d]}$.

For some programs the minimum valid delay is exponential in the size of the program, as shown in the following example.

Example 3.14. *For k an integer with $k > 0$, consider the program Π^k_{count} consisting of Rules (3.8)–(3.14). The program derives binary encodings of numbers with k bits, one per time point, starting from any time point where Start holds, and proceeding backwards in time. For i ranging over bit positions $[1, k]$, predicates Zero_i and One_i represent bit values, and predicates Flip_i and NoFlip_i allow the program to compute the successor of a number by marking the bits to flip and the bits to keep unchanged,*

respectively.

$$\mathit{Start}(t) \to \mathit{Zero}_i(t) \qquad \forall i \in [1,k] \tag{3.8}$$

$$\bigwedge_{\ell=1}^{i-1} \mathit{One}_\ell(t) \wedge \mathit{Zero}_i(t) \to \mathit{Flip}_j(t) \qquad \forall i \in [1,k],\ \forall j \in [1,i] \tag{3.9}$$

$$\bigwedge_{\ell=1}^{i-1} \mathit{One}_\ell(t) \wedge \mathit{Zero}_i(t) \to \mathit{NoFlip}_j(t) \qquad \forall i \in [1,k],\ \forall j \in (i,k] \tag{3.10}$$

$$\mathit{Zero}_i(t) \wedge \mathit{Flip}_i(t) \to \mathit{One}_i(t-1) \qquad \forall i \in [1,k] \tag{3.11}$$

$$\mathit{One}_i(t) \wedge \mathit{Flip}_i(t) \to \mathit{Zero}_i(t-1) \qquad \forall i \in [1,k] \tag{3.12}$$

$$\mathit{Zero}_i(t) \wedge \mathit{NoFlip}_i(t) \to \mathit{Zero}_i(t-1) \qquad \forall i \in [1,k] \tag{3.13}$$

$$\mathit{One}_i(t) \wedge \mathit{NoFlip}_i(t) \to \mathit{One}_i(t-1) \qquad \forall i \in [1,k] \tag{3.14}$$

Program Π^k_{count} *has minimum valid delay* $2^k - 1$*, since it derives facts for each time point in* $[\tau - 2^k + 1, \tau]$ *if* $\mathit{Start}(\tau)$ *is given in input, and it derives no fact otherwise. Given* $\mathit{Start}(\tau)$*, the program derives the binary encoding of zero at* τ *by Rule* (3.8)*. Then, if it has derived an integer* n *at time* τ'*, either* $n = 2^k - 1$ *and it stops deriving—there is no fact* $\mathit{Zero}_i(\tau')$ *and hence Rules* (3.9) *and* (3.10) *do not fire—or it derives the encoding of* $n + 1$ *at time* $\tau' - 1$ *by Rules* (3.9)–(3.14)*. As a result, the program derives the encoding of* $2^k - 1$ *at time* $\tau - 2^k + 1$*, and no fact at time* $\tau' < \tau - 2^k + 1$*.*

We will be interested in identifying programs that admit a valid delay and window size, and hence can be processed by Algorithm 1. For such programs, we will be interested in computing the minimum valid delay and window size, since they minimise the latency of the algorithm.

In the following Chapter 4 we look at syntactic conditions to compute valid delays and window sizes. These conditions do not always guarantee that the computed delays and window sizes are optimal, and hence in Chapter 5 we study the problem of computing minimum valid delays and window sizes; we determine its undecidability in the general case, and its complexity in a number of decidable cases.

Chapter 4

Syntactic Conditions for Delay and Window Size Validity

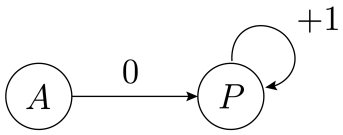
In this chapter we provide syntactic conditions on programs for delay and window size validity.

4.1 Syntactic Conditions for Delay Validity

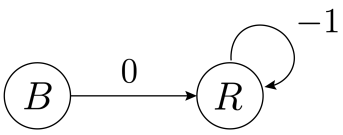
We define backward-bounded programs, which ensure existence of a valid delay by precluding the kind of temporal recursion towards past time points illustrated in Example 3.13. Backward-boundedness is defined over an edge-weighted dependency graph of the program, which has predicates as nodes and differences between time offsets as weights. The graph aims at capturing in a shallow way (but easy to check) how rules propagate facts over time.

Definition 4.1. *The* weighted dependency graph *of a program* Π *is an edge-weighted graph having a node for each predicate in* Π *and an edge* $\langle P, R, k - k' \rangle$ *whenever there is a rule* $r \in \Pi$ *of the form* $\varphi \wedge P(\mathbf{s}', t + k') \wedge \psi \rightarrow R(\mathbf{s}, t + k)$. *A program* Π *has* backward bound k *(for* $k \geq 0$*) if, for each path in the weighted dependency graph of* Π *that starts in an EDB predicate, the sum of the edge weights is at least* $-k$*;* Π *is* backward-bounded *if it has a backward bound.*

In the following example we give two simple programs, discuss their respective weighed dependency graphs, and then argue that the former has a backward bound and the latter does not.



(a) Weighted dependency graph of Π_{AP}.



(b) Weighted dependency graph of Π_{BR}.

Figure 4.1: Weighted dependency graphs of the programs in Example 4.2.

Example 4.2. *Consider the program Π_{AP} consisting of the following two rules.*

$$A(t) \to P(t) \tag{4.1}$$

$$P(t) \to P(t+1) \tag{4.2}$$

The weighted dependency graph of Π_{AP} is shown in Figure 4.1a. It has a node for A and one for P, an edge from A to P with weight zero because of Rule (4.1)*, and an edge from P to itself with weight one because of Rule* (4.2)*. All the weights in the graph are non-negative, and hence we can conclude that zero is a backward bound of Π_{AP}—specifically, its* least backward bound*. Consider now the program Π_{BR} consisting of the following two rules.*

$$B(t) \to R(t) \tag{4.3}$$

$$R(t) \to R(t-1) \tag{4.4}$$

The graph of Π_{BR}, shown in Figure 4.1b, is built similarly to the previous one, except that the edge from R to itself has weight -1, due to Rule (4.4)*. Because of that edge, the graph has paths starting in an EDB predicate—namely B—where the sum of the weights is arbitrarily small, and hence Π_{BR} has no backward bound.*

The programs in our motivating Examples 1.1–1.3 are backward-bounded, with the program in Example 1.2 having backward bound zero as it is forward-propagating. Note that checking whether a given program is backward-bounded, and if so computing the least backward bound, is feasible in polynomial time using standard graph algorithms. We next show that the backward bound of a program Π is guaranteed to be a valid delay for Π.

Theorem 4.3. *Let* Π *be a program with backward bound* k*. Each integer* $d \geq k$ *is a valid delay for* Π*.*

Proof. Assume that $d \geq k$ is not a valid delay for Π. We show that k is not a backward bound for Π. By the definition of valid delay, there is a stream S, a time point $\tau \geq d$, and a fact β with time argument $\tau - d$ such that $\beta \in \Pi(S) \setminus \Pi(S\restriction_{[0,\tau]})$. As a result, there exists a fact α with time argument $\tau' > \tau$ such that $\beta \notin \Pi(S \setminus \{\alpha\})$. By a simple induction on the height of a derivation of β from $\Pi \cup S$ we can show existence of a path from the predicate P_α of α to the predicate P_β of β in the weighted dependency graph of Π having weight $\tau - d - \tau'$. Note that $\tau - d - \tau' < \tau' - d - \tau' = -d \leq -k$, where the first inequality holds since $\tau' > \tau$ and the second one since $d \geq k$. Thus, the weighted dependency graph of Π has a path of weight strictly smaller than $-k$, and hence k is not a backward bound for Π. □

It follows from Theorem 4.3 that delay existence, as given in Definition 5.1, is trivial for backward-bounded programs. Delay and window size validity checking, however, remain important for these programs since the valid delay established by the theorem may not be minimal. Furthermore, there are simple programs that obviously admit a valid delay, but for which the backward-boundedness condition falls short.

Example 4.4. *The program consisting of rules* $A(t) \rightarrow B(t)$ *and* $A(t) \wedge B(t) \rightarrow B(t-1)$ *is not backward-bounded; however,* $d = 1$ *is a valid delay. In fact, the second rule can be simplified as* $A(t) \rightarrow B(t-1)$*, yielding a program with backward bound one.*

We will show in the next chapter that the delay existence and validity problems are undecidable in general, and the latter is so even for backward-bounded programs.

4.2 Syntactic Conditions for Window Size Validity

We can show that if a program Π admits a valid delay d, then it must admit a valid window size w as well. Furthermore, the window size is bounded linearly in d and

the forward radius of Π.

Theorem 4.5. *Let Π be a program, let d be a valid delay for Π, and let ρ be the maximum forward radius of a rule in Π. Then, $w = d + \rho$ is a valid window size for Π and d.*

Proof. Let S be an EDB stream, let $\tau \geq d$ a time point, and let $w = d + \rho$. We show the inclusion

$$\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-d,\infty)} \subseteq \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi(S\restriction_{[0,\tau]})\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)$$

by induction on the height of a shortest derivation δ of a fact $\alpha \in \Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-d,\infty)}$; the main claim is then immediate. Let α and δ be as required, and let τ' be the time argument of α. In the base case, we have $\alpha \in S\restriction_{[0,\tau]}$, and hence $\alpha \in S\restriction_{[\tau-w,\tau]}$ as $\tau' \geq \tau - d \geq \tau - w$, and the claim follows. In the inductive case, let r be the rule labelling the root of δ and let β be an arbitrary atom in the body of r with time argument τ''. Note that every time point in r is at least $\tau' - \rho \geq \tau - d - \rho = \tau - w$ since each rule in Π is temporally-guarded; thus, every time argument of r is at least $\tau - w$, and in particular $\tau'' \geq \tau - w$. They imply that it suffices to show

$$\beta \in \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi(S\restriction_{[0,\tau]})\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)$$

in the following two cases: $\tau'' \in [\tau - w, \tau - d - 1]$ and $\tau'' \geq \tau - d$. In the former case, the claim follows directly from $\beta \in \Pi(S\restriction_{[0,\tau]})$; and in the latter case, the claim follows from $\beta \in \Pi(S\restriction_{[0,\tau]})$ by the inductive hypothesis as β has a derivation that is strictly shorter than δ. □

The idea of a window size that is based on the maximum forward radius of a program can be found in [25] for the case of a forward-propagating language where the maximum forward radius is one. Theorem 4.5 generalises that idea to programs with arbitrary forward radius and, most importantly, with the power of propagating information backwards in time. This requires the notion of delay, which needs to be taken into account explicitly in the syntactic condition for window size validity.

It is important to notice that the valid window size provided by Theorem 4.5 may not be the minimal one.

Example 4.6. *Consider the program consisting of rules*

$$A(t) \to P(t+1) \quad \text{and} \quad P(t) \wedge B(t+1) \to Q(t).$$

The program admits a minimum valid delay of one, and its maximum forward radius is one as well. As a result, $w = 2$ is a valid window size by Theorem 4.5. We can show, however, that the minimum window size of this program is one.

We will show in the next chapter that window validity is, in fact, undecidable in general and computationally hard even in restricted cases.

4.2.1 The Case of Forward-Propagating Programs

We show that, in the case of forward-propgating programs, the sufficient condition for window size validity established in Theorem 4.5 is also necessary up to rule redundancy.

Theorem 4.7. *For Π a forward-propagating program, a non-negative integer w is a valid window size for Π (and delay zero) if and only if $\Pi \sqsubseteq \Pi^w$ with Π^w the set of rules in Π having forward radius at most w.*

Proof. Let S be an EDB stream, τ be a time point, $N = \Pi(S\restriction_{[0,\tau]})$, $N^w = \Pi^w(S\restriction_{[0,\tau]})$, and $T = [\tau - w, \infty)$. To see that $\Pi \sqsubseteq \Pi^w$ implies that w is a valid window size for Π, note that:

$$N\restriction_\tau \subseteq N^w\restriction_\tau \subseteq \Pi^w\restriction_T(N^w\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]}) \subseteq \Pi\restriction_T(N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]})$$

where the first inclusion holds by $\Pi \sqsubseteq \Pi^w$, the second one by Theorem 4.5, and the third one by monotonicity. To see that $\Pi \sqsubseteq \Pi^w$ holds whenever w is a valid window size for Π, note that, for $T' = [\tau - w, \tau]$ and $F = N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]}$, we have:

$$N\restriction_\tau \subseteq \Pi\restriction_T(F) \subseteq \Pi\restriction_{T'}(F) \subseteq \Pi^w\restriction_{T'}(F) \subseteq N^w$$

where the first inclusion holds because w is a valid window size for Π, the second one holds because Π is forward-propagating, the third one since $\Pi\restriction_{T'} = \Pi^w\restriction_{T'}$, and the last one by monotonicity. □

The previous theorem allows one to easily reduce window size validity to program containment. However, in Section 5.3.3 we will show a reduction that applies to more general programs and still allows one to obtain tight complexity bounds for the restricted case of forward-propagating programs.

As a consequence of the previous theorem, if a program contains no redundant rule—i.e., $\Pi \not\sqsubseteq \Pi \setminus \{r\}$ for each $r \in \Pi$—the radius-based condition is also necessary.

Corollary 4.8. *If* Π *is a forward-propagating program that contains no redundant rule, then the maximum forward radius of a rule in* Π *is the minimum valid window size for* Π *(and delay zero).*

Chapter 5

The Delay and Window Size Problems

In this chapter we study the computation and decision problems relative to delay and window size validity. We define the problems in Section 5.1, we show undecidability of the problems in Section 5.2, we establish complexity upper bounds in Section 5.3, and lower bounds in Section 5.4.

5.1 Definition of the Problems

The computational task of interest in our setting is to first check whether a program Π admits a valid delay and, if it does, to subsequently compute the corresponding minimal valid delay d for Π and the minimal window size w for Π and d. Reducing delays and window sizes is important in practice, as it ensures that a correct algorithm will keep to a minimum the number of facts stored in memory at any point in time, and will minimise latency by returning each output fact as early as possible. In this chapter, we will focus on the decision problems relevant to our task at hand, which we define next.

Definition 5.1. *We define the following decision problems, where the numbers in the input program are encoded in unary and the other input numbers are encoded in binary.*

- Delay existence*: Given a program* Π*, decide whether a valid delay for* Π *exists.*

- Delay validity*: Given a program* Π *and a non-negative integer* d*, decide whether* d *is a valid delay for* Π*.*
- Window size validity*: Given a program* Π*, a valid delay* d *for* Π*, and* $w \geq d$*, decide whether* w *is a valid window size for* Π *and* d*.*

5.2 Undecidability Results

In this section we explore the limitations of our approach and establish undecidability results for all the reasoning problems considered in this thesis. Our undecidability proofs are obtained by reduction from *containment of forward-propagating/backward-bounded programs*, which is undecidable since forward-propagating programs capture (non-temporal) Datalog—as argued in the preliminaries—and containment of Datalog programs is undecidable [33, 46]. We start by establishing undecidability of delay existence. As discussed in Section 5.1, a program may not admit a valid delay, e.g., if the program can recursively propagate derived facts arbitrarily far towards past time points.

Theorem 5.2. *Delay existence is undecidable.*

Proof. We provide a reduction from containment of forward-propagating programs, which maps a pair of forward-propagating programs $\langle \Pi_1, \Pi_2 \rangle$ to a program Π such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if Π admits a valid delay. We construct Π as follows. For each predicate P, let P^1, P^2, A_P and B_P be fresh predicates of the same arity as (and uniquely associated to) P. Let Π_1' (respectively Π_2') be the program obtained by replacing each IDB predicate P in Π_1 (respectively in Π_2) with P^1 (respectively with P^2). Then, Π is the program obtained by extending $\Pi_1' \cup \Pi_2'$ with the following rules, where A and B are fresh unary (EDB) predicates:

- Rules (5.1)–(5.4) for each IDB predicate P of Π_1:

$$P^1(\mathbf{x}, t) \wedge A(t) \rightarrow A_P(\mathbf{x}, t) \tag{5.1}$$

$$A_P(\mathbf{x}, t) \rightarrow A_P(\mathbf{x}, t+1) \tag{5.2}$$

$$A_P(\mathbf{x}, t) \wedge B(t) \rightarrow B_P(\mathbf{x}, t) \tag{5.3}$$

$$B_P(\mathbf{x}, t+1) \wedge A_P(\mathbf{x}, t) \rightarrow B_P(\mathbf{x}, t). \tag{5.4}$$

- Rules (5.5) and (5.6) for each IDB predicate P occurring in Π_2:

$$P^2(\mathbf{x}, t) \wedge A(t) \rightarrow B_P(\mathbf{x}, t) \tag{5.5}$$

$$B_P(\mathbf{x}, t) \rightarrow B_P(\mathbf{x}, t+1). \tag{5.6}$$

Let us assume first that $\Pi_1 \sqsubseteq \Pi_2$. We argue that zero is a valid delay for Π. For this, observe that $\Pi_1 \sqsubseteq \Pi_2$ implies that, for each set of EDB facts and each IDB P, the extension of P^1 will be contained in that of P^2 by our construction. Thus, Rules (5.3) and (5.4) become subsumed by (5.5) and (5.6) and hence can be removed from Π without altering fact entailment; in doing so, we can observe that Π would become forward-propagating since so are Π_1 and Π_2, and hence it would admit zero as a valid delay by Theorem 4.3.

Assume now that $\Pi_1 \not\sqsubseteq \Pi_2$ and fix any d; we argue that d cannot be a valid delay for Π. By our assumption, there must be a set D of EDB facts and facts $P^1(\mathbf{o}, \tau)$ and $P^2(\mathbf{o}, \tau)$ such that $\Pi_1' \cup D \models P^1(\mathbf{o}, \tau)$ but $\Pi_2' \cup D \not\models P^2(\mathbf{o}, \tau)$. Let D' be the set extending D with facts $A(\tau)$ and $B(\tau + d + 1)$. But then, we have that $B_P(\mathbf{o}, \tau)$ follows from $\Pi \cup D'$ but $B_P(\mathbf{o}, \tau)$ does not follow from $\Pi \cup D'\restriction_{[0,\tau+d]}$, because $B(\tau + d + 1) \notin D'\restriction_{[0,\tau+d]}$. Therefore, d cannot be a valid delay. □

We next show undecidability of the delay validity problem. Our undecidability result holds even for backward-bounded programs, which, by Theorem 4.3, always admit a valid delay of linear size.

Theorem 5.3. *Delay validity is undecidable, even if restricted to backward-bounded programs.*

Proof. We provide a reduction from containment of backward-bounded programs. The reduction maps a pair of backward-bounded programs $\langle \Pi_1, \Pi_2 \rangle$ to a program Π and a number k such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if k is a valid delay for Π.

We assign k to the maximum between the least backward bounds of Π_1 and Π_2. To define Π, we proceed by defining Π'_1 and Π'_2 as in the proof of Theorem 5.2 and by introducing also a fresh unary (EDB) predicate A. Then, Π extends $\Pi'_1 \cup \Pi'_2$ with two rules of the form (5.7) and (5.8), respectively, for each IDB predicate P in $\Pi_1 \cup \Pi_2$:

$$P^1(\mathbf{x}, t) \wedge A(t + k + 1) \rightarrow A_P(\mathbf{x}, t) \tag{5.7}$$

$$P^2(\mathbf{x}, t) \rightarrow A_P(\mathbf{x}, t). \tag{5.8}$$

Clearly, Π is backward-bounded since so are Π_1 and Π_2 and predicates A_P is fresh.

We next argue correctness of the reduction. Let us assume $\Pi_1 \sqsubseteq \Pi_2$. We argue that k is a valid delay for Π. For this, observe that $\Pi_1 \sqsubseteq \Pi_2$ implies that, for each finite set of EDB facts (and hence for each infinite one) and each IDB predicate P, the extension of P^1 is contained in the extension of P^2 by our construction. As a result, Rule (5.7) becomes subsumed by Rule (5.8) and hence can be removed from Π without altering fact entailment; in doing so, we can observe that Π would admit k as a backward bound, and hence as a valid delay by Theorem 4.3.

Assume now that k is a valid delay for Π. Let D be a finite set of EDB facts, let α be an IDB fact of the form $P(\mathbf{o}, \tau)$, and assume that $\Pi_1 \cup D \models \alpha$. We argue that $\Pi_2 \cup D \models \alpha$, which implies that $\Pi_1 \sqsubseteq \Pi_2$. Since $\Pi_1 \cup D \models \alpha$ we have that $\Pi'_1 \cup D \models P^1(\mathbf{o}, \tau)$. Let $D' = D \cup \{A(\tau + k + 1)\}$; clearly, $\Pi \cup D' \models A_P(\mathbf{o}, \tau)$ since Rule (5.7) becomes applicable. It follows that $\Pi \cup D'\restriction_{[\tau+k]} \models A_P(\mathbf{o}, \tau)$ since k is a valid delay for Π. It follows that $\Pi \cup D'\restriction_{[\tau+k]} \models P^2(\mathbf{o}, \tau)$ since A_P occurs only in Rules (5.7) and (5.8), and Rule (5.7) requires $A(\tau + k + 1)$ to derive $A_P(\mathbf{o}, \tau)$, but $\Pi \cup D'\restriction_{[\tau+k]} \not\models A(\tau + k + 1)$. It follows that $\Pi_2 \cup D'\restriction_{[\tau+k]} \models P(\mathbf{o}, \tau)$, and hence $\Pi_2 \cup D\restriction_{[\tau+k]} \models P(\mathbf{o}, \tau)$ as required, since $D' \setminus D = \{A(\tau + k + 1)\}$ and predicate A does not occur in Π_2. □

We conclude by showing undecidability of window size validity checking. Our undecidability result applies even to forward-propagating programs, which admit zero as a valid delay (see Theorem 4.3) and a linear valid window size (see Theorem 4.5).

Theorem 5.4. *Window size validity is undecidable, even if restricted to forward-propagating programs.*

Proof. We provide a reduction from containment of forward-propagating programs, which is undecidable. The reduction maps a pair of forward-propagation programs $\langle \Pi_1, \Pi_2 \rangle$ to a program Π with valid delay zero and a number w such that $\Pi_1 \sqsubseteq \Pi_2$ if and only if w is a valid window size for program Π and delay zero.

In our reduction, we define w as the maximum forward radius of a rule in $\Pi_1 \cup \Pi_2$. To define Π, we proceed by constructing Π_1' and Π_2' as in the proofs of Theorems 5.2 and 5.3, and by introducing also a fresh unary (EDB) predicate A. Then, Π extends $\Pi_1' \cup \Pi_2'$ with two rules of the form (5.9) and (5.10), respectively, for each IDB predicate P occurring in $\Pi_1 \cup \Pi_2$:

$$P^1(\mathbf{x}, t) \wedge A(t - w - 1) \rightarrow A_P(\mathbf{x}, t) \tag{5.9}$$

$$P^2(\mathbf{x}, t) \rightarrow A_P(\mathbf{x}, t). \tag{5.10}$$

Clearly, Π is forward-propagating if so are Π_1 and Π_2 and hence admits zero as a valid delay.

We next argue correctness of the reduction. Assume that $\Pi_1 \sqsubseteq \Pi_2$, let S be an EDB stream for Π_1, let α be a fact with time argument τ, and assume that $\alpha \in \Pi(S\restriction_{[0,\tau]})$. We show that $\alpha \in \Pi\restriction_T(N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]})$ for $N = \Pi(S\restriction_{[0,\tau]})$ and $T = [\tau - w, \infty)$; it is sufficient to establish that w is a valid window size for program Π and delay zero. The claim clearly holds if α is a fact of the form $P^i(\mathbf{o}, \tau)$, as a direct consequence of Theorem 4.5 and our definition of w. So, assume that α is of the form $A_P(\mathbf{o}, \tau)$. Since A_P occurs in Π only in the head of Rules (5.9) and (5.10), we have that either $P^2(\mathbf{o}, \tau) \in \Pi(S\restriction_{[0,\tau]})$ or $\{P^1(\mathbf{o}, \tau), A(\tau - w - 1)\} \subseteq \Pi(S\restriction_{[0,\tau]})$. In the former case, we have $P^2(\mathbf{o}, \tau) \in \Pi\restriction_T(N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]})$ again as a consequence of Theorem 4.5 and hence $\alpha \in \Pi\restriction_T(N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]})$ by Rule (5.10), as required.

In the latter case, we have $P^1(\mathbf{o}, \tau) \in \Pi(S\restriction_{[0,\tau]})$, which implies $P(\mathbf{o}, \tau) \in \Pi_1(S\restriction_{[0,\tau]})$. Since $\Pi_1 \sqsubseteq \Pi_2$ by assumption, we then have $P(\mathbf{o}, \tau) \in \Pi_2(S\restriction_{[0,\tau]})$, and $P^2(\mathbf{o}, \tau) \in \Pi\restriction_T(N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]})$ by Theorem 4.5; thus, $\alpha \in \Pi\restriction_T(N\restriction_{[\tau-w,\tau-1]} \cup S\restriction_{[\tau-w,\tau]})$ by Rule (5.10), as required.

Assume now that w is a valid window size for program Π and delay zero. Let D be a finite set of EDB facts, let α be an IDB fact of the form $P(\mathbf{o}, \tau)$, and assume that $\alpha \in \Pi_1(D)$. We argue that $\alpha \in \Pi_2(D)$, which suffices to show that $\Pi_1 \sqsubseteq \Pi_2$. Since $\alpha \in \Pi_1(D)$, we have that $P^1(\mathbf{o}, \tau) \in \Pi(D)$, and hence $A_P(\mathbf{o}, \tau) \in \Pi(D)$ by Rule (5.10). For $N = \Pi(D\restriction_{[0,\tau]})$, it follows that $A_P(\mathbf{o}, \tau) \in \Pi(N\restriction_{[\tau-w,\tau-1]} \cup D\restriction_{[\tau-w,\tau]})$ since w is a valid window size for Π and zero. Hence, $P^2(\mathbf{o}, \tau) \in \Pi(N\restriction_{[\tau-w,\tau-1]} \cup D\restriction_{[\tau-w,\tau]})$ since A_P occurs in Π only in Rules (5.9) and (5.10) and $A(\tau-w-1) \notin \Pi(N\restriction_{[\tau-w,\tau-1]} \cup D\restriction_{[\tau-w,\tau]})$. As a result, we have $P^2(\mathbf{o}, \tau) \in \Pi(D)$ by the definition of N and the monotonicity and transitivity of entailment. We then have $P(\mathbf{o}, \tau) \in \Pi_2(D)$, as required. □

5.3 Decidability and Complexity Upper Bounds

In this section, we show that decidability of all our reasoning problems can be regained by making rather mild assumptions on the input streams. In particular, we consider the situation where the set of domain objects that can occur in a stream is fixed in advance. This is a reasonable assumption in many applications; for instance, in Examples 1.1 and 1.2, it amounts to assuming that the nodes in the network and the sensors generating temperature readings, respectively, remain the same throughout a given query session. From a technical point of view, fixing the object domain allows us to ground the object variables in a program to a set of known objects; such grounding is exponential in general (and polynomial if the maximum arity of a predicate is considered fixed) and results in an object-ground program, where all variables are time variables. In turn, an object-ground program Π can be equivalently restated as an *object-free* program—i.e., a program with no object terms—by replacing each atom $A(\mathbf{o}, s)$ in Π with the atom $A_{\mathbf{o}}(s)$, for $A_{\mathbf{o}}$ a fresh predicate uniquely associated with A and $\mathbf{o}$.

Thus, we first obtain upper bounds for object-free programs, and then immediately derive upper bounds for unrestricted programs via grounding. We proceed as follows.

- In Section 5.3.1, we characterise delay existence, delay validity and program containment for object-free programs in terms of certain languages over finite words.
- In Section 5.3.2, we show that the aforementioned languages can be recognised by finite automata, and show that these automata can be constructed in polynomial space in the size of the input program.
- In Section 5.3.3, we use the automata to establish PSpace upper bounds for delay existence, delay validity, and program containment restricted to object-free programs; a PSpace upper bound for window size validity is then obtained by polynomially reducing the problem to containment. Finally, the fixed object domain assumption allows us to exploit grounding in order to lift all of these results to unrestricted programs and bounded-arity programs, and obtain corresponding ExpSpace and PSpace upper bounds, respectively.

5.3.1 Language-Theoretic Characterisations

In this section, we describe word languages that can be used to characterise delay existence, delay validity, and program containment. The languages capture the respective problems for a fixed object-free program Π, and take an additional parameter $\mathcal{I}$ that is a subset of the IDB predicates occurring in Π. The parameter $\mathcal{I}$ helps us to deal with a normalisation step that we perform. Our normalisation yields a conservative extension of the program Π—i.e., a program Π' which is equivalent to Π with respect to the predicates in Π but also contains additional predicates. Such predicates are problematic, and the parameter $\mathcal{I}$ allows us to focus on the original set of predicates, so to preserve the languages despite of normalisation. For instance, it might be the case that Π admits a valid delay but the normalised program Π' does not admit a valid delay ; we will show that, however, program Π' admits a valid delay if we 'focus' on the set $\mathcal{I}_\Pi$ of IDB predicates coming from Π.

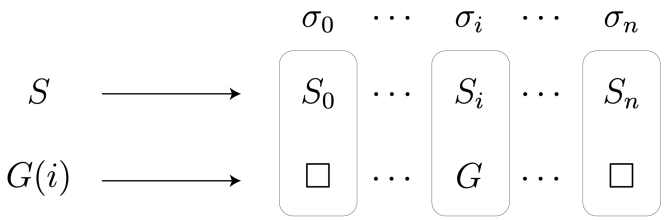


Figure 5.1: A finite set S of facts and a fact $G(i)$ are encoded into the word $\sigma_0 \dots \sigma_n$ where n is the maximum between i and the largest time point occurring in S. Each S_j is the set of predicates occurring in $S\restriction_j$, and $\square$ is a blank symbol.

All the languages introduced here share the same alphabet, which is parametrised by the object-free program of interest Π and a subset $\mathcal{I}$ of the IDB in Π. The alphabet is constructed so that each word represents an instance $\langle S, \alpha \rangle$ of the fact entailment problem with respect to a fixed program Π, where S is a finite input set of EDB facts—thought of as a stream prefix—and α is a fact over a predicate in $\mathcal{I}$. The encoding is explained in Figure 5.1. The key property of such encoding is that the time points in S and the time argument of α are encoded away into word positions.

The language $\mathsf{implies}(\Pi, \mathcal{I})$ will contain all words representing a stream prefix S and fact α over a predicate in $\mathcal{I}$ such that $\Pi \cup S \models \alpha$; in this way, the language captures the logical $\mathcal{I}$-consequences of Π when applied to an arbitrary stream. Dually, the language $\mathsf{notimplies}(\Pi, \mathcal{I})$ will contain all words representing S and α such that $\Pi \cup S \not\models \alpha$. Note that the language $\mathsf{implies}(\Pi, \mathcal{I}) \cup \mathsf{notimplies}(\Pi, \mathcal{I})$ does not contain all words over the considered alphabet, since it contains only well-formed encodings of instances of the entailment problem.

Finally, the language $\mathsf{delay}(\Pi, \mathcal{I})$ is defined in terms of the languages $\mathsf{implies}(\Pi, \mathcal{I})$ and $\mathsf{notimplies}(\Pi, \mathcal{I})$. The key property of this language is that it contains a word of length d if and only if d is not a valid delay for Π (with respect to the predicates in $\mathcal{I}$). The intuition why this property holds is given in Figure 5.1.

The definition of the alphabet and languages is given next, where we also define a handy notation for words representing instances of the entailment problem; such notation factors out the encoding of the 'answer' fact by making use of an explicit

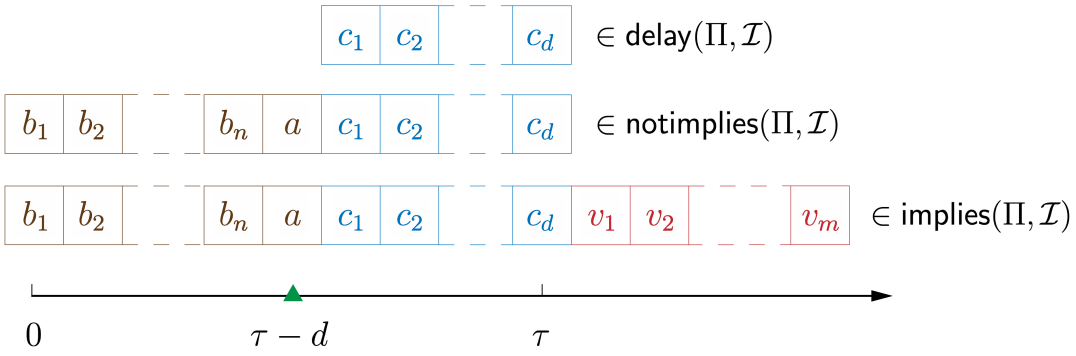


Figure 5.2: A word $w_1 = c_1 \dots c_d$ is in $\mathsf{delay}(\Pi, \mathcal{I})$ if it can be extended to a word $w_2 = b_1 \dots b_n a w_1$ of $\mathsf{notimplies}(\Pi, \mathcal{I})$ for a encoding an 'answer' fact, and in addition w_2 can be extended to a word $w_3 = w_2 v_1 \dots v_m$ of $\mathsf{implies}(\Pi, \mathcal{I})$. This means that w_3 encodes a stream S and a fact α holding at $\tau - d$ such that $\Pi \cup S \models \alpha$, and w_2 encodes that $\Pi \cup S\restriction_{[0,\tau]} \not\models \alpha$. Thus, the length d of w_1 is not a valid delay for Π, and hence words in delay witness the non-validity of delays via their length. It may help to compare this figure with Figure 3.1.

time point.

Definition 5.5. *Let Π be an object-free program, let $\mathcal{E}$ be the set of all EDB predicates in Π, and let $\mathcal{I}$ be a set of IDB predicates in Π. We write $\Sigma(\Pi, \mathcal{I})$ for the set $2^{\mathcal{E}} \times (\mathcal{I} \cup \{\Box\})$. Elements of $\Sigma(\Pi, \mathcal{I})$ are called* letters. *A letter $\langle E, b\rangle$ is an* answer letter *if $b \neq \Box$. For $E_0, \dots, E_n \in 2^{\mathcal{E}}$, $i \in [0, n]$, and $P \in \mathcal{I}$, we write $\langle i, P, E_0, \dots, E_n\rangle$ for the word $\langle E_0, b_0\rangle \dots \langle E_n, b_n\rangle$ over $\Sigma(\Pi, \mathcal{I})$ where $b_i = P$ and $b_j = \Box$ for each $j \neq i$.*

We define the following languages over the aforementioned alphabet:

- $\mathsf{implies}(\Pi, \mathcal{I})$ *consists of each word $\langle i, P, E_0, \dots, E_n\rangle$ such that $\Pi \cup S \models P(i)$, where $S = \{A(j) \mid A \in E_j,\ 0 \leq j \leq n\}$.*
- $\mathsf{notimplies}(\Pi, \mathcal{I})$ *consists of each word $\langle i, P, E_0, \dots, E_n\rangle$ such that $\Pi \cup S \not\models P(i)$, where $S = \{A(j) \mid A \in E_j,\ 0 \leq j \leq n\}$.*
- $\mathsf{delay}(\Pi, \mathcal{I})$ *consists of each word u_1 for which there exists a word u_2 ending with an answer letter such that $u_2 u_1$ is in $\mathsf{prefimplies}(\Pi, \mathcal{I}) \cap \mathsf{notimplies}(\Pi, \mathcal{I})$, for $\mathsf{prefimplies}(\Pi, \mathcal{I})$ the language consisting of each prefix of a word in $\mathsf{implies}(\Pi, \mathcal{I})$.*

The following theorem characterises program containment, delay validity, and delay existence in terms of the languages in Definition 5.5. It is immediate that program containment can be characterised in terms of entailment and non-entailment. The characterisation of the delay problems follows directly from the key property of the delay language that we have discussed above.

Theorem 5.6. *Let* Π *and* Π' *be object-free programs with the same set of EDB predicates, and let* $\mathcal{I}$ *be the set of IDB predicates in* Π*. Then, the following statements hold:*

1. $\Pi \sqsubseteq \Pi'$ *if and only if* $\mathsf{implies}(\Pi, \mathcal{I}) \cap \mathsf{notimplies}(\Pi', \mathcal{I}) = \emptyset$.
2. *A non-negative integer* d *is a valid delay for* Π *if and only if each word in* $\mathsf{delay}(\Pi, \mathcal{I})$ *is of length at most* $d - 1$.
3. Π *has a valid delay if and only if* $\mathsf{delay}(\Pi, \mathcal{I})$ *is a finite language.*

Proof. We show each of the statements in the theorem.

Statement 1. Let $\Pi \sqsubseteq \Pi'$ and let $u \in \mathsf{implies}(\Pi, \mathcal{I})$ be of the form $\langle i, P, E_0, \ldots, E_n \rangle$. By the definition of $\mathsf{implies}(\Pi, \mathcal{I})$, we have $\Pi \cup D \models P(i)$ for $D = \{A(j) \mid A \in E_j,\ 0 \leq j \leq n\}$. Since $\Pi \sqsubseteq \Pi'$, we have $\Pi' \cup D \models P(i)$, and hence $u \notin \mathsf{notimplies}(\Pi', \mathcal{I})$.

For the converse direction, assume $\mathsf{implies}(\Pi, \mathcal{I}) \cap \mathsf{notimplies}(\Pi', \mathcal{I}) = \emptyset$ and let $\Pi \cup D \models P(i)$ for i a time point, D a finite set of EDB predicates and P an IDB predicate. Let n be the largest time point in D and let E_j for $0 \leq j \leq n$ be the subset of facts in D with time argument j; then, we can write D as $\{A(j) \mid A \in E_j,\ 0 \leq j \leq n\}$. Consider the word $u = \langle i, P, E_0, \ldots, E_n \rangle$, which by construction belongs to $\mathsf{implies}(\Pi, \mathcal{I})$. By our assumption, we have that $u \notin \mathsf{notimplies}(\Pi', \mathcal{I})$, and hence $\Pi' \cup D \models P(i)$ by the definition of $\mathsf{notimplies}(\Pi', \mathcal{I})$, as required.

Statement 2. Assume that $u_1 \in \mathsf{delay}(\Pi, \mathcal{I})$ and $|u_1| \geq d$; we show that d is not a valid delay for Π. Since $u_1 \in \mathsf{delay}(\Pi, \mathcal{I})$, there exists u_2 ending with an answer letter and satisfying $u_2 u_1 \in \mathsf{prefimplies}(\Pi, \mathcal{I}) \cap \mathsf{notimplies}(\Pi, \mathcal{I})$. Since $u_2 u_1 \in \mathsf{notimplies}(\Pi, \mathcal{I})$, there are sets $E_1, \ldots, E_n$ EDB of predicates, an integer $p \in [1, n]$,

and an IDB predicate P such that $u_2u_1 = \langle i, P, E_0, \ldots, E_n\rangle$. Since u_2 ends with an answer letter and u_2u_1 has exactly one answer letter, we have that u_1 is of the form $\langle E_{i+1}, \square\rangle, \ldots, \langle E_n, \square\rangle$. Furthermore, since $u_2u_1 \in \mathsf{prefimplies}(\Pi, \mathcal{I})$, there must exist u_3 such that $u_2u_1u_3 \in \mathsf{implies}(\Pi, \mathcal{I})$; furthermore, u_3 will be of the form $\langle E_{n+1}, \square\rangle, \ldots, \langle E_m, \square\rangle$ for some $m \geq n$. Let $S = \{A(j) \mid A \in E_j,\ 0 \leq j \leq m\}$. It follows that $\Pi \cup S \models P(i)$ since $u_2u_1u_3 \in \mathsf{implies}(\Pi, \mathcal{I})$, and $\Pi \cup S\restriction_{[0,n]} \not\models P(i)$ since $u_2u_1 \in \mathsf{notimplies}(\Pi, \mathcal{I})$. Since $S\restriction_{[0,i+d]} \subseteq S\restriction_{[0,n]}$ (note that $n - i \geq d$ is the length of u_1), it follows that $\Pi \cup S\restriction_{[0,i+d]} \not\models P(i)$ by the monotonicity of first-order logic, and hence d is not a valid delay for Π.

For the other direction, assume that d is not a valid delay for Π; we show that $\mathsf{delay}(\Pi, \mathcal{I})$ contains a word of length at least d. By the definition of valid delay, there exists a stream S, a predicate P, and a time point i with $i \geq d$ such that $\Pi \cup S \models P(i-d)$ and $\Pi \cup S\restriction_{[0,i]} \not\models P(i-d)$. By compactness of first-order logic, we can assume w.l.o.g. that S is finite; furthermore, we can assume that S is object-free since so is Π. Let n be the maximum time point occurring in S and let $E_0, \ldots, E_n$ be sets of EDB predicates such that $S = \{A(i) \mid A \in E_i,\ 0 \leq i \leq n\}$. Let $u_1 = \langle i - d, P, E_0, \ldots, E_n\rangle$, and let $u_2 = \langle i - d, P, E_0, \ldots, E_i\rangle$. We have $u_2 \in \mathsf{notimplies}(\Pi, \mathcal{I})$ since $\Pi \cup S\restriction_{[0,i]} \not\models P(i-d)$. Furthermore, $u_1 \in \mathsf{implies}(\Pi, \mathcal{I})$ since $\Pi \cup S \models P(i-d)$; hence $u_2 \in \mathsf{prefimplies}(\Pi, \mathcal{I})$ since u_2 is a prefix of u_1. Let $\sigma_0, \ldots, \sigma_n$ be such that $u_1 = \sigma_0, \ldots, \sigma_n$. Let $u_3 = \sigma_{i-d+1}, \ldots, \sigma_i$, and let $u_4 = \sigma_0, \ldots, \sigma_{i-d}$. It suffices to show $u_3 \in \mathsf{delay}(\Pi, \mathcal{I})$, for which it remains to show that the following two conditions hold: (i) the last letter of u_4 is an answer letter, and (ii) $u_4u_3 \in \mathsf{prefimplies}(\Pi, \mathcal{I}) \cap \mathsf{notimplies}(\Pi, \mathcal{I})$. The last letter of u_4 is σ_{i-d}, which is an answer letter by its definition and the definition of u_1. We have $u_4u_3 \in \mathsf{prefimplies}(\Pi, \mathcal{I}) \cap \mathsf{notimplies}(\Pi, \mathcal{I})$ since $u_4u_3 = u_2$ and we have already argued both $u_2 \in \mathsf{prefimplies}(\Pi, \mathcal{I})$ and $u_2 \in \mathsf{notimplies}(\Pi, \mathcal{I})$.

Statement 3 is an immediate corollary of the second statement. □

Note that the delay language is prefix-closed. This is clear from Figure 5.2, where we can see that, if we remove the last letter from $c_1 \ldots c_d$, the resulting word $c_1 \ldots c_{d-1}$ is still in the delay language, since now $b_1 \ldots b_n a c_1 \ldots c_{d-1}$ is in notimplies because it

encodes a shorter stream, and $b_1 \dots b_n a c_1 \dots c_{d-1} c_d v_1 \dots v_m$ still works as a witness, with the only conceptual difference that now c_d should be considered as being one of the red letters.

Proposition 5.7. *For Π and object-free program and $\mathcal{I}$ a subset of its IDB predicates, we have that $c_1, \dots, c_d \in \mathsf{delay}(\Pi, \mathcal{I})$ implies $c_1, \dots, c_i \in \mathsf{delay}(\Pi, \mathcal{I})$ for every $i \leq d$.*

Proof. Let $w_1 = c_1 \dots c_d$ be a word in $\mathsf{delay}(\Pi, \mathcal{I})$. Thus, there is a word $w_2 = b_1 \dots b_n a c_1 \dots c_d \in \mathsf{notimplies}(\Pi, \mathcal{I})$ and a word $w_3 = b_1 \dots b_n a c_1 \dots c_d v_1 \dots v_m \in \mathsf{implies}(\Pi, \mathcal{I})$. We have that w_2 is of the form $\langle n, G, E_1, \dots, E_{n+d+1} \rangle$. By the definition of $\mathsf{notimplies}(\Pi, \mathcal{I})$, we have $\Pi \cup S \not\models G(n)$ for $S = \{A(i) \mid 0 \leq i \leq n+d,\ A \in E_{i+1}\}$, hence $\Pi \cup S\restriction_{[0,n+d]} \not\models G(n)$ by monotonicity of entailment, and hence $b_1 \dots b_n a c_1 \dots c_{d-1} \in \mathsf{notimplies}(\Pi, \mathcal{I})$. We conclude that $c_1 \dots c_{d-1} \in \mathsf{delay}(\Pi, \mathcal{I})$, since we already know that $w_3 \in \mathsf{implies}(\Pi, \mathcal{I})$. □

5.3.2 Automata Construction

In this section, we show that all the languages given in Definition 5.5 can be recognised by finite automata, which can be computed in polynomial space. Our automata are for programs in a normal form in which all rules have a forward and backward radius of at most one. The convenience of such a normal form is that it allows the automata to reason about any time point τ by taking into account only the previous time point $\tau - 1$, the considered time point τ, and the next time point $\tau + 1$.

Definition 5.8. *A program is* normal *if it contains only rules of the forms:*

(i) $P_1(\mathbf{s}_1, t+1) \to P(\mathbf{s}, t)$;

(ii) $P_1(\mathbf{s}_1, t-1) \to P(\mathbf{s}, t)$; *or*

(iii) $\bigwedge_i P_i(\mathbf{s}_i, t) \to P(\mathbf{s}, t)$.

Theorem 5.9 justifies that our restriction to normal programs in the construction of automata is without loss of generality: each (object-free) program Π can be normalised in polynomial time (by introducing fresh predicates) to a program $\Xi(\Pi)$

such that the languages in Definition 5.5 are preserved by the transformation when parametrised by the set of predicates $\mathcal{I}_\Pi$ of the initial program Π.

Theorem 5.9. *Let Π be an object-free program with maximum forward radius ρ_1 and maximum backward radius ρ_2. Let $\Xi(\Pi)$ be the (normal) program consisting of the following rules for each predicate P occurring in Π, where P^i is a fresh (IDB) unary predicate uniquely associated with P and $i \in [-\rho_2, \rho_1]$, and t is a time variable:*

(i) $P(t) \to P^0(t)$;

(ii) $P^i(t+1) \to P^{i+1}(t)$ *for each* $i \in [0, \rho_1 - 1]$;

(iii) $P^i(t-1) \to P^{i-1}(t)$ *for each* $i \in [-\rho_2 + 1, 0]$; *and*

(iv) $\bigwedge_i P_i^{k_i - k}(t') \to P(t')$ *for each rule in* Π *of the form* $\bigwedge_i P_i(t' + k_i) \to P(t' + k)$.

Then, the following statements hold, where $\mathcal{I}$ the set of IDB predicates in Π:

1. $\mathsf{implies}(\Pi, \mathcal{I}_\Pi) = \mathsf{implies}(\Xi(\Pi), \mathcal{I}_\Pi)$;
2. $\mathsf{notimplies}(\Pi, \mathcal{I}_\Pi) = \mathsf{notimplies}(\Xi(\Pi), \mathcal{I}_\Pi)$; *and*
3. $\mathsf{delay}(\Pi, \mathcal{I}_\Pi) = \mathsf{delay}(\Xi(\Pi), \mathcal{I}_\Pi)$.

Proof. We claim that $\Pi \cup D \models \alpha$ if and only if $\Xi(\Pi) \cup D \models \alpha$, for each set D of facts and each fact α over predicates in Π. The first two statements in the theorem follow as a straightforward consequence of this claim and the definition of the relevant languages; in turn, the third statement follows by the definition of the delay language and the previous two statements.

It is straightforward to show (by induction on i) that

$$\Xi(\Pi) \cup D \models P(\tau) \text{ if and only if } \Xi(\Pi) \cup D \models P^i(\tau - i) \tag{5.11}$$

for each P in Π, each predicate P^i corresponding to P in $\Xi(\Pi)$, and each set D of facts over predicates in Π.

Assume now that $\Pi \cup D \models \alpha$. Let δ be a derivation of α from $\Pi \cup D$. We show $\Xi(\Pi) \cup D \models \alpha$ by induction on the height h of δ. In the base case, we have $h = 0$

and hence $\alpha \in \Pi \cup D$. But then, since Π does not mention time points, $\alpha \in D$, and hence $\Xi(\Pi) \cup D \models \alpha$. In the inductive case, we have $h > 0$, and we assume that the claim holds for $h - 1$. Let r be the rule labelling the root of δ, and let r' be a rule in Π such that r is an instance of r'. Then r' has the form $\bigwedge_{i=1}^{m} P_i(t + k_i) \rightarrow P(t + k)$ and r is of the form $\bigwedge_{i=1}^{m} P_i(\tau - k + k_i) \rightarrow P(\tau)$ (where τ and $\tau - k + k_i$ are time points and $P(\tau) = \alpha$). By the definition of Ξ, it follows that $\Xi(\Pi)$ contains a rule of the form $\bigwedge_{i=1}^{m} P_i^{k_i - k}(t') \rightarrow P(t')$, and hence it suffices to show $\Xi(\Pi) \cup D \models P_i^{k_i - k}(\tau)$ for each $i \in [1, m]$. To this end, note that, by assumption, for each $i \in [1, m]$, we have $\Pi \cup D \models P_i(\tau - k + k_i)$, hence $\Xi(\Pi) \cup D \models P_i(\tau - k + k_i)$ by the inductive hypothesis, and hence $\Xi(\Pi) \cup D \models P_i^{k_i - k}(\tau)$ by (5.11).

Assume $\Xi(\Pi) \cup D \models \alpha$. Let δ be a derivation of α from $\Xi(\Pi) \cup D$, and let $\sharp\delta$ be the maximum number of nodes on a root-to-leaf path in δ that are labelled with an instance of a rule of the form (iv). We show $\Pi \cup D \models \alpha$ by induction on $\sharp\delta$. In the base case, we have $\sharp\delta = 0$, and hence $\alpha \in D$. Consequently, $\Pi \cup D \models \alpha$, as required. In the inductive case, let r be the rule labelling the root of δ, and let r' be a rule in $\Xi(\Pi)$ such that r is an instance of r'. By assumption, r' has the form $\bigwedge_{i=1}^{m} P_i^{k_i - k}(t') \rightarrow P(t')$, and thus r has the form $\bigwedge_{i=1}^{m} P_i^{k_i - k}(\tau) \rightarrow P(\tau)$. Moreover, Π contains a rule of the form $\bigwedge_{i=1}^{m} P_i(t' + k_i) \rightarrow P(t' + k)$. Therefore, it suffices to show $\Pi \cup D \models P_i(\tau + k_i - k)$ for each $i \in [1, m]$. To this end, note that, by assumption, for each $i \in [1, m]$, we have $\Xi(\Pi) \cup D \models P_i^{k_i - k}(\tau)$, and hence $\Xi(\Pi) \cup D \models P_i(\tau + k_i - k)$ by (5.11). Moreover, by the definition of Ξ, the subderivation of δ deriving $P_i^{k_i - k}(\tau)$ must include a derivation δ' of $P_i(\tau + k_i - k)$, and consequently $\sharp\delta' < \sharp\delta$. Hence, the inductive hypothesis applies to δ' yielding $\Pi \cup D \models P_i(\tau + k_i - k)$, as required. $\square$

Our next step will be to define a non-deterministic automaton $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ that recognises the language $\mathsf{implies}(\Pi, \mathcal{I})$. Intuitively, the automaton checks whether an input word corresponds to a finite stream prefix S and fact α such that $\Pi \cup S \models \alpha$ by guessing a finite set of facts M such that $\Pi \cup S \models M$ and $\alpha \in M$. The key challenge in the construction of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ is making sure that each fact in M can be derived from $\Pi \cup S$; indeed, a derivation of a fact from $\Pi \cup S$ may

involve facts at many different time points, whereas a transition of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ can only check whether a fact holding at a time point τ is entailed by facts holding at $[\tau - 1, \tau + 1]$. Thus, set M is checked for the property that, for each such τ, $\Pi \cup M\restriction_{\tau-1} \cup S\restriction_{\tau} \cup M\restriction_{\tau+1} \models M\restriction_{\tau}$. The automaton checks such property incrementally one time point at a time; during this process, the set $M\restriction_{\tau-1}$ can be thought of as a set of facts that the automaton has already checked to be entailed, the set $M\restriction_{\tau}$ as the facts being checked, and the set $M\restriction_{\tau+1}$ as facts that the automaton assumes to hold with the promise that it will check at the next transition whether they actually hold. These checks alone, however, are not enough to ensure $\Pi \cup S \models M$; for instance, for $\Pi = \{A(t-1) \to B(t), B(t+1) \to A(t)\}$ and $S = \emptyset$, set $M = \{A(0), B(1)\}$ satisfies the above property yet is clearly not entailed by $\Pi \cup S$. It is easily seen that all such spurious sets contain a fact that can only be derived using itself as an assumption. Thus, to exclude such sets, the states of the automaton are enriched with dependency information, requiring that no fact depends on itself in a minimal derivation.

Similarly to the way time is encoded away in the input words, also the automaton encodes time by exploiting the natural order of states in a run of the automaton. For instance, the automaton encodes $M\restriction_{\tau}$ by the set M_τ of predicates occurring in $M\restriction_{\tau}$; note that, once time arguments are encoded in the order, it only remains to store predicates, since programs are object-free. Due to such encoding of sets of facts as sets of predicates, it is convenient to introduce the notation $X(\tau) = \{P(\tau) \mid P \in X\}$ for X a set of unary predicates and τ a time point. For instance, this notation allows us to easily write the relationship $M\restriction_{\tau} = M_\tau(\tau)$ between $M\restriction_{\tau}$ and M_τ.

Before proceeding with the definition of the automaton, we illustrate the concepts introduced above in Figure 5.3.

Definition 5.10. *Let Π be an object-free normal program, let $\mathcal{P}$ be the set of all predicates occurring in Π, and let $\mathcal{I}$ be a set of IDB predicates in Π. Then $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ is the non-deterministic finite automaton $\langle \Sigma(\Pi, \mathcal{I}), S, S_{\text{init}}, F, \Delta \cup \Delta_\varepsilon \rangle$ defined as follows.*

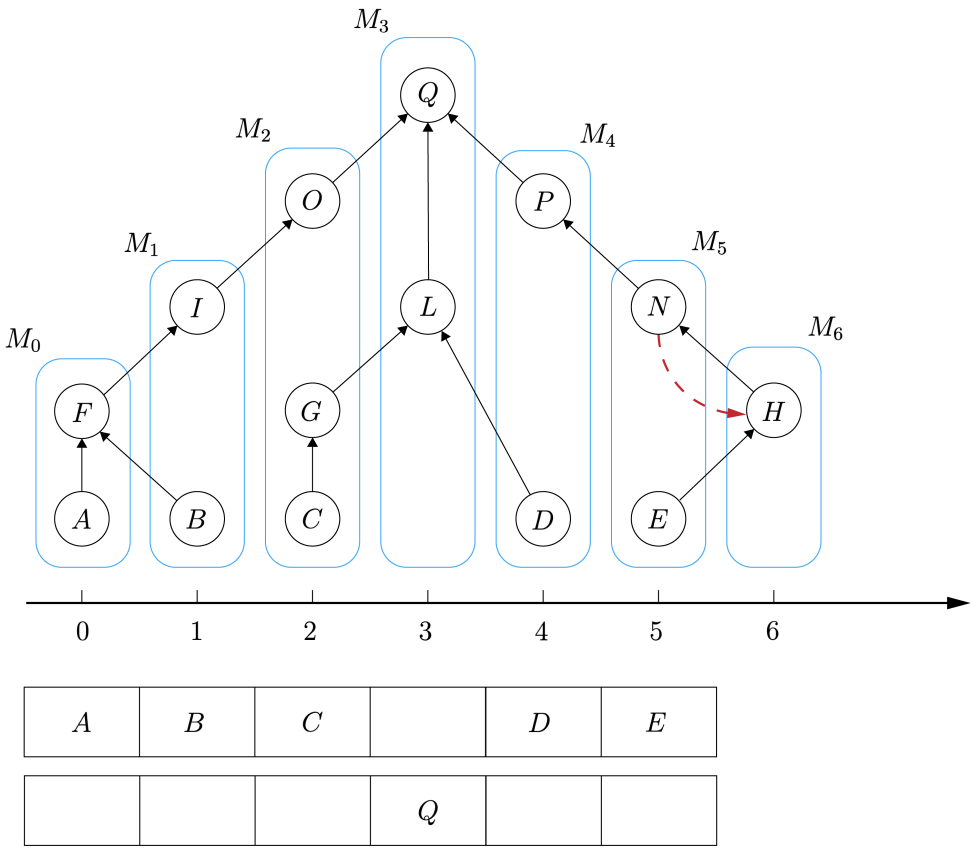


Figure 5.3: Below the timeline, we have an input word represented as two separate tapes: the upper one represents the input stream $S = \{A(0), B(1), C(2), D(4), E(5)\}$ and the lower one represents the answer fact $Q(3)$. Above the timeline, we have a set of facts $M_0 \cup \cdots \cup M_6$ which encodes a derivation of $Q(3)$ from $S \cup \Pi$ where Π is the considered program. A predicate X in the figure positioned with respect to a point τ of the timeline encodes the fact $X(\tau)$; for instance, A encodes $A(1)$. A box labelled with M_τ encloses the content of M_τ; for instance, the left-most box encodes the content $\{A, F\}$ of M_0. Arrows denote how IDB facts are derived from other facts, where multiple incoming arrows are to be interpreted as a logical 'and'; for instance, $F(0)$ is derived by the rule $A(0) \wedge B(1) \rightarrow F(0)$. The automaton keeps track of the transitive closure of the graph induced by the derivation, and prevents cycles from appearing in the closure, so to have a well-formed derivation. For instance, when the automaton processes time 6, it may consider to derive $H(6)$ from $N(5)$, but it avoids to do so because a cycle would form—see the red dashed arrow.

- *The set S of all states consists of each 5-tuple $\langle a, X, Y, \Pi', \Gamma\rangle$ where $a \in \{1,2,3\}$, $X, Y \subseteq \mathcal{P}$, $\Pi' \subseteq \Pi\{t \mapsto 1\}$,[1] and $\Gamma \subseteq \mathcal{P} \times \mathcal{P}$.*

- *The set S_{init} of initial states consists of each state in S of the form $\langle 1, \emptyset, X, \emptyset, \emptyset\rangle$.*

- *The set F of final states consists of each state in S of the form $\langle a, X, \emptyset, \Pi', \Gamma\rangle$ with $a \in \{2,3\}$.*

- *Δ consists of each transition $\langle a_1, X, Y, \Pi_1, \Gamma_1\rangle \xrightarrow{\langle U,b\rangle} \langle a_2, Y, Z, \Pi_2, \Gamma_2\rangle$ such that:*

  (2.1) if $b \in \mathcal{I}$ then $b \in Y$, $a_1 = 1$, and $a_2 = 2$;

  (2.2) if $b = \Box$ then $a_1 = a_2 = 1$ or $a_1 = a_2 = 2$;

  (2.3) $X(0) \cup U(1) \cup Z(2) \cup \Pi_2 \models Y(1)$;

  (2.4) Γ_2 contains the transitive closure of the binary relation consisting of each pair $\langle P, R\rangle$ such that either

    (2.4.1) $P(1)$ depends on $R(1)$ in Π_2, or

    (2.4.2) there exists a predicate T such that $P(1)$ depends on $T(0)$ in Π_2 and $T(1)$ depends on $R(2)$ in Π_1, or

    (2.4.3) there exists $\langle T_1, T_2\rangle \in \Gamma_1$ such that $P(1)$ depends on $T_1(0)$ in Π_2 and $T_2(1)$ depends on $R(2)$ in Π_1.

  (2.5) there exists no predicate P such that $\langle P, P\rangle \in \Gamma_2$;

- *Δ_ε consists of each transition $\langle a, X, Y, \Pi_1, \Gamma_1\rangle \xrightarrow{\varepsilon} \langle 3, Y, Z, \Pi_2, \Gamma_2\rangle$ such that:*

  (3.1) $a \in \{2,3\}$;

  (3.2) $X(0) \cup Z(2) \cup \Pi_2 \models Y(1)$;

  (3.3) Conditions (2.4) and (2.5) hold.

[1] In using substitutions of the form $\{t \mapsto \tau\}$, we assume that t is the only time variable mentioned in the program. The assumption is w.l.o.g. because, due to temporal-guardedness, any other time variable can be renamed to t without changing the semantics of the program.

In a state $\langle a, X, Y, \Pi', \Gamma \rangle$ of automaton $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$, the integer a describes in which of the three stages the automaton currently is, the two sets X and Y represent two consecutive subsets $M\restriction_\tau$ and $M\restriction_{\tau+1}$ of the set M described above, Π' is an instance of Π over the time interval $[0, 2]$ which encodes labels of the derivation the automaton (implicitly) builds, and the pairs in Γ describe how the derived facts depend on each other. The three stages of the automaton are: (*i*) expecting to see an answer letter, (*ii*) having seen an answer letter, and (*iii*) having performed at least one ε-transition. The automaton goes through the three stages sequentially, and hence ensures that it accepts only if it sees one answer letter and does not go back to reading letters of the input word after having performed an ε-transition. Time in X and Y is implicit in the position they occur in a run, whereas time in Π' is relative and can be mapped to absolute time through the position Π' occurs in a run; specifically, time 1 in Π' stands for the current time point, time 0 for the previous time point, and 2 for the next time point. Each Π' is subset of $\Pi\{t \mapsto 1\}$, which denotes the instance of Π obtained by substituting the time variable with 1.

The transitions of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ do three main things. First, they progress through the three stages described above. Second, they enforce the above-mentioned property $\Pi \cup M\restriction_{\tau-1} \cup S\restriction_\tau \cup M\restriction_{\tau+1} \models M\restriction_\tau$; in particular, the transitions in Δ read $S\restriction_\tau$ from the input word, and the ε-transitions in Δ_ε allow to reason beyond the time points of the input, implicitly taking $S\restriction_\tau = \emptyset$. Third, they incrementally maintain the dependency information in Γ and avoid the presence of cycles. Considering a transition $\langle \_, X, Y, \_, \_ \rangle \longrightarrow \langle \_, Y, Z, \_, \_ \rangle$, we can see that X represents the facts already checked to hold at the previous time point, set Y the facts to be checked now, and Z a set of facts that we assume to hold for the next time point and we promise to check at the next transition. Following this line of reasoning, a state $\langle a, X, Y, \Pi', \Gamma \rangle$ is initial if no facts have been derived in the previous time points, and hence if it has $X = \Pi' = \Gamma = \emptyset$; and it is final if an answer fact has been read and all assumptions have been proved, and hence if it has $a \in \{2, 3\}$ and $Y = \emptyset$.

The properties and intuitions given above are stated clearly and proved in the following theorem.

Theorem 5.11. *Let* Π *be an object-free normal program and* $\mathcal{I}$ *a set of IDB predicates in* Π. *Automaton* $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ *recognises the language* $\mathsf{implies}(\Pi, \mathcal{I})$.

Proof. Let Π and $\mathcal{I}$ be as required. We first prove that $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ accepts each word in $\mathsf{implies}(\Pi, \mathcal{I})$. Our proof relies on properties of derivations in Temporal Datalog, as well as on the connection between certain types of derivations and runs in the automaton. These properties are formalised and proved in Claims 5.12 and 5.13, respectively. We say that a derivation δ is *uniform* if, for each (ground) atom α occurring in δ, all the α-subderivations of δ coincide, and it is straightforward to show that any derivation of a fact can be transformed into a uniform derivation of the same fact (whenever a derivation has two distinct subderivations of the same fact, one can safely replace one with the other and still obtain a derivation).

Claim 5.12. *For* Π *a program,* D *a set of EDB facts, and* α *a fact, let* δ *be a uniform derivation of* α *from* $\Pi \cup D$, *and let* Π_δ *be the set of labels of* δ. *Then no fact depends on itself in* Π_δ.

It suffices to show that, if β and γ are facts such that β depends on γ in Π_δ, then then the (unique) β-subderivation of δ has a strict γ-subderivation. This is shown by induction on the height $k \geq 1$ of the partial derivation justifying the dependency relation. In the base case, $k = 1$ and there is a rule $r \in \Pi_\delta$ such that β is the head of r and γ is in the body of r. Consider a subderivation δ_1 of δ whose root has label r. Clearly, δ_1 is a β-subderivation of δ and, moreover, has an immediate γ-subderivation, as required. In the inductive case, $k > 1$, and hence there is a fact ψ and a rule $r \in \Pi_\delta$ such that β is the head of r, ψ is in the body of r, and ψ depends on γ in Π_δ via $k - 1$ steps. Consider a subderivation δ_1 of δ whose root has label r. Then δ_1 is a β-subderivation of δ and has an immediate ψ-subderivation δ_2. Furthermore, by the inductive hypothesis, δ_2 (which is the unique ψ-subderivation of δ since δ is uniform) has a strict γ-subderivation δ_3. Thus, δ_3 is a strict subderivation of δ_1, which concludes the proof of the claim.

Claim 5.13. *For each finite set of EDB facts D, each uniform derivation δ of a fact $P(\tau)$ with $P \in \mathcal{I}$ from $\Pi \cup D$, and each $m \geq -1$, the sequence $s_{-1}\sigma_0 s_0 \dots s_{m-1}\sigma_m s_m$ is a run of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ where:*

- *for each $i \in [-1, m]$, $s_i = \langle a_i, X_i, X_{i+1}, \Pi_i, \Gamma_i \rangle$ where, for Π'_i the set of all rules labelling δ that are not facts and have time point i in the head,*
  - *$a_i = 1$ if $i < \tau$, $a_i = 2$ if $\tau \leq i \leq \tau_{\max}$, and $a_i = 3$ if $i > \tau_{\max}$ for $\tau_{\max}$ the maximum time point in $D \cup \{P(\tau)\}$;*
  - *$X_i = \{Q \mid Q(i) \in F\}$ for F the union of D with the set of all rule heads occurring in δ (note that $X_{-1} = \emptyset$);*
  - *Π_i is obtained from Π'_i by replacing each time point τ' with $\tau' - i + 1$ (note that $\Pi_{-1} = \emptyset$ as $\Pi'_{-1} = \emptyset$);*
  - *$\Gamma_i = \{\langle Q, R \rangle \mid Q(i)$ depends on $R(i)$ in $\bigcup_{j=0}^{i} \Pi'_j\}$ (note that $\Gamma_{-1} = \emptyset$);*
- *for each $i \in [0, \tau_{\max}]$, $\sigma_i = \langle U_i, b_i \rangle$ where $U_i = \{Q \mid Q(i) \in D\}$, $b_i = \square$ if $i \neq \tau$, and $b_\tau = P$;*
- *for each $i \in [\tau_{\max} + 1, m]$, $\sigma_i = \varepsilon$.*

If $m = -1$, the claim holds since $s_{-1} = \langle 1, \emptyset, X_0, \emptyset, \emptyset \rangle$, and hence s_{-1} is an initial state of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$. Hence, without loss of generality, let $0 \leq n \leq m$ be arbitrary. It suffices to show that the transition $s_{n-1} \xrightarrow{\sigma_n} s_n$ is in Δ if $n \leq \tau_{\max}$, and in Δ_ε otherwise. We consider the two cases separately.

Suppose $n \leq \tau_{\max}$. Then $\sigma_n = \langle U_n, b_n \rangle$. It suffices to show satisfaction of Conditions (2.1)–(2.5) in the definition of Δ.

Condition (2.1). Assume $b_n \in \mathcal{I}$. Then $b_n = P$ and $n = \tau$, and hence it suffices to show $P \in X_\tau$, $a_{\tau-1} = 1$, and $a_\tau = 2$. We have $P \in X_\tau$ because δ contains a rule with head $P(\tau)$, while $a_{\tau-1} = 1$ and $a_\tau = 2$ is immediate by definition.

Condition (2.2). Assume $b_n = \square$. Then $n \neq \tau$. If $n < \tau$, we have $a_{n-1} = a_n = 1$ by definition, while $n > \tau$ implies $n - 1 \geq \tau$, and hence $a_{n-1} = a_n = 2$, as required.

Condition (2.3). We show that $X_{n-1}(0) \cup U_n(1) \cup X_{n+1}(2) \cup \Pi_n \models X_n(1)$. To this end, by construction, it suffices to show that $X_{n-1}(n-1) \cup U_n(n) \cup X_{n+1}(n+1) \cup \Pi'_n \models X_n(n)$, which can be equivalently restated as $F\restriction_{n-1} \cup D\restriction_n \cup F\restriction_{n+1} \cup \Pi'_n \models F\restriction_n$. In turn, $F\restriction_{n-1} \cup D\restriction_n \cup F\restriction_{n+1} \cup \Pi'_n \models F\restriction_n$ can be shown by a straightforward induction on derivations.

Condition (2.4). By the definition of Γ_n, it suffices to show that, for each pair of predicates Q and R, $Q(n)$ depends on $R(n)$ in $\bigcup_{j=0}^{n} \Pi'_j$ if any of the following conditions holds:

(a) $Q(1)$ depends on $R(1)$ in Π_n,

(b) there exists a predicate T such that $Q(1)$ depends on $T(0)$ in Π_n and $T(1)$ depends on $R(2)$ in Π_{n-1}, or

(c) there exists $\langle T_1, T_2 \rangle \in \Gamma_{n-1}$ such that $Q(1)$ depends on $T_1(0)$ in Π_n and $T_2(1)$ depends on $R(2)$ in Π_{n-1}.

We consider the three cases separately. In Case (a), $Q(n)$ depends on $R(n)$ in Π'_n, and hence also in $\bigcup_{j=0}^{n} \Pi'_j$. In Case (b), $Q(n)$ depends on $T(n-1)$ in Π'_n and $T(n-1)$ depends on $R(n)$ in Π'_{n-1}, and hence $Q(n)$ depends on $R(n)$ in $\bigcup_{j=0}^{n} \Pi'_j$. In Case (c), $Q(n)$ depends on $T_1(n-1)$ in Π'_n and $T_2(n-1)$ depends on $R(n)$ in Π'_{n-1}. Furthermore, since $\langle T_1, T_2 \rangle \in \Gamma_{n-1}$, we also have that $T_1(n-1)$ depends on $T_2(n-1)$ in $\bigcup_{j=0}^{n-1} \Pi'_j$, and hence $Q(n)$ depends on $R(n)$ in $\bigcup_{j=0}^{n} \Pi'_j$.

Condition (2.5). The condition holds by Claim 5.12 since δ is uniform.

Next, suppose $n > \tau_{\max}$. Then $\sigma_n = \varepsilon$. It suffices to show satisfaction of Conditions (3.1)–(3.3) in the definition of Δ_ε.

Condition (3.1). By definition, we have $a_n = 3$, $a_{n-1} = 2$ if $n = \tau_{\max} + 1$ and $a_{n-1} = 3$ if $n > \tau_{\max} + 1$, as required.

Condition (3.2). Similarly to the case of Condition (2.3), it suffices to show $F\restriction_{n-1} \cup F\restriction_{n+1} \cup \Pi'_n \models F\restriction_n$. This follows from the corresponding claim in Condition (2.3)—$F\restriction_{n-1} \cup D\restriction_n \cup F\restriction_{n+1} \cup \Pi'_n \models F\restriction_n$—since the claim also holds for $n > \tau_{\max}$ and since $D\restriction_n$ is empty (as $n > \tau_{\max}$).

Condition (3.3). The arguments for Conditions (2.4) and (2.5) do not depend on $n \leq \tau_{\max}$, so both conditions still hold for $n > \tau_{\max}$.

This concludes the proof of Claim 5.13.

Let now $w = \langle \tau, P, U_0, \ldots, U_n \rangle$ be a word in $\mathsf{implies}(\Pi, \mathcal{I})$. We show that automaton $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ accepts w. By definition, we have $\Pi \cup D \models P(\tau)$ for $D = \{A(j) \mid A \in U_j,\ 0 \leq j \leq n\}$. Let δ be a uniform derivation of $P(\tau)$ from $\Pi \cup D$, and let m be the maximum between n and the maximum time point occurring in δ. By Claim 5.13, $\rho = s_{-1}\sigma_0 s_0 \ldots s_{m-1}\sigma_m s_m$ is a run of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ where (i) $w = \sigma_0, \ldots, \sigma_n$, (ii) $\sigma_i = \varepsilon$ for each $i \in [n+1, m]$, and (iii) $s_m = \langle a, X, Y, \Pi, \Gamma \rangle$ where $a \in \{2, 3\}$ and $Y(m+1)$ is the set of atoms with time argument $m+1$ occurring in either D or δ. Note that $Y = \emptyset$ since D and δ contain no fact with time argument $m+1$ by the choice of m. Thus, s_m is a final state, meaning that ρ is an accepting run of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ on w.

We finally show that $\mathsf{implies}(\Pi, \mathcal{I})$ contains each word accepted by $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$. Let $w = \sigma_0 \ldots \sigma_n$ be a word accepted by $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$. Then there exists a run $s_{-1}\sigma_0 s_0 \ldots s_{m-1}\sigma_m s_m$ of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ where $m \geq n$ and s_m is final. Since s_m is final, by the definition of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$, there exist an integer $k \in [0, n]$, a predicate $P \in \mathcal{I}$, and sets $U_0, \ldots, U_n$ of EDB predicates such that $\sigma_k = \langle U_k, P \rangle$, $\sigma_j = \langle U_j, \Box \rangle$ for each $j \in [0, n]$ with $j \neq k$, and $\sigma_j = \varepsilon$ for each $j \in [n+1, m]$. It follows that $w = \langle k, P, U_0, \ldots, U_n \rangle$. Thus, it remains to show that $\Pi \cup D \models P(k)$, for $D = \bigcup_{j=0}^{n} U_j(j)$. Let, for each $i \in [-1, m]$, $s_i = \langle a_i, X_i, Y_i, \Pi_i, \Gamma_i \rangle$. Since $\sigma_k = \langle U_k, P \rangle$, we have $P \in X_k$, and hence it suffices to show that, for each $i \in [0, m]$, $\Pi \cup D \models X_i(i)$. To this end, we first show the following claims, where, for each $i \in [0, m]$, $\Pi'_i = \bigcup_{j=0}^{i} \Pi\{t \mapsto j\}$.

Claim 5.14. *For each $i \in [0, m]$, Γ_i contains each pair $\langle P, R \rangle$ such that $P(i)$ depends on $R(i)$ in Π'_i.*

We show the claim by induction on i. In the base case ($i = 0$), whenever $P(0)$ depends on $R(0)$ in Π'_0, we have that $P(1)$ depends on $R(1)$ in Π_0, and hence $\langle P, R \rangle \in \Gamma_0$ by Condition (2.4.1). In the inductive case, we have $0 < i \leq m$ and we assume that the claim holds for $i - 1$. Suppose that $P(i)$ depends on $R(i)$ in Π'_i via k steps. We

show $\langle P, R\rangle \in \Gamma_i$ by induction on k. In the base case ($k = 1$), the claim again holds by Condition (2.4.1) since $P(1)$ depends on $R(1)$ in Π_i. In the inductive case, we have $k > 1$ and we assume that the claim holds for $k-1$. Then there is a rule $r \in \Pi'_i$ with head $P(i)$ and a body atom $T(j)$ that depends on $R(i)$ in Π'_i via $k-1$ steps. We have $j \in \{i-1, i\}$, since Π is normal and no rule of Π'_i has head time argument $i+1$. If $i = j$, then $\langle P, T\rangle \in \Gamma_i$ by Condition (2.4.1) since $P(1)$ depends on $T(1)$ in Π_i, and $\langle T, R\rangle \in \Gamma_i$ by the inner inductive hypothesis; $\langle P, R\rangle \in \Gamma_i$ then follows by the transitivity of Γ_i. Next, assume $j = i-1$. Since Π is normal, either $T(i-1)$ depends on a fact $R'(i)$ via one step in Π'_{i-1} or $T(i-1)$ depends (via possibly more then one step) on a fact $T'(i-1)$, which in turn depends on a fact $R'(i)$. We consider the two cases separately. In the first case, we have $\langle P, R'\rangle \in \Gamma_i$ by Condition (2.4.2) since $P(1)$ depends on $T(0)$ in Π_i and $T(1)$ depends on $R'(2)$ in Π_{i-1}. Furthermore, either $R' = R$ (which immediately implies the claim) or $R'(i)$ depends on $R(i)$ in Π'_i via at most $k-1$ steps, which implies $\langle R', R\rangle \in \Gamma_i$ by the inner inductive hypothesis. Thus, $\langle P, R\rangle \in \Gamma_i$ holds by the transitivity of Γ_i. In the second case, we have $\langle P, R'\rangle \in \Gamma_i$ by Condition (2.4.3) since $P(1)$ depends on $T(0)$ in Π_i, $\langle T, T'\rangle \in \Gamma_{i-1}$ by the outer inductive hypothesis, and $T'(1)$ depends on $R'(2)$ in Π_{i-1}. Then, as before, either $R' = R$ or $R'(i)$ depends on $R(i)$ in Π'_i via at most $k-1$ steps, and hence, once again, $\langle P, R\rangle \in \Gamma_i$. This concludes the proof of Claim 5.14.

Claim 5.15. *For each $i \in [0, m]$, we have $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Pi'_i \models X_i(i)$.*

We show the claim by induction on i. The base case ($i = 0$) holds by Condition (2.3) as $X_{-1} = \emptyset$ because s_{-1} is an initial state. In the inductive case, we have $0 < i \leq m$ and we assume $D\restriction_{[0,i-1]} \cup Y_{i-1}(i) \cup \Pi'_{i-1} \models X_{i-1}(i-1)$, or, equivalently, $D\restriction_{[0,i-1]} \cup X_i(i) \cup \Pi'_{i-1} \models X_{i-1}(i-1)$ since $Y_{i-1} = X_i$. By Conditions (2.3) and (3.2), we have $X_{i-1}(i-1) \cup U(i) \cup Y_i(i+1) \cup \Pi\{t \mapsto i\} \models X_i(i)$ for $U = U_i$ if $i \leq n$ and $U = \emptyset$ otherwise. Thus, it suffices to show $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Pi'_i \models \alpha$ for α a fact with time argument i that is entailed by $X_{i-1}(i-1) \cup U(i) \cup Y_i(i+1) \cup \Pi\{t \mapsto i\}$. We show this by induction on the minimal rank k of α in Π'_i, which exists because no fact depends on itself in Π'_i by Claim 5.14 and Condition (2.5). In the base case, we have $k = 0$,

hence α does not depend on any fact in Π'_i, and hence $\alpha \in U(i) \subseteq D\restriction_{[0,i]}$. In the inductive case, we have $k \geq 1$ and we assume that the claim holds for $k-1$. Let δ be a derivation of α from $X_{i-1}(i-1) \cup U(i) \cup Y_i(i+1) \cup \Pi\{t \mapsto i\}$, r be the rule labelling the root of δ, and β be a body atom of r. It suffices to show $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Pi'_i \models \beta$. Let j be the time argument of β. We have $j \in \{i-1, i, i+1\}$ since Π is normal, and we consider the three cases separately. In the first case, we have $j = i$, and hence the claim holds by the inner inductive hypothesis since β has minimal rank at most $k-1$ in Π'_i. In the second case, we have $j = i+1$, and hence $\beta \in Y_i(i+1)$ since β is not an instance of a head in $\Pi\{t \mapsto i\}$. In the third case, we have $j = i-1$, and hence $\beta \in X_{i-1}(i-1)$ since β is not an instance of any head in $\Pi\{t \mapsto i\}$. The outer inductive hypothesis yields $D\restriction_{[0,i-1]} \cup X_i(i) \cup \Pi'_{i-1} \models \beta$, and hence $D\restriction_{[0,i-1]} \cup A \cup \Pi'_{i-1} \models \beta$ for A the subset of $X_i(i)$ where each fact has minimal rank at most $k-1$ in Π'_i. Then, $D\restriction_{[0,i]} \cup Y_i(i+1) \cup \Pi'_i \models A$ holds by the inner inductive hypothesis, and hence the claim follows by transitivity of entailment. This concludes the proof of Claim 5.15.

Finally, we show $\Pi \cup D \models X_i(i)$ by induction on $m - i$. In the base case, we have $i = m$, thus $\Pi \cup D \cup Y_m(m+1) \models X_m(m)$ by Claim 5.15, and hence $\Pi \cup D \models X_m(m)$ since $Y_m = \emptyset$ (as s_m is final). In the inductive case, we have $i < m$, and we assume $\Pi \cup D \models X_{i+1}(i+1)$. By Claim 5.15, we have $\Pi \cup D \cup Y_i(i+1) \models X_i(i)$. But then, since $Y_i = X_{i+1}$, the inductive hypothesis immediately yields $\Pi \cup D \models X_i(i)$, as required. □

The automaton $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ can be computed in polynomial space because each of its elements—i.e., input letter, state, or transition—is of polynomial size, and we can guess a candidate element at a time and then easily check whether it belongs to the automaton.

Theorem 5.16. *Automaton $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ can be computed in polynomial space with respect to the size of Π for each $\mathcal{I}$; furthermore, the size of each input letter, each state, and each transition of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ is polynomial in the size of Π.*

Proof. Let Π be an object-free normal program and $\mathcal{I}$ a set of IDB predicates in Π. Note that the size of $\mathcal{I}$ is bounded by that of Π, and hence can be disregarded. It is clear that the size of each input letter, each state, and each transition of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ is polynomial in the size of Π, so it remains to show how to compute $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ in polynomial space with respect to the size of Π (disregarding the space taken by the output).

Note that we can compute and store the set $\mathcal{P}$ of predicates occurring in Π as well as the program $\Pi' = \Pi\{t \mapsto 1\}$ as both sets have polynomial size in that of Π.

We next show how to compute the transitions in Δ (as in Definition 5.10) through a 'generate and filter' algorithm—the remaining transitions as well as the input alphabet, the states, and the final states of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ can be computed in a similar way. For each $a_1, a_2 \in \{1, 2, 3\}$, each $X_1, X_2, Y_1, Y_2 \subseteq \mathcal{P}$, each $\Pi_1, \Pi_2 \subseteq \Pi'$, each $\Gamma_1, \Gamma_2 \subseteq \mathcal{P} \times \mathcal{P}$, each subset U of EDB predicates in Π, and each $b \in \mathcal{P} \cup \{\Box\}$, we check whether the triple $\langle a_1, X_1, Y_1, \Pi_1, \Gamma_1 \rangle \xrightarrow{\langle U, b \rangle} \langle a_2, X_2, Y_2, \Pi_2, \Gamma_2 \rangle$ is a transition of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$—note that this check is feasible in polynomial time with respect to the size of the triple; in particular, note that Condition (2.3) amounts to fact entailment checking for propositional Horn clauses, which can be done in polynomial time. If the triple is a valid transition, it is added to the output of the algorithm and otherwise it is ignored. In either case, the triple is deleted from memory before the next triple is considered. Clearly, this computation takes polynomial space provided that each considered triple is of polynomial size in that of Π, which we already observed above. □

We next define a non-deterministic automaton $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ that recognises the language $\mathsf{notimplies}(\Pi, \mathcal{I})$. The automaton implements the algorithm for fact non-entailment by Chomicki and Imieliński [26]. Intuitively, the automaton checks whether an input word corresponds to a finite stream prefix S and fact α such that $\Pi \cup S \not\models \alpha$ by checking whether S can be extended to an infinite model M of $\Pi \cup S$ such that $\alpha \notin M$. To this end, we first define a family of automata $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I}), n]$ that, for each $n \geq 0$, accept words representing a stream prefix S and a fact α such

that S can be extended to a *partial* model M_n that contains every fact in S, does not contain α, and is closed under the rules of Π on the interval $[0, \tau_{\max} + n]$, for $\tau_{\max}$ the maximum time point in S. It is then argued that, when n is at least $2^{2 \cdot p}$, for p the number of predicates in Π, any such partial model M_n exhibits a periodic pattern that allows it to be extended to an infinite model of $\Pi \cup S$. Thus, we define $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ as $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I}), 2^{2 \cdot p}]$.

Definition 5.17. *Let Π be an object-free normal program, let $\mathcal{P}$ be the set of all predicates occurring in Π, let $\mathcal{I}$ be a set of IDB predicates occurring in Π, and let n be a non-negative integer. Then $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I}), n]$ is the non-deterministic finite automaton $\langle \Sigma(\Pi, \mathcal{I}), S, S_{\text{init}}, F, \Delta \cup \Delta_\varepsilon \rangle$ defined as given next.*

- *The set S of all states consists of each 4-tuple $\langle a, X, Y, i \rangle$ where $a \in \{1, 2, 3\}$, $X, Y \subseteq \mathcal{P}$, and $i \in [0, n]$.*
- *The set S_{init} of initial states consists of each state in S of the form $\langle 1, \emptyset, X, 0 \rangle$.*
- *The set F of final states consists of each state in S of the form $\langle 3, X, Y, n \rangle$.*
- *Δ consists of each transition $\langle a_1, X, Y, 0 \rangle \xrightarrow{\langle U, b \rangle} \langle a_2, Y, Z, 0 \rangle$ such that:*

  (2.1) if $b \in \mathcal{I}$ then $b \notin Y$, $a_1 = 1$, and $a_2 = 2$;

  (2.2) if $b = \Box$ then $a_1 = a_2 = 1$ or $a_1 = a_2 = 2$;

  (2.3) $X(0) \cup Y(1) \cup Z(2) \models \Pi\{t \mapsto 1\} \cup U(1)$;

- *Δ_ε consists of each transition $\langle a, X, Y, i \rangle \xrightarrow{\varepsilon} \langle 3, Y, Z, i + 1 \rangle$ such that:*

  (3.1) $a \in \{2, 3\}$;

  (3.2) $X(0) \cup Y(1) \cup Z(2) \models \Pi\{t \mapsto 1\}$.

We define $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ as $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I}), 2^{2 \cdot p}]$, where p is the number of predicates occurring in Π.

Many aspects of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ resemble the ones of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$—e.g., the way the automaton abstracts time away, the three stages to detect well-formed inputs, and the use of ε-transitions to reason beyond the time points for which we have input data. However, the core functionality is quite different. Rather than checking entailments, each transition of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ checks, for increasing time points τ, whether the portion $M\restriction_{[\tau-1,\tau+1]}$ of the guessed facts M satisfies the considered program and the input facts, while making sure that the answer fact α is not present in M. Each check is less dependent on the previous/next one than in the case of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$; for instance, no dependency information needs to be stored. However, $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ has to guarantee that it performs checks for a sufficient number of time points, which it does by maintaining a counter in its state. In particular, it has to ensure that a certain number of checks is performed after reading the last letter of the input word, and hence it keeps the counter to zero when it performs non-ε-transitions, and maintains the counter when it performs ε-transitions.

The properties and intuitions about $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ are stated clearly and proved in the following theorem.

Theorem 5.18. *Let Π be an object-free normal program and $\mathcal{I}$ a set of IDB predicates in Π. Automaton $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ recognises the language $\mathsf{notimplies}(\Pi, \mathcal{I})$.*

Proof. Let Π and $\mathcal{I}$ be as required, and let $N = 2^{2 \cdot p}$ for p the number of predicates in Π. We first prove that $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ accepts each word in $\mathsf{notimplies}(\Pi, \mathcal{I})$. To this end, we begin by showing the following technical claim.

Claim 5.19. *For each finite set of EDB facts D with maximum time point $\tau_{\max}$, each fact $P(\tau)$ with $P \in \mathcal{I}$, each model M of $\Pi \cup D$ such that $P(\tau) \notin M$, and each $m \in [-1, \tau_{\max} + N]$, the sequence $s_{-1}\sigma_0 s_0 \dots s_{m-1}\sigma_m s_m$ is a run of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ where:*

- *for each $i \in [-1, m]$, $s_i = \langle a_i, X_i, X_{i+1}, c_i \rangle$ where*
  - *$a_i = 1$ if $i < \tau$, $a = 2$ if $\tau \leq i \leq \tau_{\max}$, and $a = 3$ if $i > \tau_{\max}$;*
  - *$X_i = \{Q \mid Q(i) \in M\}$;*

– $c_i = 0$ *if* $i \leq \tau_{\max}$ *and* $c_i = i - \tau_{\max}$ *otherwise;*

- *for each* $i \in [0, \tau_{\max}]$, $\sigma_i = \langle U_i, b_i \rangle$ *where* $U_i = \{Q \mid Q(i) \in D\}$, $b_i = \Box$ *if* $i \neq \tau$, *and* $b_\tau = P$*;*

- *for each* $i \in [\tau_{\max} + 1, m]$, $\sigma_i = \varepsilon$.

If $m = -1$, the claim holds since, by definition, s_{-1} is an initial state of the automaton.

Hence, without loss of generality, let $0 \leq n \leq m$ be arbitrary. It suffices to show that the transition $s_{n-1} \xrightarrow{\sigma_n} s_n$ is in Δ if $n \leq \tau_{\max}$ and in Δ_ε otherwise. We consider the two cases separately.

Suppose $n \leq \tau_{\max}$. Then $\sigma_n = \langle U_n, b_n \rangle$ and $c_n = 0$, and hence it suffices to show Conditions (2.1)–(2.3) in the definition of Δ.

Condition (2.1). Assume $b_n \in \mathcal{I}$. Then $b_n = P$, $n = \tau$, $P \notin X_n$ since $P(\tau) \notin M$, $a_{n-1} = 1$, and $a_n = 2$, as required.

Condition (2.2). Assume $b_n = \Box$. Then $n \neq \tau$. If $n < \tau$, we have $a_{n-1} = a_n = 1$ while $n > \tau$ implies $a_{n-1} = a_n = 2$, as required.

Condition (2.3). Since M is a model for $\Pi \cup D$, we have $M \models \Pi\{t \mapsto n\} \cup D\restriction_n$, and hence $M\restriction_{n-1} \cup M\restriction_n \cup M\restriction_{n+1} \models \Pi\{t \mapsto n\} \cup D\restriction_n$ as Π is normal and hence each time point occurring in $\Pi\{t \mapsto n\} \cup D\restriction_n$ is in $[n-1, n+1]$; equivalently, this can be written as $X_{n-1}(n-1) \cup X_n(n) \cup X_{n+1}(n+1) \models \Pi\{t \mapsto n\} \cup U_n(n)$, which implies the claim.

Next, suppose $n > \tau_{\max}$. Then $\sigma_n = \varepsilon$, $a_{n-1} \in \{2, 3\}$, $a_n = 3$, $c_n = c_{n-1} + 1$, and $c_n \in [1, N]$, as required. Furthermore, the justification of Condition (2.3) above also proves Condition (3.2) since $D\restriction_n = \emptyset$. This concludes the proof of Claim 5.19.

Let now $w = \sigma_0 \dots \sigma_n$ be a word in $\mathsf{notimplies}(\Pi, \mathcal{I})$. We show that automaton $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ accepts w. By definition, there is an integer $i \in [0, n]$, a predicate $P \in \mathcal{I}$, and sets $E_0, \dots, E_n$ of EDB predicates in Π such that w has the form $\langle i, P, E_0, \dots, E_n \rangle$ where $\Pi \cup D \not\models P(i)$ for $D = \{A(j) \mid A \in E_j, 0 \leq j \leq n\}$.

Thus, there is a model M of $\Pi \cup D$ that does not contain $P(i)$. Then, by the claim, $\rho = s_{-1}\sigma_0 s_0 \dots \sigma_n s_n \varepsilon s_{n+1} \dots \varepsilon s_{n+N}$ is a run of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$. Furthermore, by definition, s_{n+N} has the form $\langle 3, X, Y, N\rangle$, and thus ρ is an accepting run of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ on w.

We next show that $\mathsf{notimplies}(\Pi, \mathcal{I})$ contains each word accepted by automaton $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$. Let $w = \sigma_0, \dots, \sigma_n$ be a word accepted by $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$. Then there exists a run $\rho = s_{-1}\sigma_0 s_0 \dots \sigma_{n+N} s_{n+N}$ of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ where s_{n+N} is final, and $s_j = \langle a_j, Y_j, Z_j, c_j\rangle$ for $j \in [0, n+N]$. Since s_{n+N} is final, by the definition of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$, there exists $k \in [0, n]$ such that $\sigma_k = \langle U_k, P\rangle$ with $P \in \mathcal{I}$, $\sigma_j = \langle U_j, \square\rangle$ for each $j \in [0, n]$ with $j \neq k$, and $\sigma_j = \varepsilon$ for each $j \in [n+1, n+N]$. It follows that $w = \langle k, P, U_0, \dots, U_n\rangle$. Thus, it remains to show that $\Pi \cup \bigcup_{j=0}^{n} U_j(j) \not\models P(k)$, which we do next by constructing a model M of $\Pi \cup \bigcup_{j=0}^{n} U_j(j)$ such that $P(k) \notin M$. To this end, we first show the following claim.

Claim 5.20. *There exist integers $p \in [0, N]$ and $T \in [1, N-p]$ such that, for each $i \geq 1$, $\rho_i = s_{-1}\sigma_0 \dots \sigma_{n+p} s_{n+p} \varepsilon s'_1 \dots \varepsilon s'_i$ is an accepting run of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I}), p+i]$, where $s'_j = \langle 3, Y_{n+p+(j \bmod T)}, Y_{n+p+(j+1 \bmod T)}, p+j\rangle$ for each $j \in [1, i]$.*

Since ρ is a run of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$, there exist integers j_1 and j_2 in $[n, n+N]$ with $j_1 < j_2$ such that $Y_{j_1} = Y_{j_2}$ and $Z_{j_1} = Z_{j_2}$; indeed, $\big|[n, n+N]\big| = N+1$ while there are at most N distinct pairs $\langle Y, Z\rangle$ such that, for $a \in \{2, 3\}$ and some integer c, $\langle a, Y, Z, c\rangle$ is a state of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$. Let $p = j_1 - n$ and $T = j_2 - j_1$. Clearly, $p \in [0, N]$ and $T \in [1, N-p]$, as required. For the rest, let $\mathcal{A}_i = \mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I}), p+i]$ for each $i \geq 0$. For $1 \leq i \leq T$, ρ_i is an accepting run of $\mathcal{A}_i$ as it is a prefix of ρ and as $s'_i = s_{n+p+i}$ is final in $\mathcal{A}_i$. So, without loss of generality, let $i > T$. Let us refer to s_{n+p} as s'_0. Clearly, s'_i is again final in $\mathcal{A}_i$ so it suffices to show that, for each $j \in [T, i]$, $s'_{j-1} \xrightarrow{\varepsilon} s'_j$ is a transition of $\mathcal{A}_i$. This follows as $s'_{j-1 \bmod T} \xrightarrow{\varepsilon} s'_{j \bmod T}$ is a transition of $\mathcal{A}_i$, s'_{j-1} coincides with $s'_{j-1 \bmod T}$ in its second and third component, s'_j coincides with $s'_{j \bmod T}$ in its second and third component, and the first and last components of s'_{j-1} and s'_j are as required by ε-transitions of $\mathcal{A}_i$. This concludes the proof of Claim 5.20.

Let now p and T be as asserted by the claim. We define

$$M = \bigcup_{j=0}^{n+p} Y_j(j) \cup \bigcup_{j \geq 1} Y_{n+p+(j \bmod T)}(n+p+j)$$

We conclude the proof by showing that M is as required. First, we have $P(k) \notin M$ by Condition (2.1). Second, we have $\bigcup_{j=0}^{n} U_j(j) \subseteq \bigcup_{j=0}^{n} Y_j(j) \subseteq M$, where the first containment holds by Condition (2.3). Third, we show $M \models r$ for an arbitrary rule $r \in \Pi$. Since Π is normal, it suffices to show, for each $i \geq 0$, $M\restriction_{[i-1,i+1]} \models r\{t \mapsto i\}$. Let $k_j = j$ for $j \leq n+p$, and $k_j = n+p+(j-n-p \bmod T)$ for $j \geq n+p+1$. Then, $M\restriction_{[i-1,i+1]} \models r\{t \mapsto i\}$ can be restated as $Y_{k_{i-1}}(i-1) \cup Y_{k_i}(i) \cup Y_{k_{i+1}}(i+1) \models r\{t \mapsto i\}$, which holds by Claim 5.20 together with Conditions (2.3) and (3.2). □

Automaton $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ can be computed in polynomial space similarly to $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$.

Theorem 5.21. *Automaton* $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ *can be computed in polynomial space with respect to the size of* Π *for each* $\mathcal{I}$*; furthermore, the size of each input letter, each state, and each transition of* $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ *is polynomial in the size of* Π*.*

Proof. Let Π be an object-free normal program and $\mathcal{I}$ a set of IDB predicates in Π. Note that the size of $\mathcal{I}$ is bounded by that of Π, and hence can be disregarded. It is clear that the size of each input letter, each state, and each transition of $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ is polynomial in the size of Π. Automaton $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$ can be computed in polynomial space with the respect to the size of Π through a 'generate and filter' algorithm similar to the one described in the proof of Theorem 5.16. Note that checking Conditions 2.3 and 3.2 amounts to model checking in propositional logic, which can be done in polynomial time. □

Finally, we define an automaton $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ recognising $\mathsf{delay}(\Pi, \mathcal{I})$ for any object-free normal program Π and set $\mathcal{I}$ of IDB predicates in Π. To this end, we first define an automaton $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ for the language $\mathsf{prefimplies}(\Pi, \mathcal{I})$ and show that it can be computed in polynomial space with respect to the size of Π. We

then use the automaton $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ to define $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$, and show that $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ can also be computed in polynomial space.

Definition 5.22. *Let* Π *be an object-free normal program, let* $\mathcal{I}$ *be a set of IDB predicates in* Π*, and let* $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})] = \langle \Sigma, S, S_{\mathrm{I}}, F, \Delta \rangle$*. We define the automaton* $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ *as* $\langle \Sigma, S, S_{\mathrm{I}}, F', \Delta \rangle$*, where* F' *is the set of all states occurring in an accepting run of* $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$*.*

Automaton $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ captures the prefix language of $\mathsf{implies}(\Pi, \mathcal{I})$ because its accepting runs can be completed into accepting runs of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ by construction.

Proposition 5.23. *Let* Π *be an object-free normal program and* $\mathcal{I}$ *a set of IDB predicates in* Π*. Automaton* $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ *recognises* $\mathsf{prefimplies}(\Pi, \mathcal{I})$*.*

Some of the PSpace algorithms given below make use of the following lemma, which relies on an adaptation of the well-known NL algorithm for graph reachability—described in, e.g., Example 2.10 of [42]—to the case where the input graph is concisely given as a function that can be computed in polynomial space.

Lemma 5.24. *Let* S *be a set and* g *a total function from* S *to labelled directed graphs computable in polynomial space and such that, for each* $x \in S$*, the size of each node and each edge of* $g(x)$ *is polynomial in the size of* x*. The problem of checking, given some* $x \in S$ *and two nodes* u, v *of* $g(x)$*, whether* v *is reachable from* u *in* $g(x)$ *can be solved in polynomial space.*

Proof. Note that, since PSpace = NPSpace, it suffices to give a non-deterministic algorithm, which we do next. Given $x \in S$ and nodes u, v in $g(x)$, we first establish the number n of nodes in $g(x)$ by enumerating and counting them. We then initialise a counter i to 0 and store u in a variable p. While $i < n$, we repeat the following loop. We check if $p = v$, in which case we accept. Otherwise, we guess an integer $j \in [1, m]$ and enumerate the edges of $g(x)$, picking the j-th edge $\langle a, \sigma, b \rangle$. If $a \neq p$, we reject. Otherwise we increment i by one, set p to b, and repeat the loop. If the loop terminates with $i = n$, we reject.

The correctness of the algorithm is immediate since v is reachable from u in $g(x)$ if and only if it is reachable by a path of length at most $n-1$. Finally, we argue that the algorithm runs in polynomial space with respect to the size of x. By assumption, we can enumerate each node and each edge of $g(x)$ in polynomial space, and each such node and edge is of polynomial size. It follows that polynomial memory suffices to represent p, a, σ, and b. It also follows that the number of nodes and edges is at most exponential, and hence polynomial memory suffices for the counters i and j. □

Lemma 5.24 implies that polynomial space suffices to check reachability in the graph induced by any automaton that can be computed in polynomial space. This makes it easy to compute $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ out of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ by extending its set of final states. The following lemma is then immediate.

Lemma 5.25. *Automaton* $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ *can be computed in polynomial space with respect to the size of* Π *for each* $\mathcal{I}$*; furthermore, the size of each input letter, each state, and each transition of* $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ *is polynomial in the size of* Π*.*

Proof. The second claim follows by Theorem 5.16 as the input alphabet, the states, and the transitions of $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ coincide with those of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$. It also follows that we can compute all the components of $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ in polynomial space except for the set of final states. It thus remains to show how to compute the set of final states of $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$. To do so, we enumerate all triples $\langle s, s', s'' \rangle$ of states of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$ where s is initial and s'' is final; for each such triple, we check whether s' is reachable from s and whether s'' is reachable from s' in the labelled directed graph induced by the transitions of $\mathcal{A}[\mathsf{implies}(\Pi, \mathcal{I})]$; if both checks succeed, we add s' to the set of final states of $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$, and otherwise we ignore the triple. This can be done in polynomial space with respect to the size of Π by Lemma 5.24 and Theorem 5.16. □

The automaton for the delay language can now be defined starting from the automata we have defined above.

Definition 5.26. *Let* Π *be an object-free normal program, let* $\mathcal{I}$ *be a subset of the IDB predicates in* Π*, and let* $\mathcal{A} = \langle \Sigma, S, S_{\mathrm{I}}, F, \Delta \rangle$ *be the product automaton* $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})] \times \mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$*. We define* $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ *as the automaton* $\langle \Sigma, S \cup \{s_{\mathrm{I}}\}, \{s_{\mathrm{I}}\}, F, \Delta \cup \Delta' \rangle$ *where* s_{I} *is a fresh state and* Δ' *consists of each transition* $s_{\mathrm{I}} \xrightarrow{\varepsilon} s$ *of* Δ *such that* $\mathcal{A}$ *admits a run of the form* $s_0 \sigma_1 s_1 \dots \sigma_n s_n \sigma s$ *with* $s_0 \in S_{\mathrm{I}}$ *and* σ *an answer letter.*

The construction of $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$, apart from capturing the intersection of two languages by means of the well-known cross-product construction, captures suffices similarly to the way automaton $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ captures prefixes. One difference is that here we are interested in specific suffices—the ones starting right after an answer letter.

Proposition 5.27. *Let* Π *be an object-free normal program and let* $\mathcal{I}$ *be a set of IDB predicates in* Π*. Automaton* $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ *recognises* $\mathsf{delay}(\Pi, \mathcal{I})$*.*

Theorem 5.28. *Automaton* $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ *can be computed in polynomial space with respect to the size of* Π *for each* $\mathcal{I}$*; furthermore, the size of each input letter, each state, and each transition of* $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ *is polynomial in the size of* Π*.*

Proof. Let Π be an object-free normal program, $\mathcal{I}$ a set of IDB predicates in Π, and let Σ, S, S_{I}, s_{I}, F, Δ, and Δ' be as in Definition 5.26. The second claim follows by construction from Theorem 5.21 and Lemma 5.25. Furthermore, since Σ, S, S_{I}, s_{I}, F, and Δ are polynomially constructed from the corresponding sets in $\mathcal{A}[\mathsf{prefimplies}(\Pi, \mathcal{I})]$ and $\mathcal{A}[\mathsf{notimplies}(\Pi, \mathcal{I})]$, Theorem 5.21 and Lemma 5.25 imply that their construction takes polynomial space in the size of Π. Thus, it remains to show that Δ' can be computed in polynomial space. To do so, we enumerate the transitions in Δ and, for each such transition $s \xrightarrow{\sigma} s'$, output the transition $s_{\mathrm{I}} \xrightarrow{\varepsilon} s'$ if σ is an answer letter and s is reachable from an initial state of $\mathcal{A}$ in the labelled directed graph induced by Δ; the reachability check can be performed in polynomial space with respect to the size of Π by Lemma 5.24. □

5.3.3 Complexity Upper Bounds

In this section, we give complexity upper bounds assuming that all the EDB streams (and sets of EDB facts) are over the same finite object domain. In the next section, we will give matching lower bounds. We give bounds for the decision problems of checking delay existence, delay validity, program containment, and window size validity. We conclude the section by showing that the former upper bounds can be combined into an upper bound for the problem of rejecting if no valid delay exists and computing a minimum valid delay and window size otherwise.

We will always proceed by first showing an upper bound for object-free programs, and then lifting the result to arbitrary programs by arguing that programs can be grounded over the considered object domain.

We first give upper bounds for delay existence and validity. In a nutshell, since the length n of a word in a delay language witnesses the non-validity of n as a delay, we can check: (i) *delay existence* by checking whether words in the language are of bounded length, hence whether the language is finite, and hence whether the corresponding automaton has no cycle such that it contains at least one non-ε-transition, it is reachable from the initial state and, it can reach a final state; (ii) *validity of a delay d* by checking whether words in the language have length at most $d-1$, and hence whether the corresponding automaton contains no path from the initial state to a final state that contains at least $d-1$ non-ε-transition. The proof of the theorem also involves normalisation, since our automata are for normal programs.

Theorem 5.29. *Delay existence and delay validity are in* PSPACE *if both the object domain of streams and the arity of predicates in programs are considered fixed (hence when the problems are restricted to object-free programs). Both problems are in* EXPSPACE *if the object domain of streams is considered fixed (and the arity of predicates is arbitrary).*

Proof. We argue below the PSPACE upper bounds for delay existence and delay validity, separately for the two problems, and only for the case when they are restricted

to object-free programs. The upper bounds for the remaining cases will then follow immediately since each program can be transformed to an object-free program by grounding with respect to the fixed object domain in exponential time in general (and in polynomial time if the maximum arity of a predicate is fixed); clearly, the original program admits a delay if and only so does its grounding.

In the following, let Π be an object-free program, and let $\mathcal{I}$ be the set of IDB predicates in Π.

Delay Existence. By Theorem 5.6, a program Π admits a valid delay if and only if $\mathsf{delay}(\Pi, \mathcal{I})$ is finite. Furthermore, by Theorem 5.9, $\mathsf{delay}(\Pi, \mathcal{I}) = \mathsf{delay}(\Xi(\Pi), \mathcal{I})$, and hence, by Proposition 5.27, the automaton $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ recognises $\mathsf{delay}(\Pi, \mathcal{I})$, and $\mathsf{delay}(\Pi, \mathcal{I})$ is finite iff each cycle of each accepting run of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ is over ε-transitions only. It therefore suffices to provide a non-deterministic polynomial space algorithm that takes Π as input and accepts if and only if there is an accepting run of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ having a cycle that contains a non-ε transition. The algorithm starts by computing $\Xi(\Pi)$, which takes polynomial time and hence polynomial space. It then computes the initial state s_0 of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$, after which it guesses a pair of states $\langle s, s' \rangle$, and accepts if s and s' are states in $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ such that s' is a final state, s is reachable from s_0, s' is reachable from s, and s is reachable from s through a path containing at least one edge whose label is different from ε. The algorithm can be realised in polynomial space. Indeed, by Theorem 5.28, we can check in polynomial space whether a state belongs to the automaton and whether it is a final state; furthermore, each of the reachability checks can be performed in polynomial space in the combined size of Π, s_0, s, and s' by Lemma 5.24, and hence in polynomial space in the size of Π.

Delay Validity. Let d be a non-negative integer. By Theorem 5.6, d is a valid delay for Π if and only if each word in $\mathsf{delay}(\Pi, \mathcal{I})$ is of length at most $d-1$. Since automaton $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ recognises $\mathsf{delay}(\Pi, \mathcal{I})$ by Proposition 5.27 and Theorem 5.9, d is a valid delay for Π if and only if each accepting run of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ consists of at most $d-1$ non-ε transitions. To show the PSPACE upper bound, it therefore suffices to

provide a polynomial space algorithm that takes Π and d as input and accepts iff there is an accepting run of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ consisting of at least d non-ε transitions. The algorithm computes $\Xi(\Pi)$ and the initial state s_0 of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$; then, it guesses a state s and accepts if s is a final state of $\mathcal{A}[\mathsf{delay}(\Xi(\Pi), \mathcal{I})]$ and it is reachable from s_0 via a path containing at least d non-ε edges. The aforementioned reachability check is feasible in polynomial space in the combined size of Π, s_0, s, and d by a straightforward generalisation of Lemma 5.24, and hence in space polynomial in the combined size of Π and d, since s_0 and s are of size polynomial in the size of Π by Theorem 5.28. □

We show upper bounds for program containment as an intermediate result, which we will use next to show upper bounds for window size validity. We decide the containment $\Pi_1 \sqsubseteq \Pi_2$ by checking for emptiness of the intersection of the $\mathsf{implies}$ language for Π_1 with the $\mathsf{notimplies}$ language for Π_2. As in the previous theorem, the proof also involves a normalisation step, since our automata are for normal programs.

Theorem 5.30. *Program containment is in* PSPACE *if both the object domain of streams and the arity of predicates in programs are considered fixed (hence when the problem is restricted to object-free programs). The problem is in* EXPSPACE *if the object domain of streams is considered fixed (and the arity of predicates is arbitrary).*

Proof. We first show the PSPACE upper bound for program containment restricted to object-free programs, and then derive the remaining upper bounds via grounding.

Let Π_1 and Π_2 be object-free programs with the same set of EDB predicates. By Theorem 5.6, $\Pi_1 \sqsubseteq \Pi_2$ if and only if $\mathsf{implies}(\Pi_1, \mathcal{I}) \cap \mathsf{notimplies}(\Pi_2, \mathcal{I}) = \emptyset$ with $\mathcal{I}$ the set of IDB predicates in Π_1. Furthermore automaton $\mathcal{A}[\mathsf{implies}(\Xi(\Pi_1), \mathcal{I})]$ recognises $\mathsf{implies}(\Pi_1, \mathcal{I})$ by Theorems 5.9 and 5.11, and automaton $\mathcal{A}[\mathsf{notimplies}(\Xi(\Pi_2), \mathcal{I})]$ recognises $\mathsf{notimplies}(\Pi_2, \mathcal{I})$ by Theorems 5.9 and 5.18. Let $\mathcal{A}$ be the product automaton $\mathcal{A}[\mathsf{implies}(\Xi(\Pi_1), \mathcal{I})] \times \mathcal{A}[\mathsf{notimplies}(\Xi(\Pi_2), \mathcal{I})]$. We have that $\mathcal{A}$ recognises $\mathsf{implies}(\Pi_1, \mathcal{I}) \cap \mathsf{notimplies}(\Pi_2, \mathcal{I})$, and hence $\mathsf{implies}(\Pi_1, \mathcal{I}) \cap \mathsf{notimplies}(\Pi_2, \mathcal{I}) = \emptyset$ if

and only if $\mathcal{A}$ has no accepting run. Summing up, we have that $\Pi_1 \sqsubseteq \Pi_2$ if and only if $\mathcal{A}$ has no accepting run. Clearly, checking this is feasible in PSPACE since reachability between an initial state and a final state can be checked in space polynomial in the size of Π.

The remaining upper bounds follow directly since programs Π_1 and Π_2 can be transformed into object-free programs via grounding with respect to the fixed object domain in exponential time in general (and in polynomial time if the maximum arity of a predicate is fixed); furthermore, grounding does not affect the result of containment. □

The upper bound proof for window size validity amounts to modelling the definition of valid window size as a containment $\Pi_1 \sqsubseteq \Pi_2$, where Π_1 and Π_2 capture the left-hand and right-hand side, respectively, of the inclusion in Definition 3.10.

Theorem 5.31. *Window size validity is in* PSPACE *if both the object domain of streams and the arity of predicates in programs are considered fixed (hence when the problem is restricted to object-free programs). The problem is in* EXPSPACE *if the object domain of streams is considered fixed (and the arity of predicates is arbitrary).*

Proof. We show below the PSPACE upper bound for the problem restricted to object-free programs. The remaining upper bounds will then follow via a grounding argument as in Theorems 5.29 and 5.30.

To establish the PSPACE membership, we provide a log-space many-one reduction from window size validity restricted to object-free programs to containment of object-free programs, which is in PSPACE by Theorem 5.30. Our reduction φ maps an instance $\langle \Pi, d, w \rangle$ of window size validity, with Π object-free, to a pair $\langle \Pi_1, \Pi_2 \rangle$ of object-free programs (defined below) if $w < d + \rho$, with ρ be the maximum forward radius of Π, and to $\langle \emptyset, \emptyset \rangle$ otherwise.

We next show how programs Π_1 and Π_2 in our reduction φ are constructed from Π, d and w. Let k be the number of bits required to encode d. For each $i \in [1, k]$, let Zero_i and One_i be fresh unary predicates—intuitively representing the i-th bit of a

number. Furthermore, for each $i \in [1, k]$, let A_i be Zero_i if the i-th bit of the binary encoding of d is one and let it be One_i otherwise—i.e., the A_i's encode the bitwise complement of d. Finally, let *Now*, *Continue*, *Flip*, *NoFlip* be fresh unary predicates. We define Π_{count} as the program consisting of Rules (5.12)–(5.19).[2]

$$\mathit{Now}(t) \rightarrow \mathit{Zero}_i(t) \qquad \forall i \in [1, k] \tag{5.12}$$

$$\bigwedge_{\ell=1}^{i-1} \mathit{One}_\ell(t) \wedge \mathit{Zero}_i(t) \rightarrow \mathit{Flip}_j(t) \qquad \forall i \in [1, k],\ \forall j \in [1, i] \tag{5.13}$$

$$\bigwedge_{\ell=1}^{i-1} \mathit{One}_\ell(t) \wedge \mathit{Zero}_i(t) \rightarrow \mathit{NoFlip}_j(t) \qquad \forall i \in [1, k],\ \forall j \in (i, k] \tag{5.14}$$

$$A_i(t) \rightarrow \mathit{Continue}(t) \quad \forall i \in [1, k] \tag{5.15}$$

$$\mathit{Zero}_i(t) \wedge \mathit{Flip}_i(t) \wedge \mathit{Continue}(t) \rightarrow \mathit{One}_i(t-1) \quad \forall i \in [1, k] \tag{5.16}$$

$$\mathit{One}_i(t) \wedge \mathit{Flip}_i(t) \wedge \mathit{Continue}(t) \rightarrow \mathit{Zero}_i(t-1) \quad \forall i \in [1, k] \tag{5.17}$$

$$\mathit{Zero}_i(t) \wedge \mathit{NoFlip}_i(t) \wedge \mathit{Continue}(t) \rightarrow \mathit{Zero}_i(t-1) \quad \forall i \in [1, k] \tag{5.18}$$

$$\mathit{One}_i(t) \wedge \mathit{NoFlip}_i(t) \wedge \mathit{Continue}(t) \rightarrow \mathit{One}_i(t-1) \quad \forall i \in [1, k] \tag{5.19}$$

The construction of Π_{count} ensures that, for each EDB stream S containing at most one fact about *Now*, each time point τ, each $m \in [0, d]$, and for predicates $B_1, \ldots, B_k$ where B_i is Zero_i if the i-th bit in the binary encoding of m is zero and One_i otherwise,

$$\{B_0(\tau), \ldots, B_k(\tau)\} \subseteq \Pi_{\text{count}}(S) \text{ if and only if } \mathit{Now}(\tau + m) \in S.$$

We next define $\Pi_{\text{intervals}}$ as the extension of Π_{count} with Rules (5.20)–(5.29), where $[\![\mathit{now} - d]\!]$, $[\![\mathit{now} - w, \mathit{now}]\!]$, $[\![\mathit{now} - w, \mathit{now} - d - 1]\!]$, $[\![\mathit{now} - w, \infty]\!]$, $[\![0, \mathit{now}]\!]$ are fresh unary predicates intuitively standing for time intervals; furthermore, for each $i \in [1, k]$, Bit_i is a fresh unary predicate, and C_i is Zero_i if the i-th bit of the binary

[2]Program Π_{count} is a close variant of the program given in Example 3.14, with the addition of the *Continue* predicate to control how far the program derives facts into the past.

encoding of d is zero and One_i otherwise—i.e., the C_i's encode d.

$$C_1(t) \wedge \cdots \wedge C_k(t) \rightarrow [\![now - d]\!](t) \tag{5.20}$$

$$Zero_i(t) \rightarrow Bit_i(t) \qquad \forall i \in [1, k] \tag{5.21}$$

$$One_i(t) \rightarrow Bit_i(t) \qquad \forall i \in [1, k] \tag{5.22}$$

$$Bit_1(t) \wedge \cdots \wedge Bit_k(t) \rightarrow [\![now - w, now]\!](t) \tag{5.23}$$

$$[\![now - d]\!](t) \rightarrow [\![now - w, now]\!](t - \ell) \qquad \forall \ell \in [1, w - d] \tag{5.24}$$

$$[\![now - d]\!](t) \rightarrow [\![now - w, now - d - 1]\!](t - \ell) \qquad \forall \ell \in [1, w - d] \tag{5.25}$$

$$[\![now - d]\!](t) \rightarrow [\![now - w, \infty]\!](t - w + d) \tag{5.26}$$

$$[\![now - w, \infty]\!](t) \rightarrow [\![now - w, \infty]\!](t + 1) \tag{5.27}$$

$$Now(t) \rightarrow [\![0, now]\!](t) \tag{5.28}$$

$$[\![0, now]\!](t) \rightarrow [\![0, now]\!](t - 1) \tag{5.29}$$

Intuitively, the rules above populate time intervals with the relevant time points. Specifically, we can see that, for each EDB stream S containing at most one fact about Now and each time point τ, we have $Now(\tau) \in S$ if and only if all the following points hold:

1. $[\![now - d]\!](\tau - d) \in \Pi_{\text{intervals}}(S)$;
2. $[\![now - w, now]\!](\tau') \in \Pi_{\text{intervals}}(S)$ for each $\tau' \in [\tau - w, \tau]$;
3. $[\![0, now]\!](\tau') \in \Pi_{\text{intervals}}(S)$ for each $\tau' \in [0, \tau]$;
4. $[\![now - w, now - d - 1]\!](\tau') \in \Pi_{\text{intervals}}(S)$ for each $\tau' \in [\tau - w, \tau - d - 1]$;
5. $[\![now - w, \infty]\!](\tau') \in \Pi_{\text{intervals}}(S)$ for each $\tau' \in [\tau - w, \infty)$.

Let Π' and Π'' be the programs obtained from Π by renaming each predicate P to fresh predicates P' and P'', respectively. Let Π''_{R} be the program consisting of a rule of the form

$$[\![now - w, \infty]\!](u) \wedge \bigwedge_i [\![now - w, \infty]\!](u_i) \wedge P''_i(\mathbf{s}_i, u_i) \rightarrow P''(\mathbf{s}, u) \tag{5.30}$$

for each rule in Π'' of the form $\bigwedge_i P_i''(\mathbf{s}_i, u_i) \to P''(\mathbf{s}, u)$. Intuitively, Π''_{R} restricts the rules of Π'' to those time points for which $[\![now - w, \infty]\!]$ holds.

Let Π_{aux} be $\Pi' \cup \Pi''_{\mathrm{R}} \cup \Pi_{\mathrm{intervals}}$ extended with a rule of the form (5.31) for each EDB predicate A occurring in Π, a rule of the form (5.32) for each EDB predicate A occurring in Π, and a rule of the form (5.33) for each predicate P occurring in Π.

$$[\![0, now]\!](t) \wedge A(t) \to A'(t) \tag{5.31}$$

$$[\![now - w, now]\!](t) \wedge A(t) \to A''(t) \tag{5.32}$$

$$[\![now - w, now - d - 1]\!](t) \wedge P'(t) \to P''(t) \tag{5.33}$$

By construction, Π_{aux} satisfies the following properties for each EDB stream S containing at most one fact about *Now*, each pair of time points τ and τ', and each predicate P:

1. $P(\tau') \in \Pi(S\restriction_{[0,\tau]})$ and $Now(\tau) \in S$ if and only if $P'(\tau') \in \Pi_{\mathrm{aux}}(S)$;
2. $P(\tau') \in \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)$ and $Now(\tau) \in S$ if and only if $P''(\tau') \in \Pi_{\mathrm{aux}}(S)$.

Let Π_{flood} be the program consisting of Rules (5.34)–(5.38) together with a rule of the form (5.39) for each IDB predicate P occurring in Π, where $[\![now + 1, \infty]\!]$ and *Flood* are fresh unary predicates.

$$Now(t) \to [\![now + 1, \infty]\!](t + 1) \tag{5.34}$$

$$[\![now + 1, \infty]\!](t) \to [\![now + 1, \infty]\!](t + 1) \tag{5.35}$$

$$[\![now + 1, \infty]\!](t) \wedge Now(t) \to Flood(t) \tag{5.36}$$

$$Flood(t) \to Flood(t + 1) \tag{5.37}$$

$$Flood(t) \to Flood(t - 1) \tag{5.38}$$

$$Flood(t) \to P''(t) \tag{5.39}$$

Clearly, for each stream S that contains two facts about *Now*, $\Pi_{\mathrm{flood}} \cup S$ entails each fact about P'' for P a predicate in Π. Finally, program Π_1 (respectively, Π_2) is defined

as the extension of $\Pi_{\text{aux}} \cup \Pi_{\text{flood}}$ with a rule of the form (5.40) (respectively, of the form (5.41)) for each predicate P occurring in Π, where Goal_P is a fresh unary predicate.

$$[\![\mathit{now} - d]\!](t) \wedge P'(t) \rightarrow \mathit{Goal}_P(t) \tag{5.40}$$

$$[\![\mathit{now} - d]\!](t) \wedge P''(t) \rightarrow \mathit{Goal}_P(t) \tag{5.41}$$

The construction of these programs ensures that the following statements hold for each EDB stream S containing at most one fact about Now, each time point τ, and each predicate P:

1. $P(\tau - d) \in \Pi(S\restriction_{[0,\tau]})$ and $\mathit{Now}(\tau) \in S$ if and only if $\mathit{Goal}_P(\tau - d) \in \Pi_1(S)$;
2. $P(\tau - d) \in \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big)$ and $\mathit{Now}(\tau) \in S$ if and only if $\mathit{Goal}_P(\tau - d) \in \Pi_2(S)$.

This completes the description of our reduction φ. Clearly, φ can be computed in logarithmic space: k is linear in the size of d, and $\Pi_{\text{intervals}}$ contains a polynomial number of Rules (5.24) and (5.25) when $w < d + \rho$ (specifically, $w - d < \rho$ rules of each type, where ρ is linear in the size of Π).

We conclude by arguing correctness of φ. If $w \geq d + \rho$ then, by Theorem 4.5, w is a valid window size for Π and d and, as required, the containment $\emptyset \sqsubseteq \emptyset$ holds trivially. Let us now consider the case where $w < d + \rho$.

Assume $\Pi_1 \sqsubseteq \Pi_2$; we show that w is a valid window size for Π and d. Let S be an EDB stream, let τ be a time point with $\tau \geq d$, let S' be $S \cup \{\mathit{Now}(\tau)\}$, and let α be a fact with time argument $\tau - d$. We assume w.l.o.g. that S contains no fact about Now, and hence S' contains one fact about Now. Assume that $\alpha \in \Pi(S\restriction_{[0,\tau]})$ with α of the form $P(\tau - d)$. It suffices to show

$$\alpha \in \Pi\restriction_{[\tau-w,\infty)}\Big(\Pi\big(S\restriction_{[0,\tau]}\big)\restriction_{[\tau-w,\tau-d-1]} \cup S\restriction_{[\tau-w,\tau]}\Big). \tag{5.42}$$

By monotonicity, we have $P(\tau - d) \in \Pi(S'\restriction_{[0,\tau]})$, and hence $\mathit{Goal}_P(\tau - d) \in \Pi_1(S')$ by the aforementioned property of Π_1. It follows that $\mathit{Goal}_P(\tau - d) \in \Pi_2(S')$ since $\Pi_1 \sqsubseteq \Pi_2$. But then, (5.42) holds by the properties of Π_2—note that we can replace S' with S because Now does not occur in Π.

Finally, assume that w is a valid window size for Π and d, and that $\alpha \in \Pi_1(S)$ for S an EDB stream and α a fact. It suffices to show $\alpha \in \Pi_2(S)$. If α is not a fact about any $Goal_P$, then $\alpha \in \Pi_{\text{aux}}(S)$ and hence $\alpha \in \Pi_2(S)$ since $\Pi_{\text{aux}} \subseteq \Pi_2$. Now assume that α is of the form $Goal_P(\tau)$. If S contains two facts about Now, then $P''(\tau) \in \Pi_2(S)$ by the properties of Π_{flood}, and hence $Goal_P(\tau) \in \Pi_2(S)$ by Rule (5.41); so suppose that S contains at most one fact about Now. Then, $P(\tau) \in \Pi(S\restriction_{[0,\tau+d]})$ and $Now(\tau+d) \in S$ by the properties of our construction stated above. Furthermore, it holds that

$$P(\tau) \in \Pi\restriction_{[\tau+d-w,\infty)}\Big(\Pi\big(S\restriction_{[0,\tau+d]}\big)\restriction_{[\tau+d-w,\tau-1]} \cup S\restriction_{[\tau+d-w,\tau+d]}\Big)$$

since w is a valid window size for Π and d. As a result, $Goal_P(\tau) \in \Pi_2(S)$ by the properties of Π_2, as required. □

We conclude by giving upper bounds for the problem of computing a minimum valid delay and window size. The proof of the theorem combines the previous results in a straightforward manner.

Theorem 5.32. *There exists an algorithm running in polynomial space that takes as input an object-free program* Π *and computes the smallest valid delay* d *for* Π *and the smallest valid window size for* Π *and* d *whenever* Π *admits a valid delay, or rejects* Π *if it does not admit a valid delay.*

Proof. On input Π, the algorithm performs the following steps:

1. It checks whether Π admits a valid delay, and it rejects the input if it does not.
2. It computes the smallest valid delay d by iteratively incrementing d from 0 to the number k of states in the automaton $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$, and stopping as soon as the the resulting value is a valid delay for Π.
3. It computes the smallest valid window size w by iteratively incrementing w from 0 to $d + \rho$, with ρ the maximum forward radius of a rule in Π, and stopping as soon as the the resulting value is a valid window size for Π and d.
4. It outputs d and w.

Correctness of the algorithm follows directly from the following observations. Recall that if Π admits a valid delay then accepting runs cannot involve a cycle over a non-ε transition, as argued in the proof of Theorem 5.29. As a result, if Π admits a valid delay d, then d cannot be larger than the number k of states of the automaton $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$, which justifies Step 2. Furthermore, by Theorem 4.5, if d is a valid delay then $d + \rho$ is a valid window size, which justifies Step 3.

We now argue that the algorithm can be realised in polynomial space. The first step is clearly feasible in polynomial space by Theorem 5.29. For Step 2, note that the number of states in $\mathcal{A}[\mathsf{delay}(\Pi, \mathcal{I})]$ is exponentially bounded in the maximum size of a state. As a result, we can do no more than exponentially many delay validity calls in Step 2, each of which feasible in polynomial space in Π and the tested value d (which can be stored in binary using only polynomial space in the size of Π). Finally, Step 3 involves no more than $d+\rho$ window size validity tests, each of which also takes polynomial space (again, note that each tested w requires only polynomial space in Π). □

The theorem can be trivially lifted using grounding to arbitrary programs when considering a fixed object domain.

Corollary 5.33. *There exists an algorithm running in exponential space that, for a fixed object domain for streams, takes as input a program* Π *and computes the smallest valid delay* d *for* Π *and the smallest valid window size for* Π *and* d *whenever* Π *admits a valid delay, or rejects* Π *if it does not admit a valid delay; furthermore, such an algorithm runs in polynomial space if the maximum arity of a predicate in* Π *is fixed.*

5.4 Complexity Lower Bounds

We next show that the upper bounds established in the previous section are tight. For this, we first establish complexity lower bounds for containment of forward-propagating programs with respect to a fixed object domain as well as containment of object-free forward-propagating programs, showing that the former problem is

ExpSpace-hard while the latter is PSpace-hard. Note that fixing an object domain makes no difference for object-free programs, and also that lower bounds for object-free programs immediately imply lower bounds for programs with a fixed arity with respect to a fixed object domain—since the maximum arity in an object-free program is zero. We will then combine the former results with the reductions from program containment to delay existence, delay validity, and window size validity developed in Section 5.2, concluding that, over a fixed object domain, all three problems are ExpSpace-hard for unrestricted programs, and PSpace-hard for programs where predicates are of bounded arity (and hence for object-free programs).

Theorem 5.34. *Containment of forward-propagating programs with respect to a fixed object domain is* ExpSpace-hard*; the problem becomes* PSpace-hard *if additionally restricted to forward-propagating programs where predicates have a fixed maximum arity (in particular, to object-free forward-propagating programs).*

Proof. We first show that containment of forward-propagating programs with respect to a fixed object domain is ExpSpace-hard, by providing a reduction from containment of succinct regular expressions (SREs) to containment of forward-propagating programs. The language of SREs extends regular expressions with the exponentiation operator, which allows one to write expressions of the form R^k where R is an SRE and k is a non-negative integer; the expression R^k matches any concatenation of k words matched by R. The containment problem for SREs, which assumes exponents to be coded in binary, is known to be ExpSpace-complete [47]. Formally, we consider SREs over a given alphabet Σ generated by the grammar

$$R ::= \emptyset \mid \varepsilon \mid \sigma \mid R \cup R \mid R \circ R \mid R^+ \mid R^k$$

where σ ranges over Σ and $k \geq 1$. Our reduction maps a pair $\langle R_1, R_2 \rangle$ of SREs to a pair of programs $\langle \Pi_1, \Pi_2 \rangle$ such that $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$ if and only if $\Pi_1 \sqsubseteq \Pi_2$, for $\mathcal{L}(R_i)$ the language of R_i. Programs Π_1 and Π_2 consist of several components, which we describe next.

- For each $m \geq 1$, program Π^m_{succ} implements a binary counter with m bits. It consists of Rules (5.43)–(5.46) and a rule of the form (5.47) for each $i \in [0, m-1]$ where F and A are fresh unary predicates, *Bit* is a fresh binary predicate, $succ^m$ is a fresh $(2m+1)$-ary predicate, $\bar{0}$ and $\bar{1}$ are fresh objects—intuitively standing for zero and one, respectively—each x_j is a fresh object variable, each $\mathbf{x}_j$ is the list of variables $x_1, \ldots, x_j$, each $\bar{\mathbf{1}}_j$ and each $\bar{\mathbf{0}}_j$ is the list of $\bar{1}$'s and $\bar{0}$'s, respectively, having length $m - j - 1$:

$$F(t) \rightarrow A(t) \tag{5.43}$$

$$A(t) \rightarrow A(t+1) \tag{5.44}$$

$$A(t) \rightarrow Bit(\bar{0}, t) \tag{5.45}$$

$$A(t) \rightarrow Bit(\bar{1}, t) \tag{5.46}$$

$$\textstyle\bigwedge_{j=1}^{i} Bit(x_j, t) \rightarrow succ^m(\mathbf{x}_i, \bar{0}, \bar{\mathbf{1}}_i, \mathbf{x}_i, \bar{1}, \bar{\mathbf{0}}_i, t) \tag{5.47}$$

  Formally, for each $m \geq 1$, each time point τ, and each $\tau' \geq \tau$, it holds that $\Pi^m_{\text{succ}} \cup \{F(\tau)\} \models succ^m(\mathbf{i}, \mathbf{j}, \tau')$ if and only if $\mathbf{i}$ and $\mathbf{j}$ are m-tuples over $\{\bar{0}, \bar{1}\}$, and $i + 1 = j$ for i and j the numbers encoded by $\mathbf{i}$ and $\mathbf{j}$, respectively.

- For a given SRE R over alphabet Σ, program Π_R is constructed inductively as follows, where G is a fresh unary predicate, A_σ is a fresh unary predicate for each $\sigma \in \Sigma$, and $\phi(\Pi)$ and $\psi(\Pi)$ are the programs obtained from an arbitrary program Π by renaming each predicate $P \notin \{A_\sigma \mid \sigma \in \Sigma\} \cup \{succ^m \mid m \geq 0\}$ to fresh predicates P^ϕ and P^ψ, respectively:

  - If $R = \emptyset$, then $\Pi_R = \emptyset$.
  - If $R = \varepsilon$, then Π_R consists of the rule

$$F(t) \rightarrow G(t). \tag{5.48}$$

  - If $R = \sigma$ for $\sigma \in \Sigma$, then Π_R consists of the rule

$$F(t) \wedge A_\sigma(t) \rightarrow G(t+1). \tag{5.49}$$

– If $R = S \cup T$, then Π_R extends $\phi(\Pi_S) \cup \psi(\Pi_T)$ with the rules

$$F(t) \to F^{\phi}(t) \tag{5.50}$$

$$F(t) \to F^{\psi}(t) \tag{5.51}$$

$$G^{\phi}(t) \to G(t) \tag{5.52}$$

$$G^{\psi}(t) \to G(t). \tag{5.53}$$

– If $R = S \circ T$, then Π_R extends $\phi(\Pi_S) \cup \psi(\Pi_T)$ with the rules

$$F(t) \to F^{\phi}(t) \tag{5.54}$$

$$G^{\phi}(t) \to F^{\psi}(t) \tag{5.55}$$

$$G^{\psi}(t) \to G(t). \tag{5.56}$$

– If $R = S^{+}$, then Π_R extends $\phi(\Pi_S)$ with the rules

$$F(t) \to F^{\phi}(t) \tag{5.57}$$

$$G^{\phi}(t) \to F^{\phi}(t) \tag{5.58}$$

$$G^{\phi}(t) \to G(t). \tag{5.59}$$

– If $R = S^{k}$ with $k \geq 1$, then Π_R is constructed from Π_S as follows. First, we replace each atom $P(\mathbf{p}, s)$ with $P'(\mathbf{p}, \mathbf{x}, s)$, with P' fresh and $|\mathbf{x}| = m$ for m the number of bits required to encode k. Then, we extend the resulting program with the following rules where $\mathbf{a}$ is the encoding of $k - 1$ as a binary string over $\bar{0}$ and $\bar{1}$ using m bits:

$$F(t) \to F'(\bar{\mathbf{0}}, t) \tag{5.60}$$

$$G'(\mathbf{a}, t) \to G(t) \tag{5.61}$$

$$G'(\mathbf{x}, t) \wedge succ^{m}(\mathbf{x}, \mathbf{y}, t) \to F'(\mathbf{y}, t) \tag{5.62}$$

Then, Π_1 and Π_2 are defined as follows, where Π_{succ} is the union of all Π^{m}_{succ} such that $succ^{m}$ occurs in $\Pi_{R_1} \cup \Pi_{R_2}$, program Π'_{R_2} is obtained from Π_{R_2} by renaming each IDB predicate P to a fresh predicate P' (in particular, G is renamed to G'), and G^{*} is a fresh unary predicate:

- $\Pi_1 = \Pi_{R_1} \cup \Pi'_{R_2} \cup \Pi_{\text{succ}} \cup \{G(t) \to G^*(t)\}$
- $\Pi_2 = \Pi_{R_1} \cup \Pi'_{R_2} \cup \Pi_{\text{succ}} \cup \{G'(t) \to G^*(t)\}$

We next argue correctness of the reduction. For this, we first show that our construction captures the language of the relevant SREs in the sense of the following two claims.

Claim 5.35. *Let $s = \sigma_1 \dots \sigma_n$ be a word in $\mathcal{L}(R)$ and let D be a finite set of EDB facts. If Π_{succ} includes each Π^m_{succ} such that succ^m occurs in Π_R, $F(\tau) \in D$, and $A_{\sigma_i}(\tau + i - 1) \in D$ for each $i \in [1, n]$, then $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + n)$.*

We show the claim by induction on R. Consider the base cases. Clearly, $R \neq \emptyset$ as $s \in \mathcal{L}(R)$. If $R = \sigma$, we have $s = \sigma$; but then, $A_\sigma(\tau) \in D$ implies $\Pi_R \cup D \models G(\tau + 1)$ by rule (5.49). If $R = \varepsilon$, then $s = \varepsilon$ (and hence $n = 0$); but then, $\Pi_R \cup D \models G(\tau)$ by rule (5.48). Next, we consider the inductive cases.

- $R = S \cup T$ and $s \in \mathcal{L}(S) \cup \mathcal{L}(T)$. If $s \in \mathcal{L}(S)$, then $\Pi_S \cup \Pi_{\text{succ}} \cup D \models G(\tau+n)$ by the inductive hypothesis; in this case, $\phi(\Pi_S) \cup \Pi_{\text{succ}} \cup D \cup \{F^\phi(\tau)\} \models G^\phi(\tau + n)$ by the definition of $\phi(\Pi_S)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + n)$ by Rules (5.50) and (5.52). The case $s \in \mathcal{L}(T)$ proceeds analogously using Rules (5.51) and (5.53).

- $R = S \circ T$ and $s = s_1 s_2$ with $s_1 = \sigma^1_1 \dots \sigma^1_{n_1} \in \mathcal{L}(S)$ and $s_2 = \sigma^2_1 \dots \sigma^2_{n_2} \in \mathcal{L}(T)$. By the inductive hypothesis, $\Pi_S \cup \Pi_{\text{succ}} \cup D \models G(\tau + n_1)$, hence $\phi(\Pi_S) \cup \Pi_{\text{succ}} \cup D \cup \{F^\phi(\tau)\} \models G^\phi(\tau + n_1)$ by the construction of $\phi(\Pi_S)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F^\psi(\tau + n_1)$ by Rules (5.54) and (5.55). Furthermore, again by the inductive hypothesis, we have $\Pi_T \cup \Pi_{\text{succ}} \cup D \cup \{F(\tau + n_1)\} \models G(\tau + n_1 + n_2)$, hence $\psi(\Pi_T) \cup \Pi_{\text{succ}} \cup D \cup \{F^\psi(\tau + n_1)\} \models G^\psi(\tau + n_1 + n_2)$ by the construction of $\psi(\Pi_T)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + n_1 + n_2)$ by rule (5.56).

- $R = S^+$ and $s = s_1 s_2 \dots s_k$ with $s_i = \sigma^i_1 \dots \sigma^i_{n_i} \in \mathcal{L}(S)$ for each $i \in [1, k]$. By a straightforward induction on i, which uses the outer inductive hypothesis applied to S together with Rules (5.57) and (5.58) applied in the base and

inductive case respectively, we can show $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G^{\phi}(\tau + \sum_{j=1}^{i} n_j)$ for each $i \in [1, k]$. This implies $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G^{\phi}(\tau + \sum_{i=1}^{k} n_i)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + \sum_{i=1}^{k} n_i)$ by rule (5.59).

- $R = S^k$ and $s = s_1 s_2 \dots s_k$ with $s_i = \sigma_1^i \dots \sigma_{n_i}^i \in \mathcal{L}(S)$ for each $i \in [1, k]$. We show by induction on $i \geq 1$ that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau + \sum_{j=1}^{i} n_j)$ for $\mathbf{b}$ the binary encoding of $i - 1$; this implies $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau + \sum_{i=1}^{k} n_i)$, and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau + \sum_{i=1}^{k} n_i)$ by rule (5.61). In the base case ($i = 1$), we have $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\bar{\mathbf{0}}, \tau)$ by rule (5.60), and hence $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\bar{\mathbf{0}}, \tau + n_1)$, because $\Pi_S \cup \Pi_{\text{succ}} \cup D \models G(\tau + n_1)$ by the outer inductive hypothesis for S, and by the construction of Π_R. For $i > 1$, the inner inductive hypothesis yields $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{c}, \tau + \sum_{j=1}^{i-1} n_j)$ where $\mathbf{c}$ is the binary encoding of $i - 2$. Hence, $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau + \sum_{j=1}^{i-1} n_j)$ where $\mathbf{b}$ encodes $i - 1$, by rule (5.62) and by the construction of Π_{succ}. But then, $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau + \sum_{j=1}^{i} n_j)$ follows by the construction of Π_R from $\Pi_S \cup D \cup \{F(\tau + \sum_{j=1}^{i-1} n_j)\} \models G(\tau + \sum_{j=1}^{i} n_j)$, which holds by the outer inductive hypothesis for S.

Claim 5.36. *Let D be a finite set of EDB facts and let τ be a time point. If $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau)$, then there exists a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(R)$ such that $F(\tau - n) \in D$ and $A_{\sigma_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$.*

We show the claim by induction on R. For the base cases, note first that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G(\tau)$ implies that $R \neq \emptyset$. If $R = \sigma$, we take $s = \sigma$ since $\{F(\tau - 1), A_{\sigma}(\tau - 1)\} \subseteq D$ by the construction of Π_R. Finally, if $R = \varepsilon$ we take $s = \varepsilon$ since $F(\tau) \in D$ by the construction of Π_R. We now consider the inductive cases.

- If $R = S \cup T$, we have $\Pi_R \cup D \models G^{\phi}(\tau)$ or $\Pi_R \cup D \models G^{\psi}(\tau)$. By the construction of Π_R and the inductive hypothesis, it follows that there is a word $s = a_1 \dots a_n \in \mathcal{L}(S) \cup \mathcal{L}(T)$ such that $A_{a_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$ and either $\Pi_R \cup D \models F^{\phi}(\tau - n)$ or $\Pi_R \cup D \models F^{\psi}(\tau - n)$, which both imply $F(\tau - n) \in D$ as required.

- If $R = S \circ T$, we have $\Pi_R \cup D \models G^{\psi}(\tau)$. By the construction of Π_R and the inductive hypothesis, there is a word $s_1 = a_1 \dots a_n \in \mathcal{L}(T)$ such that $A_{a_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$ and $\Pi_R \cup D \models F^{\psi}(\tau - n)$. Then, $\Pi_R \cup D \models F^{\psi}(\tau - n)$ implies $\Pi_R \cup D \models G^{\phi}(\tau - n)$, and hence, again by the construction of Π_R and the inductive hypothesis, we have that there is a word $s_2 = b_1 \dots b_{n'} \in \mathcal{L}(S)$ such that $A_{b_i}(\tau - n - n' + i - 1) \in D$ for each $i \in [1, n']$ and $\Pi_R \cup D \models F^{\phi}(\tau - n - n')$, which implies $F(\tau - n - n') \in D$. Finally, note that $s = s_1 s_2 \in \mathcal{L}(R)$, and hence s and D are as required.

- $R = S^+$. By the construction of Π_R and the inductive hypothesis, $\Pi_R \cup D \models G^{\phi}(\tau')$ with $\tau' \leq \tau$ implies that there is a word $\sigma_1 \dots \sigma_n \in \mathcal{L}(S)$ such that $\Pi_R \cup D \models F^{\phi}(\tau' - n)$ and $A_{\sigma_i}(\tau' - n + i - 1) \in D$ for each $i \in [1, n]$. By generalising the argument for $R = S \circ T$, we can then deduce the existence of words $s_1, \dots, s_k$ (for $k \geq 1$) such that, for each $i \in [1, k]$:

  - $s_i = \sigma_1^i \dots \sigma_{n_i}^i \in \mathcal{L}(S)$,
  - $\Pi_R \cup D \models F^{\phi}(\tau - \sum_{j=i}^{k} n_j)$,
  - $A_{\sigma_j^i}(\tau + j - 1 - \sum_{\ell=i}^{k} n_\ell) \in D$ for each $j \in [1, n_i]$,
  - $F(\tau - \sum_{i=1}^{k} n_i) \in D$.

  Then, $s = s_1 \cdots s_k$ and D are as required.

- $R = S^k$ for $k \geq 1$. By the construction of Π_R and the inductive hypothesis, the assertion $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{b}, \tau')$ with $\tau' \leq \tau$ implies the existence of a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(S)$ such that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau' - n)$ and $A_{\sigma_i}(\tau' - n + i - 1) \in D$ for each $i \in [1, n]$; furthermore, by the construction of Π_{succ}, if $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau')$ with $\mathbf{b}$ encoding $k \geq 0$ then $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{c}, \tau')$ with $\mathbf{c}$ encoding $k - 1$ due to rule (5.62). Considering that $\Pi_R \cup \Pi_{\text{succ}} \cup D \models G'(\mathbf{a}, \tau)$ with $\mathbf{a}$ the binary encoding of $k - 1$, due to rule (5.61), the two properties above imply (as it can be shown by a straightforward induction on i from k to 1) the existence of words $s_1, \dots, s_k$ such that, for each $i \in [1, k]$:

- $s_i = \sigma_1^i \dots \sigma_{n_i}^i \in \mathcal{L}(S)$;
- $\Pi_R \cup \Pi_{\text{succ}} \cup D \models F'(\mathbf{b}, \tau - \sum_{j=i}^{k} n_j)$ where $\mathbf{b}$ is the binary encoding of $i-1$; and
- $A_{\sigma_j^i}(\tau + j - 1 - \sum_{\ell=i}^{k} n_\ell) \in D$ for each $j \in [1, n_i]$.

Then, $s = s_1 \cdots s_k$ and D are as required. This concludes the proof of the claim.

We now proceed with the correctness proof. Assume first that $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$. Pick an arbitrary finite set D of EDB facts and a fact α such that $\Pi_1 \cup D \models \alpha$. We show $\Pi_2 \cup D \models \alpha$, which is sufficient to establish $\Pi_1 \sqsubseteq \Pi_2$. It is clear that $\Pi_2 \cup D \models \alpha$ if α is not a fact about G^*, so assume that α has the form $G^*(\tau)$. It follows that $\Pi_1 \cup D \models G(\tau)$, and hence $\Pi_{R_1} \cup \Pi_{\text{succ}} \cup D \models G(\tau)$. By Claim 5.36, there is a word $s = \sigma_1 \dots \sigma_n \in \mathcal{L}(R_1)$ such that $F(\tau - n) \in D$ and $A_{\sigma_i}(\tau - n + i - 1) \in D$ for each $i \in [1, n]$. Then, by assumption, $s \in \mathcal{L}(R_2)$, and hence $\Pi_{R_2} \cup \Pi_{\text{succ}} \cup D \models G(\tau)$ by Claim 5.35. Consequently $\Pi'_{R_2} \cup \Pi_{\text{succ}} \cup D \models G'(\tau)$, and thus $\Pi_2 \cup D \models G^*(\tau)$.

Assume now that $\Pi_1 \sqsubseteq \Pi_2$. We show $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$. Let $s = a_1 \dots a_n$ be a word in $\mathcal{L}(R_1)$ and let D be the set consisting of the fact $F(0)$ and a fact $A_{a_i}(i-1)$ for each $i \in [1, n]$. It follows that $\Pi_{R_1} \cup \Pi_{\text{succ}} \cup D \models G(n)$ by Claim 5.35, and hence $\Pi_1 \cup D \models G^*(n)$. By assumption, this implies $\Pi_2 \cup D \models G^*(n)$, and hence, by construction, $\Pi_{R_2} \cup \Pi_{\text{succ}} \cup D \models G(n)$. Then, by Claim 5.36, there is some $s' = b_1 \dots b_{n'} \in \mathcal{L}(R_2)$ such that $F(n - n') \in D$ and $A_{b_i}(n - n' + i - 1) \in D$ for each $1 \leq i \leq n'$. Furthermore, $n' = n$ since $F(0)$ is the only fact about F in D by the definition of D, and, for each $i \in [1, n]$, we have $b_i = a_i$ since $A_{a_i}(i-1)$ is the only fact in D of the form $A_\sigma(i-1)$. Therefore, $s' = s$, and hence $s \in \mathcal{L}(R_2)$, as required.

This completes our ExpSpace-hardness proof. We conclude by arguing that containment of object-free forward-propagating programs is PSpace-hard, which also implies PSpace-hardness for programs with a fixed arity over a fixed object domain. In the case of object-free forward-propagating programs, we reduce from containment of standard regular expressions (without exponentiation), which is a well-known PSpace-hard problem. The reduction is a special case of our reduction from SRE

containment—it suffices to observe that we no longer need the program component Π_{succ}, without which the program becomes object-free. The correctness arguments transfer verbatim. □

Theorem 5.37. *Delay existence, delay validity, and window size validity with respect to a fixed object domain are* ExpSpace-hard*; the problems become* PSpace-hard *if additionally restricted to programs where predicates have a fixed maximum arity (in particular, to object-free programs). The bounds for delay validity hold already for backward-bounded programs, whereas the bounds for window size validity hold already for forward-propagating programs.*

Proof. The proof of Theorem 5.2 defines a reduction f_1 from program containment to delay existence. The proof of Theorem 5.3 defines a reduction f_2 from program containment to delay validity for backward-bounded programs. Finally, the proof of Theorem 5.4 establishes a reduction f_3 from program containment to window size validity for forward-propagating programs. The properties of these reductions do not rely on whether or not the object domain is considered fixed. Furthermore, when given object-free programs as input, the reductions produce object-free programs.

The statement of the theorem then trivially follows from Theorem 5.34, which establishes ExpSpace-hardness of containment for forward-propagating programs with respect to a fixed object domain, as well as PSpace-hardness for the case where, in addition, programs are restricted so to have a fixed maximum arity of predicates (which includes object-free programs). □

Chapter 6

Related Work

In this section, we review the related work. We discuss: languages for stream processing obtained by extending existing languages with window constructs; temporal extensions of Datalog related to Temporal Datalog; existing stream reasoning approaches based on rules; and, finally, problems and techniques that have connections to our work.

6.1 Languages for Stream Processing in Databases and Semantic Web

The formal underpinnings of stream query processing in databases were established in [6, 5]. Arasu et al. [5] proposed the Continuous Query Language (CQL) as an extension of SQL with window constructs, which specify the input data relevant for query processing at any point in time. The CQL approach has been adopted in the design of many other stream query languages, including languages for the Semantic Web, such as Streaming-SPARQL [20], C-SPARQL [10], CQELS [43], RSP-QL [32], EP-SPARQL [4], SPARQL$_{stream}$ [22], and STARQL [41].

Some of the problems we presented in this thesis are relevant to the languages mentioned above, even though they are less expressive than Temporal Datalog from a temporal point of view. For instance, database views expressed in CQL capture forward-propagating non-recursive programs; thus, window size validity is computationally hard for them, as it follows from the results in [45]. At the same time, CQL

views always admit zero as a valid delay, and hence delay existence and delay validity checking are trivial problems for CQL.

6.2 Temporal Extensions of Datalog

There have been many proposals for extending Datalog with temporal features. In this line of work, the focus is typically not on stream reasoning, but rather on standard database research problems such as determining data complexity of fact entailment [26, 24, 50] and establishing the expressive power of the language [12, 50]. In addition, many of these works also consider problems specific to temporal deductive databases such as computing finite specifications of infinite query answers [26, 27].

The language considered in this thesis is a notational variant of $\text{Datalog}_{1\text{S}}$ [26] with the additional temporal guardedness condition. Chomicki and Imieliński assume in their technical results in [26] that $\text{Datalog}_{1\text{S}}$ rules mention at most one time variable; hence, our guardedness condition only imposes the additional requirement that rules do not mention explicit time points. $\text{Datalog}_{1\text{S}}$ as proposed as a language for temporal deductive databases, which have been surveyed in [12].

Templog is an extension of Datalog with temporal modal operators [1]. As shown by Baudinet et al. [12], Templog and $\text{Datalog}_{1\text{S}}$ are interreducible, and hence both languages are equivalent in terms of expressive power. Baudinet [11] studied the expressive power of Templog and showed that it expresses exactly the finitely regular ω-languages—that is, those languages recognised by finite-acceptance automata on infinite words, where a finite-acceptance automaton is a classical finite automaton that is applied to prefixes of infinite words. This coincides with the expressive power of the μTL^+ fragment of Vardi's [49] fixpoint calculus μTL without negation or greatest fixpoints. Furthermore, Templog with stratified negation expresses exactly the ω-regular languages, and hence its expressiveness coincides with the one of full μTL.

DatalogMTL is an extension of Datalog with metric temporal operators that has been studied in the context of querying temporal data [21, 50] and reasoning over data streams [51]. DatalogMTL allows for Metric Temporal Logic expressions in rules such

as $\boxminus_{[k_1,k_2]}\varphi$ and $\diamondminus_{[k_1,k_2]}\varphi$, with k_1 and k_2 rational numbers, which hold at time t if φ holds at each and some, respectively, moment in the time interval $[t-k_2, t-k_1]$. $\text{Datalog}_{1\text{S}}$ programs (and hence also our Temporal Datalog programs) can easily be encoded in DatalogMTL.

Finally, the language proposed by Toman and Chomicki [48] extends Datalog with integer periodicity constraints, which can be used to encode and store information about periodic activities in temporal databases.

6.3 Rule-Based Stream Reasoning

Barbieri et al. [9] consider stream reasoning over non-temporal Datalog programs that are temporally interpreted as if each atom holds at the current time; this can be easily encoded in our framework by replacing each (non-temporal) atom $A(\mathbf{s})$ with the atom $A(\mathbf{s}, t)$ for t a fixed time variable. In this setting, both the delay and the window size problems are trivial since rules refer only to the present time point and hence zero is always a valid delay and window size. In contrast to our work, where data facts involve single time points, Barbieri et al. consider facts annotated with a validity interval $[\tau_1, \tau_2]$, where τ_1 and τ_2 represent insertion and expiration time, respectively.

Wałęga et al. [51] propose a stream reasoning algorithm for a forward-propagating fragment of DatalogMTL, which admits valid delay zero. Wałęga et al. do not consider the window size problem in their work, and their stream reasoning algorithm relies on a window size derived through a syntactic condition similar to ours based on the maximum rule radius. Wałęga et al. address in their algorithm a number of additional difficulties stemming from the fact that DatalogMTL is a richer language than Temporal Datalog, and that it is interpreted over the non-negative rational numbers rather than the natural numbers.

Zaniolo [52] proposes Streamlog: a language for representing standing queries that extends Temporal Datalog with non-monotonic negation while at the same time restricting the syntax so that only facts over time points mentioned in the data can

be derived. Each atom in a Streamlog rule has a single time variable, and time variables can occur in inequality conditions involving also other variables and arithmetic operations. The paper focuses on *sequential programs*, which are locally stratified by restricting the use of inequality conditions so that the time argument of a positive body literal is at most the time argument of the head, and the time argument of a negative body literal is strictly smaller than the time argument of the head. As a result, sequential programs admit a unique stable model, which can be computed by increasing values of time. Such programs thus have delay zero; furthermore, if parametrised with a valid window size (an issue not addressed in [52]), they can be evaluated by a variant of our stream reasoning algorithm that considers each past fact that has not been derived as false—in line with Zaniolo's *progressive closing world assumption* (PCWA).

LARS [15, 19] is a temporal rule-based stream reasoning language featuring built-in window constructs. LARS formulas extend propositional logic with temporal and window operators, and LARS programs are defined as ASP programs allowing for LARS formulas in place of atoms. A stream is seen a function from a finite interval of the natural numbers to sets of propositions, thus streams in LARS are intrinsically bounded. The issues stemming from reasoning over unbounded streams in LARS have been addressed only in [16, 13]. Beck et al. [16] describe a LARS-based system where reasoning over streams is reduced to repeated reasoning over datasets, with the latter task solved by a logic-programming reasoner. However, the reduction is given just for the specific LARS program considered in the paper. Laser [13] is a stream reasoner supporting a fragment of LARS called *plain LARS*, first introduced in [18]. In contrast to our work, the semantics implemented by Laser requires only that the system derive the consequences that do not depend on future facts; specifically, a consequence holding at time τ will be output if and only if it can be derived by reasoning over the interval $[0, \tau]$. By the properties of plain LARS—that can propagate into the past, but cannot distinguish the past from the present—this semantics can be implemented by an algorithm that does not propagate information backwards in time. As a consequence, the delay validity problem is not relevant in this setting.

Other works on LARS address one-off [15, 19] and incremental [14, 18] reasoning over (finite) datasets, program equivalence [17], and translation of other languages into LARS [31].

Our results in this thesis build on our prior conference publications [45, 44]. Our stream reasoning algorithm is a variant of the "offline" algorithm in [45], where the main difference is that the algorithm in [45] only keeps in memory EDB facts from the input stream (and hence does not retain derived IDB facts from one iteration to the next). As a result, the delay and window size validity notions are defined differently. Despite these differences, however, the reductions from the containment problem to delay and window size validity problems presented in this thesis (see Theorems 5.3 and 5.4) are very similar to those in [45]. Finally, our focus in [45] was on non-recursive programs, which are much weaker than backward-bounded programs. The stream reasoning algorithm we present in this thesis was first introduced in [44] for forward-propagating programs, which always admit delay zero. Therefore, in [44] we focused on window size validity and showed that window size validity and containment are interreducible for forward-propagating programs. Furthermore, we established tight complexity bounds for containment of forward-propagating programs; in particular, the lower bounds for the containment problem presented here in Theorem 5.34 were proved in [44]. The results in this thesis extend [44] to programs that are not necessarily forward-propagating by generalising the stream reasoning algorithm, studying the the window size validity problem in the extended setting, and studying for the first time the problems associated to the notion of program delay.

6.4 Related Problems and Techniques

A question closely related to stream reasoning is that of checking *dynamic integrity constraints* for relational databases [25]. In contrast to standard database constraints, dynamic constraints can refer to different states of a database, which are determined by the relevant sequence of transactions. Chomicki [25] considers First-Order Temporal Logic (FOTL), which extends First-Order Logic with past-only temporal oper-

ators, as a language to express such dynamic constraints. Chomicki also proposed an incremental update algorithm for checking dynamic FOTL constraints. Chomicki's algorithm builds on the observation that FOTL admits an inductive definition over time, where the truth of subformulas at the current time point can be derived from the truth of subformulas at the previous time point. As a consequence, the algorithm only needs to store a polynomially-bounded part of the update history (considering the constraints fixed). When applied to forward-propagating programs, our stream reasoning algorithm behaves similarly to Chomicki's: it computes and stores all (EDB and IDB) consequences of the input program for increasing time points while keeping in memory only a polynomially bounded set of facts (considering the program fixed).

There is a connection between delay existence and checking *boundedness* of a Datalog program, that is, checking whether a program is (semantically) recursive. Intuitively, a Temporal Datalog program admits a valid delay if and only if it is non-recursive with respect to temporal backward propagation, and hence delay existence can be seen conceptually as a boundedness check with respect to temporal recursion towards past time points. Cosmadakis et al. [29] established complexity bounds for monadic Datalog programs using techniques similar in spirit to those in Section 5.3.2, where automata are used to recognise languages related to the fact entailment problem and its complement.

The automaton for the notimplies language defined in Section 5.3.2 is related to the algorithm in [26] used by Chomicki and Imieliński to establish the data complexity of $\text{Datalog}_{1\text{S}}$, where the algorithm searches for counter-models over an exponentially-sized prefix of the timeline using a sliding window of polynomial size.

Chapter 7

Conclusions

7.1 Discussion

In this thesis, we have proposed and studied a suite of decision problems which enable the use of incremental stream reasoning algorithms based on a sliding window, while ensuring correctness and minimising both latency and memory consumption. Although these problems are undecidable for Temporal Datalog, we have shown decidability and established tight complexity bounds under the assumption that the set of objects that may occur in an input stream is fixed. We believe that our results constitute a first step towards the development of robust and efficient stream reasoning engines with provable correctness guarantees.

7.2 Future Work

We see many interesting avenues for future work.

Removing temporal guardedness It is unclear how to extend our results to deal with programs that do not satisfy our temporal guardedness condition. A key aspect of such an extension would be to deal effectively with multiple time variables in rules. Such rules can be rewritten into rules with a single time variable by introducing fresh predicates and recursion; for instance, rule $A(t) \wedge B(t') \rightarrow C(t)$ can be rewritten as the rules $A(t) \wedge B'(t) \rightarrow C(t)$, $B(t) \rightarrow B'(t)$, $B'(t) \rightarrow B'(t+1)$, and $B'(t) \rightarrow B'(t-1)$. Such rewriting, however, can easily turn a program admitting a valid delay

into a program with no valid delay. Multiple time variables introduce an additional challenge: a program may admit no valid window size even if it admits a valid delay. This is the case for the program consisting of rules $A(t) \to P(t)$, $P(t) \to P(t+1)$, and $P(t) \wedge A(t') \to Q(t)$; indeed, an input fact $A(0)$ can never be deleted. The same is true for rules mentioning time points, such as $A(0) \wedge B(t) \to C(t)$. We conjecture that such challenges could be tackled by extending the stream reasoning algorithm to also store witnesses for existentially quantified time variables, as well as all facts for the time points occurring explicitly in the input program.

Further decidable cases In some scenarios it may not be reasonable to assume a fixed object domain. Hence, it would be interesting to investigate recursive fragments (other than object-free Temporal Datalog) for which the delay and window size problems become decidable. A natural choice is to restrict programs so that IDB predicates have at most one object argument; such programs extend monadic Datalog [29], for which containment is known to be decidable.

Time offsets in binary All our complexity results assume that numbers occurring in input programs—i.e., time offsets—are coded in unary. It would be relevant to study complexity under binary coding of numbers. In particular, the transformation of programs into normal form used to establish complexity upper bounds and described in Theorem 5.9 yields an exponential blowup if we assume binary coding; it is left to understand whether an exponential blowup in complexity is avoidable.

Non-monotonic negation It would be interesting to extend our framework to cover programs featuring non-monotonic negation. In this setting, a stream reasoning algorithm may not only miss relevant entailments if parametrised with an invalid delay or window size, but also output unsound results. A possible way forward in this direction would be to consider an extension of the language in line with Zaniolo's sequential programs [52].

Delay and window size validity under constraints We might parametrise delay and window size validity with a set of constraints and change their definition so that they would guarantee correctness of a stream reasoning algorithm only on the streams that satisfy the constraints. A similar approach was taken for the containment problem in [23]. From a user point of view, constraints describe the set of streams that can occur in an application. Constraints would allow for a more refined analysis. For instance, a program may admit a valid delay over streams satisfying a given set of constraints while not admitting a delay on unrestricted streams. One may consider traditional integrity constraints such as key constraints and inclusion dependencies, or could allow one to specify constraints in a rich language like the constraint language in [34].

Beyond sliding windows In [45] we considered a stream reasoning algorithm that does not use a sliding window and, instead, checks on-the-fly when consequences are to be output and when stored data is to be deleted. Such algorithm can achieve a lower latency and store fewer facts. Most importantly, it can process programs that do not admit a valid delay—although it may wait arbitrarily long before outputting any fact, and it may never delete any past fact from the memory. There exist meaningful programs that do not admit a delay such as the one shown in Example 3.13. The underlying decision problems for the above-mentioned algorithm in [45] were studied only for non-recursive programs, which cannot even express the program of Example 3.13. The problems could be studied for the fragments of Temporal Datalog considered in this thesis as well as for others.

Stream reasoning in other languages Our framework can be adapted to other temporal rule languages, yielding corresponding delay and window size problems that can then be studied. In particular, it would be interesting to consider temporal languages with a dense model of time such as DatalogMTL [21, 51, 50].

References

[1] Martín Abadi and Zohar Manna. Temporal logic programming. *J. Symb. Comput.*, 8(3):277–295, 1989.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11): 1033–1044, 2013.

[4] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: A unified language for event processing and stream reasoning. In *Proc. 20th International Conference on World Wide Web (WWW 2011)*, pages 635–644. ACM, 2011.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002)*, pages 1–16. ACM, 2002.

[7] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[8] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, and Clément Sipieter. Graal: A toolkit for query answering with existential rules. In *Proc. 9th International RuleML Symposium (RuleML 2015)*, volume 9202 of *LNCS*, pages 328–344. Springer, 2015.

[9] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *Proc. 7th Extended Semantic Web Conference (ESWC 2010)*, volume 6088 of *LNCS*, pages 1–15. Springer, 2010.

[10] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: A continuous query language for RDF data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.

[11] Marianne Baudinet. On the expressiveness of temporal logic programming. *Inf. Comput.*, 117(2):157–180, 1995.

[12] Marianne Baudinet, Jan Chomicki, and Pierre Wolper. Temporal deductive databases. In Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass, editors, *Temporal Databases*, chapter 13, pages 294–320. Benjamin Cummings, 1993.

[13] Hamid R. Bazoobandi, Harald Beck, and Jacopo Urbani. Expressive stream reasoning with Laser. In *Proc. 16th International Semantic Web Conference (ISWC 2017), Part I*, volume 10587 of *LNCS*, pages 87–103. Springer, 2017.

[14] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Answer update for rule-based stream reasoning. In *Proc. 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 2741–2747. AAAI Press, 2015.

[15] Harald Beck, Minh Dao-Tran, Thomas Eiter, and Michael Fink. LARS: A logic-based framework for analyzing reasoning over streams. In *Proc. 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 1431–1438. AAAI Press, 2015.

[16] Harald Beck, Bruno Bierbaumer, Minh Dao-Tran, Thomas Eiter, Hermann Hellwagner, and Konstantin Schekotihin. Rule-based stream reasoning for intelligent administration of content-centric networks. In *Proc. 15th European Conference on Logics in Artificial Intelligence (JELIA 2016)*, volume 10021 of *LNCS*, pages 522–528. Springer, 2016.

[17] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Equivalent stream reasoning programs. In *Proc. 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 929–935. IJCAI/AAAI Press, 2016.

[18] Harald Beck, Thomas Eiter, and Christian Folie. Ticker: A system for incremental ASP-based stream reasoning. *Theory Pract. Log. Program.*, 17(5-6):744–763, 2017.

[19] Harald Beck, Minh Dao-Tran, and Thomas Eiter. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.*, 261:16–70, 2018.

[20] Andre Bolles, Marco Grawunder, and Jonas Jacobi. Streaming SPARQL: Extending SPARQL to process data streams. In *Proc. 5th European Semantic Web Conference (ESWC 2008)*, volume 5021 of *LNCS*, pages 448–462. Springer, 2008.

[21] Sebastian Brandt, Elem Güzel Kalayci, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Querying log data with Metric Temporal Logic. *J. Artif. Intell. Res.*, 62:829–877, 2018.

[22] Jean-Paul Calbimonte, Óscar Corcho, and Alasdair J. G. Gray. Enabling ontology-based access to streaming data sources. In *Proc. 9th International Semantic Web Conference (ISWC 2010)*, volume 6496 of *LNCS*, pages 96–111. Springer, 2010.

[23] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 149–158, 1998.

[24] Jan Chomicki. Polynomial time query processing in temporal deductive databases. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1990)*, pages 379–391. ACM, 1990.

[25] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186, 1995.

[26] Jan Chomicki and Tomasz Imieliński. Temporal deductive databases and infinite objects. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1988)*, pages 61–73. ACM, 1988.

[27] Jan Chomicki and Tomasz Imieliński. Relational specifications of infinite query answers. In *Proc. 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD 1989)*, pages 174–183. ACM, 1989.

[28] Charles Cosad, Kerby Dufrene, Katy Heidenreich, Mike McMillon, Alex Jermieson, Meghan O'Keefe, and Louise Simpson. Wellsite support from afar. *Oilfield Review*, 21(2):48–58, 2009.

[29] Stavros S. Cosmadakis, Haim Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs (Preliminary report). In *Proc. 20th Annual ACM Symposium on Theory of Computing (STOC 1988)*, pages 477–490. ACM, 1988.

[30] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3): 374–425, 2001.

[31] Minh Dao-Tran, Harald Beck, and Thomas Eiter. Towards comparing RDF stream processing semantics. In *Proc. 1st Workshop on High-Level Declarative Stream Processing (HiDeSt 2015)*, volume 1447 of *CEUR Workshop Proceedings*, pages 15–27. CEUR-WS.org, 2015.

[32] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Óscar Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. *Int. J. Semantic Web Inf. Syst.*, 10(4):17–44, 2014.

[33] Thomas Eiter, Michael Fink, Hans Tompits, and Stefan Woltran. Strong and uniform equivalence in answer-set programming: Characterizations and complexity results for the non-ground case. In *Proc. 20th National Conference on Artificial Intelligence (AAAI 2005)*, pages 695–700. AAAI Press / The MIT Press, 2005.

[34] Hector Garcia-Molina, Wilburt Labio, and Jun Yang. Expiring data in a warehouse. In *Proceedings of Twenty-Fourth International Conference on Very Large Data Bases (VLDB 1998)*, pages 500–511, 1998.

[35] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[36] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. Parallel materialisation of Datalog programs in centralised, main-memory RDF systems. In *Proc. 28th AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 129–137. AAAI Press, 2014.

[37] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Combining rewriting and incremental materialisation maintenance for Datalog programs with equality. In *Proc. 24th International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 3127–3133. AAAI Press, 2015.

[38] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of Datalog materialisations revisited. *Artif. Intell.*, 269:76–136, 2019.

[39] Gerhard Münz and Georg Carle. Real-time analysis of flow data for network attack detection. In *Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management (IM 2007)*, pages 100–108. IEEE, 2007.

[40] Giuseppe Nuti, Mahnoosh Mirghaemi, Philip C. Treleaven, and Chaiyakorn Yingsaeree. Algorithmic trading. *IEEE Computer*, 44(11):61–69, 2011.

[41] Özgür Lütfü Özçep, Ralf Möller, and Christian Neuenstadt. A stream-temporal query language for ontology based data access. In *Proc. 37th Annual German Conference on AI (KI 2014)*, volume 8736 of *LNCS*, pages 183–194. Springer, 2014.

[42] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[43] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *Proc. 10th International Semantic Web Conference (ISWC 2011)*, volume 7031 of *LNCS*, pages 370–388. Springer, 2011.

[44] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, and Ian Horrocks. The window validity problem in rule-based stream reasoning. In *Proc. 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018)*, pages 571–581. AAAI Press, 2018.

[45] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, Boris Motik, and Ian Horrocks. Stream reasoning in Temporal Datalog. In *Proc. 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*, pages 1941–1948. AAAI Press, 2018.

[46] Oded Shmueli. Equivalence of Datalog queries is undecidable. *J. Log. Program.*, 15(3):231–241, 1993.

[47] Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2nd edition, 2006.

[48] David Toman and Jan Chomicki. Datalog with integer periodicity constraints. *J. Log. Program.*, 35(3):263–290, 1998.

[49] Moshe Y. Vardi. A temporal fixpoint calculus. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 250–259. ACM, 1988.

[50] Przemysław Andrzej Wałęga, Bernardo Cuenca Grau, Mark Kaminski, and Egor Kostylev. DatalogMTL: Computational complexity and expressive power. In *Proc. 28th International Joint Conference on Artificial Intelligence (IJCAI 2019)*, pages 1886–1892. ijcai.org, 2019.

[51] Przemysław Andrzej Wałęga, Mark Kaminski, and Bernardo Cuenca Grau. Reasoning over streaming data in Metric Temporal Datalog. In *Proc. 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, pages 3092–3099. AAAI Press, 2019.

[52] Carlo Zaniolo. Logical foundations of continuous query languages for data streams. In *Proc. 2nd International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2012)*, volume 7494 of *LNCS*, pages 177–189. Springer, 2012.