Future-looking logics on data words and trees

Diego Figueira and Luc Segoufin

INRIA, ENS Cachan, LSV

Abstract. In a data word or a data tree each position carries a label from a finite alphabet and a data value from an infinite domain.
Over data words we consider the logic $\mathsf{LTL}^{\downarrow}_1(\mathsf{F})$, that extends $\mathsf{LTL}(\mathsf{F})$ with one register for storing data values for later comparisons. We show that satisfiability over data words of $\mathsf{LTL}^{\downarrow}_1(\mathsf{F})$ is already not primitive recursive. We also show that the extension of $\mathsf{LTL}^{\downarrow}_1(\mathsf{F})$ with either the reverse modality F^{-1} or with one extra register is undecidable. All those lower bounds were already known for $\mathsf{LTL}^{\downarrow}_1(\mathsf{X},\mathsf{F})$ and our results essentially show that the X modality was not necessary.
Moreover we show that over data trees similar lower bounds hold for certain fragments of XPath.

1 Introduction

A data word (data tree) is a word (tree) where each position carries a label from a finite alphabet and a *datum* from some infinite domain. These models have been considered in the realm of semistructured data [4], timed automata [6] and extended temporal logics [10,9,13]. In this paper we consider an infinite domain with no structure where we can only test for equality or inequality between two elements.

There have been various other logics considered to specify properties over data words and data trees. For example from the standpoint of Temporal Logics (both on data words [9] and trees [13]), of First Order Logics (see [5] for data words case and [4] for trees), or of logic based on tree patterns [8,3]. The logic $\mathsf{LTL}^{\downarrow}_1(\mathsf{X},\mathsf{F})$ is the extension of $\mathsf{LTL}(\mathsf{X},\mathsf{F})$ with the ability to use one *register* for storing a data value for later comparisons. It has been studied in [10,9] where satisfiability and expressivity issues have been addressed. In [9] it has been established that the satisfiability problem for $\mathsf{LTL}^{\downarrow}_1(\mathsf{X},\mathsf{F})$ is decidable and not primitive recursive on data words. It is also shown in [9] that the *two way* extension $\mathsf{LTL}^{\downarrow}_1(\mathsf{X},\mathsf{F},\mathsf{F}^{-1})$ is undecidable over data words and, similarly, that the extension to 2 registers $\mathsf{LTL}^{\downarrow}_2(\mathsf{X},\mathsf{F})$ is undecidable.

Here we show that even without the X modality, all the lower bounds aforementioned remain valid: the satisfiability problem for $\mathsf{LTL}^{\downarrow}_1(\mathsf{F})$ over data words is not primitive recursive, while for $\mathsf{LTL}^{\downarrow}_1(\mathsf{F},\mathsf{F}^{-1})$ and $\mathsf{LTL}^{\downarrow}_2(\mathsf{F})$ is undecidable.

Note that in the absence of data values the satisfiability problem for $\mathsf{LTL}(\mathsf{F})$ is NP-complete whereas it is PSPACE-complete for $\mathsf{LTL}(\mathsf{X},\mathsf{F})$ [16]. Our work shows that in the presence of data values, the X modality does not imply further computational complexity.

Data trees can be seen as a coding of an XML document [4,13]. Therefore XPath, the node selecting language of W3C [7], can be considered as a logic over data trees. By XPath, we refer to the 1.0 specification without all the domain specific features (arithmetic, string manipulation, etc.) As XPath is at the core of many XML standard languages (like XQuery and XSLT), deciding satisfiability of some of its fragments can be of great help during optimization stages.

A data word can be seen as a special case of an unranked ordered data tree, for instance by adding a root that is the parent of all the positions of the data word. With this consideration the fragment $\mathsf{XPath}(\rightarrow, \rightarrow^+, =)$ of XPath that contains only the axis `next-sibling` ($\rightarrow$) and `following-sibling` ($\rightarrow^+$) can be seen as a fragment of $\mathsf{LTL}_1^{\downarrow}(\mathsf{X}, \mathsf{F})$. There are nonetheless two important differences between the expressive power of $\mathsf{XPath}(\rightarrow, \rightarrow^+, =)$ and $\mathsf{LTL}_1^{\downarrow}(\mathsf{X}, \mathsf{F})$. The first one is that the axis $\rightarrow^+$ corresponds to the strict future modality in LTL, denoted $\mathsf{F_s}$ in the sequel. Of course F can be defined using $\mathsf{F_s}$ but the opposite is not true in the absence of X. The second difference lies in the fact that XPath can compare data values in a way strictly more limited than $\mathsf{LTL}_1^{\downarrow}$. We illustrate this with an example. At a position of a data word, $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ can store the current data value in the register, check for a later symbol a with a data value different from the one in the register, and then further check for a symbol b with a data value matching the one of the register. In this spirit, XPath could only compare the data values at the beginning and the end of the path and could not say anything about the data value of the intermediate a symbol. Hence $\mathsf{XPath}(\rightarrow^+, =)$ should be seen as a fragment of $\mathsf{LTL}_1^{\downarrow}(\mathsf{F_s})$, incomparable with $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ in terms of expressive power.

Based on these ideas, we exhibit a syntactic fragment $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ of $\mathsf{LTL}_1^{\downarrow}(\mathsf{F_s})$ that limits the use of register comparisons and has the same expressive power than $\mathsf{XPath}(\rightarrow^+, =)$ on data words. We then show that satisfiability over data words of $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ is not primitive recursive. Similarly the same restriction on $\mathsf{LTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ correspond to $\mathsf{XPath}(\rightarrow^+, {}^+\!\leftarrow, =)$ and is shown to be undecidable.

Our non primitive recursive results are proved by a reduction from the emptiness problem of *faulty counter automata*. These are counter automata that may have *incrementing errors* in their counters during the run. Non-emptiness for this class of automata was proven to be decidable and not primitive recursive [18]. Our reduction will be centered in a strategy of using data values for coding – with some limitations– a *next step* move of this automaton. We show that the strict future modality together with the limited data comparison capabilities of $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ are sufficient for our coding. With the extra power of $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ for comparing data values, we also show that strictness of the future modality can be avoided. Similar ideas are used for our undecidable results: The extra available expressive power being sufficient for forbidding the incrementing errors and thus to code emptiness of a Minsky Counter Automaton.

Related work. There are known complexity results concerning the satisfiability problem of several data-aware fragments of XPath on data trees. When all axes are present, XPath is undecidable [12]. When all vertical axes are present but in the absence of any horizontal axis, the status of the decidability of

$\mathsf{XPath}(\downarrow, \downarrow^+, \uparrow, \uparrow^+, =)$ is not yet known [2]. In this paper we show that if decidable, the complexity of this fragment cannot be primitive recursive, even in the absence of the axis $\downarrow$ and $\uparrow$.

In [4] decidability over data trees of the two-variable fragment of first order logic, $\mathsf{FO}_2(+1, \sim)$ is established. As a direct consequence a fragment of XPath, with no occurrence of `descendant`/`ancestor` nor `following`/`previous-sibling` in the paths occurring in expressions with a data value test, is shown to be decidable. On the opposite, in the present work, we focus on satisfiability under the absence of the *successor* axis.

In [13] it is shown that the emptiness problem for alternating automata with one register over data trees is decidable. As a direct consequence it is shown that $\mathsf{XPath}(\downarrow, \downarrow^+, \rightarrow, \rightarrow^+, =)$, with some restriction on the expressions with a data value test, is decidable and not primitive recursive. As a consequence of the present work, the hardness result already holds for $\mathsf{XPath}(\downarrow^+, \rightarrow, =)$ and for $\mathsf{XPath}(\rightarrow^+, =)$.

We finally remark that, nevertheless, the satisfiability of $\mathsf{XPath}(\downarrow, \downarrow^+, =)$ is "only" ExpTime-complete [11].

2 Preliminaries

We fix a finite set Σ of *labels* and an infinite set D of *data values*. The models we consider are either data words or data trees. A data word σ over a finite alphabet Σ is a non-empty word of $(\Sigma \times D)^*$. We assume no structure on D and D will be used only to perform tests for (in)equalities. In our examples we will always use data values from $\mathbb{N}$ seen as a *set* of numbers. The data trees we will be using are unranked and ordered. The *domain* of an unranked ordered tree is represented by a prefix-closed set T of elements from $\mathbb{N}^*$ such that whenever $n(i+1) \in T$ then $ni \in T$. The elements of T will be called *nodes* of the tree. A data tree t over Σ, D is a tree domain T together with a labeling function λ assigning an element of $\Sigma \times D$ to any node of t. We use the standard terminology for trees such as descendant, ancestor, sibling, etc.

LTL *with registers* Our most expressive logic for data words is $\mathsf{LTL}^{\downarrow}_n(\mathsf{F}, \mathsf{X}, \mathsf{F}^{-1}, \mathsf{X}^{-1})$, the Linear Temporal Logic with the freeze quantifier ($\downarrow_i$), test predicate ($\uparrow_i$) and next (X) and future (F) temporal operators together with their inverse modalities ($\mathsf{X}^{-1}, \mathsf{F}^{-1}$). Sentences are defined:

$$\varphi ::= a \mid \downarrow_i \varphi \mid \uparrow_i \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathcal{O}\varphi$$

where a is a symbol from a finite alphabet Σ, $i \in \{1, \ldots, n\}$, $\mathcal{O}$ ranges over $\{\mathsf{F}, \mathsf{X}, \mathsf{F}^{-1}, \mathsf{X}^{-1}\}$. We also use the *strict future* temporal operator $\mathsf{F_s}$ as a shortcut for XF. We will consider fragments of this logics as simple restrictions to certain temporal operators.

Given a data word σ, we write σ_i for the ith element (duple) of the word, and π_1, π_2 for the projections on Σ and D. A *valuation* is defined as a partial map $v : \{1, \ldots, n\} \to D$. The satisfaction relation $\models$ is inductively defined (we omitted X^{-1} and F^{-1} for simplicity):

$\sigma, i \models_v a$ iff $\pi_1(\sigma_i) = a$ $\quad$ $\sigma, i \models_v \uparrow_k$ iff $k \in \mathrm{dom}(v)$ and $v(k) = \pi_2(\sigma_i)$
$\sigma, i \models_v \downarrow_k \varphi$ iff $\sigma, i \models_{v[k \mapsto \pi_2(\sigma_i)]} \varphi$ $\quad$ $\sigma, i \models_v \mathsf{X}\varphi$ iff $i < |\sigma|$ and $\sigma, i+1 \models_v \varphi$
$\sigma, i \models_v \mathsf{F}\varphi$ iff for some $i \leq j \leq |\sigma|$ we have $\sigma, j \models_v \varphi$

where $1 \leq i \leq |\sigma|$. We denote $\sigma, i \models_{v_\emptyset} \varphi$ by $\sigma, i \models \varphi$, with $v_\emptyset$ the empty valuation; and we write $\sigma \models \varphi$ for $\sigma, 1 \models \varphi$.

In Section 5 we will lift our results to trees and XPath. In order to do this it is convenient to introduce now a restriction of $\mathsf{LTL}_1^{\downarrow}$ such that with this restriction $\mathsf{LTL}_1^{\downarrow}$ corresponds to XPath. A formula of $\mathsf{LTL}_1^{\downarrow}$ is said to be *simple* if (i) there is at most one occurrence of $\uparrow$ within the scope of each occurrence of $\downarrow$ and, (ii) there is no negation between an occurrence of $\uparrow$ and its matching $\downarrow$, except maybe immediately before $\uparrow$. We denote by $\mathsf{sLTL}_1^{\downarrow}$ the fragment of $\mathsf{LTL}_1^{\downarrow}$ containing only simple formulas. The correspondance between $\mathsf{sLTL}_1^{\downarrow}$ and XPath will be made explicit in Proposition 1 of Section 5.

Faulty counter systems. For proving our lower bounds we will use a reduction from faulty counter automata that we describe here. A *counter automaton* (CA) with zero testing is a tuple $\langle \Sigma, Q, q_0, n, \delta, F \rangle$, where Σ is a finite alphabet, Q is a finite set of states, q_0 is the initial state, $n \in \mathbb{N}$ is the number of counters, $\delta \subset Q \times \Sigma \times L \times Q$ is the transition relation over the instruction set $L = \{\mathtt{inc}, \mathtt{dec}, \mathtt{ifzero}\} \times \{1, \ldots, n\}$, and $F \subset Q$ is the set of accepting states. A *counter valuation* is a function $v : \{1, \ldots, n\} \to \mathbb{N}$. An error-free run over $w \in \Sigma^*$ is a finite sequence $\langle q_0, v_0 \rangle \overset{w_0, \ell_0}{\to} \langle q_1, v_1 \rangle \overset{w_1, \ell_1}{\to} \cdots$ observing the standard interpretation of the instructions $\ell_0, \ell_1, \cdots$ ($\langle \mathtt{dec}, c \rangle$ can only be performed if c is nonzero), where $v_0, v_1, \ldots$ are counter valuations, v_0 assigns 0 to each counter, and $w = w_0 w_1 \ldots$ such that $w_i \in \Sigma \cup \{\varepsilon\}$. A run is *accepting* iff it ends with an accepting state.

A *Minsky* CA has error-free runs. For these automata, already with only two counters, finitary emptiness is in undecidable [15][1, Lemma 8]. An *Incrementing* CA (from now on ICA) is defined as a Minsky CA except that its runs may contain errors which increase one or more counters non-deterministically. We write that two valuations are in the relation $v \leq v'$ iff, for every counter c, $v(c) \leq v'(c)$. Runs of Incrementing CA are defined by replacing the relation $\overset{a,\ell}{\to}$ by $\overset{a,\ell}{\rightsquigarrow}$, where $\langle p, u \rangle \overset{a,\ell}{\rightsquigarrow} \langle q, v \rangle$ iff there exist valuations u', v' such that $u \leq u'$, $v' \leq v$ and $\langle p, u' \rangle \overset{a,\ell}{\to} \langle q, v' \rangle$.

In [9, Theorem 2.5] it is shown that the results of [18] and [17] on Channel Machines can be adapted to prove the following result.

Theorem 1. *Emptiness of* ICA *is decidable and not primitive recursive.*

3 The case of $\mathsf{sLTL}_1^{\downarrow}$

In this section we show that satisfiability of $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ is not primitive recursive over data words. We then prove that satisfiability is undecidable for $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$. In the next section we will show that the logic $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ is also not primitive recursive and $\mathsf{LTL}_1^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ undecidable. In Section 5 we will use the results of this section for proving lower bounds for fragments of XPath over data trees.

3.1 Lower bound for $\mathsf{sLTL}^{\downarrow}_{\mathbf{1}}(\mathsf{F_s})$.

Theorem 2. *Satisfiability of* $\mathsf{sLTL}^{\downarrow}_{1}(\mathsf{F_s})$ *on data words is not primitive recursive.*

Proof. We exhibit a PTIME reduction from the non-emptiness of ICA to formulas of $\mathsf{sLTL}^{\downarrow}_{1}(\mathsf{F_s})$. Let $C = \langle \Sigma, Q, q_0, n, \delta, F\rangle$ be an ICA.
Let $L = \{(\mathtt{inc}_i)_{1\le i\le n}, (\mathtt{dec}_i)_{1\le i\le n}, (\mathtt{ifzero}_i)_{1\le i\le n}\}$, and $\hat{\Sigma} = Q \times (\Sigma \cup \{\varepsilon\}) \times L \times Q$. We construct a formula $\varphi_C \in \mathsf{sLTL}^{\downarrow}_{1}(\mathsf{F_s})$ that is satisfied by a data word iff C accepts the word. We view a run of C of the form:

$$\langle q_0, v_0\rangle \overset{a,\mathtt{inc}_i}{\rightsquigarrow} \langle q_1, v_1\rangle \overset{b,\mathtt{dec}_j}{\rightsquigarrow} \langle q_2, v_2\rangle \overset{b,\mathtt{ifzero}_i}{\rightsquigarrow} \langle q_3, v_3\rangle \ \cdots$$

as a string in $\hat{\Sigma}$:

$$\langle q_0, a, \mathtt{inc}_i, q_1\rangle\langle q_1, b, \mathtt{dec}_j, q_2\rangle\langle q_2, b, \mathtt{ifzero}_i, q_3\rangle \ \cdots$$

The formula φ_C will force that any string that satisfies it codes a run of C. In order to do this, φ_C must ensure that

`(begin)` the string starts with q_0,
`(end)` the string ends with a state of F,
`(tran)` every symbol of $\hat{\Sigma}$ in the string corresponds to a transition of C,
`(chain)` the last component of a symbol of $\hat{\Sigma}$ is equal to the first component of the next symbol,
`(pair)` for each i, every symbol that contains $\mathtt{inc}_i$ occurring in the string to the left of a symbol containing $\mathtt{ifzero}_i$, can be paired with a symbol containing $\mathtt{dec}_i$ and occurring in between the $\mathtt{inc}_i$ and the $\mathtt{ifzero}_i$.

Before continuing let us comment on the `(pair)` condition. If we were coding runs of a perfect Minsky CA (ie, with no incremental errors), to the left of any position containing a $\mathtt{ifzero}_i$, we would require a perfect matching between $\mathtt{inc}_i$ and $\mathtt{dec}_i$ operations in order to make sure that the value of the counter i is indeed zero at the position of the test. But as we are dealing with ICA, we only need to check that each $\mathtt{inc}_i$ has an associated $\mathtt{dec}_i$ to its right and before the test for zero, but we do not enforce the converse, that all $\mathtt{dec}_i$ match an $\mathtt{inc}_i$. This is fortunate because this would require past navigational operators.

The first difficulty comes from the fact that `(pair)` is not a regular relation. The pairing will be obtained using data values. The second difficulty is to enforce `(chain)` without having access to the string successor relation. In order to simulate the successor relation we add extra symbols to the alphabet, with suitable associated data values.

Let $\Sigma' = \hat{\Sigma} \cup \{\mathtt{N}, \mathtt{\#}, \mathtt{@}\}$. The coding of a run consists in a succession of *blocks*. Each block is a sequence of 3 or 4 symbols, "c `N #`" or "c `@ N #`", with $c \in \hat{\Sigma}$. The data value associated to the c and `#` symbols of a block is the same and uniquely determines the block: no two blocks may have the same data value. The data value associated with the symbol `N` of a block is the data value of the next block. If a block contains a symbol c that codes a $\mathtt{inc}_i$ operation that is later in the string followed by a $\mathtt{ifzero}_i$, then this block contains a symbol `@` whose data value is the same as the data value of the block containing $\mathtt{dec}_i$ that is paired with c.

For instance in the example of Fig. 1 one can see four blocks b_1, b_2, b_3, b_4. Each of them starts with a symbol from $\hat{\Sigma}$ coding a transition of the ICA and ends with a `#` with the same data value marking the end of the block. Inside the block, the data value of `N` is the same as the data value of the next block. The data value of `@` corresponds to that of a future block. In this example c_1 must correspond to a $\texttt{inc}_i$ while c_3 to $\texttt{dec}_i$ and there must be a $\texttt{ifzero}_i$ somewhere to the remaining part of the word (say, b_5). Moreover c_2 can't be a $\texttt{ifzero}_i$ as otherwise the data value of the symbol `@` would refer to a block to the left of c_2.

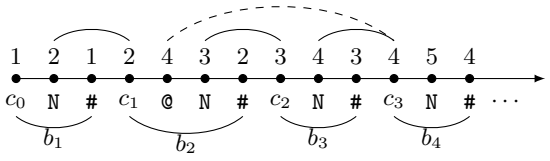

Fig. 1. Coding of an ICA run.

We now show that the coding depicted above can be enforced in $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$. By $\pi_1, \pi_2, \pi_3, \pi_4$ we denote the projection of each symbol of $\hat{\Sigma}$ into its corresponding component. To simplify the presentation we use the following abbreviations:

$$\hat{\sigma} \equiv \bigvee_{c \in \hat{\Sigma}} c \qquad \textsc{inc}(i) \equiv \bigvee_{c \in \hat{\Sigma}, \pi_3(c) = \texttt{inc}_i} c \qquad \textsc{inc} \equiv \bigvee_i \textsc{inc}(i)$$

$$\textsc{dec}(i) \equiv \bigvee_{c \in \hat{\Sigma}, \pi_3(c) = \texttt{dec}_i} c \qquad \textsc{last} \equiv \hat{\sigma} \wedge \neg \langle \rightarrow^+ [\hat{\sigma}] \rangle \qquad \textsc{iz}(i) \equiv \bigvee_{c \in \hat{\Sigma}, \pi_3(c) = \texttt{ifzero}_i} c$$

The formula φ_C that we construct is the conjunction of all the formula mentioned below.

Forcing the structure

$\mathsf{G}(\textsc{last} \Rightarrow \neg \mathsf{F_s} \top)$: The string ends with the last transition,

$\bigwedge_{c \in \{\texttt{N}, \texttt{@}, \#\}} \mathsf{G}(c \Rightarrow \neg(\downarrow \mathsf{F_s}(c \wedge \uparrow)))$: the data value associated to each `N`, `#` and `@` uniquely determines the occurrence of that symbol,

$\mathsf{G}((\hat{\sigma} \wedge \neg \textsc{last}) \Rightarrow (\downarrow \mathsf{F_s}(\texttt{N} \wedge \mathsf{F_s}(\# \wedge \uparrow))$: each occurrence of $\hat{\Sigma}$ (except the last one) is in a block that contains a `N` and then a `#`,

$\bigwedge_i \mathsf{G}\big((\textsc{inc}(i) \wedge \mathsf{F_s}(\textsc{iz}(i))) \Rightarrow \downarrow \mathsf{F_s}(\texttt{@} \wedge \mathsf{F_s}(\texttt{N} \wedge \mathsf{F_s}(\# \wedge \uparrow)))\big)$: every $\texttt{inc}_i$ block to the left of a $\texttt{ifzero}_i$ must have a `@` before the `N`,

$\mathsf{G}((\hat{\sigma} \wedge \neg \textsc{inc}) \Rightarrow \neg \downarrow \mathsf{F}(\texttt{@} \wedge \mathsf{F_s}(\# \wedge \uparrow)))$: only blocks `inc` are allowed to have a `@`,

$\bigwedge_{s \in \{\texttt{N}, \texttt{@}\}} \mathsf{G}\big(\hat{\sigma} \Rightarrow \neg(\downarrow \mathsf{F_s}(s \wedge \mathsf{F_s}(s \wedge \mathsf{F_s}(\# \wedge \uparrow)))))\big)$: there is at most one occurrence of `N` and `@` in each block,

$\bigwedge_{s \in \hat{\Sigma} \cup \{\#\}} \mathsf{G}(\hat{\sigma} \Rightarrow \neg \downarrow \mathsf{F_s}(s \wedge \mathsf{F_s}(\# \wedge \uparrow))$: there is exactly one symbol `#` and one symbol of $\hat{\Sigma}$ per block,

$\mathsf{G}(\texttt{N} \Rightarrow (\downarrow \mathsf{F_s}(\# \wedge \mathsf{F_s}(\hat{\sigma} \wedge \uparrow))))$: the data value of each symbol `N` points to a block to its right,

$\mathsf{G}(\texttt{N} \Rightarrow \neg \downarrow \mathsf{F_s}(\# \wedge \mathsf{F_s}(\# \wedge \mathsf{F_s}(\hat{\sigma} \wedge \uparrow))))$: in fact `N` has has to point be the next block.

Once the structure has the expected shape, we can enforce the run as follows. All the formulas below are based on the following trick. In a test of the form

$\varepsilon =\!\!\rightarrow^{+} [\mathtt{N}] \rightarrow^{+} [\mathtt{\#}]$ which is typically performed at a position of symbol $\hat{\Sigma}$, the last symbol # must have the same data value as the initial position. Hence, because of the structure above, both must be in the same block. Hence the middle symbol N must also be inside that block. From the structure we know that the data value of this N points to the next block. Therefore by adding in the middle bracket [N] a test of the form $(\varepsilon =\!\!\rightarrow^{+} [s])$ we can transfer some finite information from the current block to the next one. This gives the desired successor relation.

Forcing a run

(begin) $$\bigvee_{c\in\hat{\Sigma},\pi_1(c)=q_0} c$$

(end) $$\mathsf{F_s}\Big(\textsc{last} \wedge \bigvee_{c\in\hat{\Sigma},\pi_4(c)\in F} c\Big)$$

(tran) All elements used from $\hat{\Sigma}$ correspond to valid transitions. Let $\hat{\Sigma}^C$ be that set of transitions of C,

$$\mathsf{G}\Big(\bigwedge_{c\in\hat{\Sigma}\setminus\hat{\Sigma}^C} \neg c\Big)$$

(chain) For every $c \in \hat{\Sigma}$,

$$\mathsf{G}\Big(c \Rightarrow \big(\downarrow \mathsf{F_s}\big(\mathtt{N} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow) \wedge \bigvee_{\substack{d\in\hat{\Sigma},\\ \pi_4(c)=\pi_1(d)}} (\downarrow \mathsf{F_s}(d \wedge \uparrow))\big)\big)\Big)$$

(pair) We must first make sure that the block of the @ of an $\mathtt{inc}_k$ is matched with a block of a $\mathtt{dec}_k$:

$$\textstyle\bigwedge_k \mathsf{G}\big(\textsc{inc}(k) \Rightarrow \big(\neg(\downarrow \mathsf{F_s}(\mathtt{@} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))) \vee (\downarrow \mathsf{F_s}\big(\mathtt{@} \wedge (\downarrow \mathsf{F_s}(\textsc{dec}(k) \wedge \uparrow)) \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))\big)\big)\big)$$

Now, each time we are in a block with $\mathtt{inc}_k$ that afterward has a $\mathtt{ifzero}_k$ then:

– 1. The block must contain an @ element:

$$\textstyle\bigwedge_k \mathsf{G}\big(\textsc{inc}(k) \Rightarrow \big(\big(\downarrow \mathsf{F_s}(\mathtt{@} \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))\big) \vee \neg \mathsf{F_s}\textsc{iz}(k)\big)\big)$$

– 2. The data value of that @ element must point to a future block *before* any occurrence of $\mathtt{ifzero}_k$:

$$\textstyle\bigwedge_k \mathsf{G}\big(\textsc{inc}(k) \Rightarrow \neg(\downarrow \mathsf{F_s}\big(\mathtt{@} \wedge (\downarrow \mathsf{F_s}(\textsc{iz}(k) \wedge \mathsf{F_s} \uparrow)) \wedge \mathsf{F_s}(\mathtt{\#} \wedge \uparrow))\big)\big)$$

This concludes the construction of φ_C. The correctness of the construction is quite standard and can found in the appendix. □

3.2 Undecidability of $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$

We now consider $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ over data words. The extra modality can be used to code the run of a (non faulty) Minsky CA.

Theorem 3. *Satisfiability of* $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ *is undecidable over data words.*

Proof. Consider a Minsky CA C. We revisit the proof of Theorem 2. It is easy to see that we can enforce the absence of faulty increments during the run by asking that every $\mathtt{dec}_i$ element that appears is referenced by some previous $\mathtt{inc}_i$ block: $\bigwedge_i \mathsf{G}(\textsc{dec}(i) \Rightarrow (\varepsilon = {}^{+}\!\leftarrow [\mathtt{@}]))$. In this way we make sure that there are no $\mathtt{dec}$ that are unrelated to a corresponding $\mathtt{inc}$ and then the coding is that of a *perfect* (non faulty) run. □

4 The case of $\mathsf{LTL}_1^{\downarrow}$

In this section we lift the lower bounds of the previous section by considering the temporal operator F instead of $\mathsf{F_s}$. We can only do so by removing at the same time the restriction to simple formulas. Hence the results of this section cannot be applied to XPath. Notice that $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ and $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ are incomparable in term of expressive power. We do not know whether $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F})$, which is weaker than the two above mentioned logics, is already not primitive recursive. The results of this section improve the results of [9] which shows that satisfiability is not primitive recursive for $\mathsf{LTL}_1^{\downarrow}(\mathsf{X},\mathsf{F})$ and undecidable for $\mathsf{LTL}_1^{\downarrow}(\mathsf{X},\mathsf{F},\mathsf{F}^{-1})$.

Theorem 4. *Over data words,*

1. *Satisfiability of* $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ *is decidable and not primitive recursive.*
2. *Satisfiability of* $\mathsf{LTL}_1^{\downarrow}(\mathsf{F},\mathsf{F}^{-1})$ *is undecidable.*

Proof. We only prove Item 1. The proof of Item 2. is similar and can be found in the appendix.

Consider an ICA C and recall the coding of runs of C used in the proof of Theorem 2. Note that all data formulas used are of the form $\varepsilon = \alpha$ which are straightforward to code in $\mathsf{LTL}_1^{\downarrow}$. In the construction of the formula φ_C, whenever we use $\rightarrow^+$ with two different symbols on the left-hand side and the right-hand side, this axis can be equivalently replaced by the F temporal operator. This is the case in all formulas except in three places: (i) the formula saying that $\mathtt{N}$ should point to the next block contains $[\mathtt{\#}] \rightarrow^+ [\mathtt{\#}]$. But from the structure that is enforced, this can equivalently be replaced by $[\mathtt{\#}] \rightarrow^+ [\mathtt{N}] \rightarrow^+ [\mathtt{\#}]$. (ii) To enforce that each block contains at most one occurrence of symbols in $\hat{\Sigma} \cup \{\mathtt{N},\mathtt{\#},\mathtt{@}\}$. (iii) To enforce that no two symbols in $\hat{\Sigma} \cup \{\mathtt{N},\mathtt{\#},\mathtt{@}\}$ have the same data values.

In order to cope with (ii) and (iii), we use a slightly different coding for runs of C. This coding is the same as the one for the proof of Theorem 2 except that we allow succession of equal symbols, denoted as *group* in the sequel. Note that in a group two different occurrences of the same symbol in general may have different data values, as we can no longer enforce their distinctness. However, as we will see, we can enforce that the $\hat{\Sigma}$ group of elements of a block have all the same data value.

Hence a block now is either a group of $c \in \hat{\Sigma}$ followed by a group of $\mathtt{N}$ followed by a group of $\mathtt{\#}$ or the same with a group of $\mathtt{@}$ in between. A coding of a run is depicted in Figure 2.

This structure is enforced by modifying the formulas of the proof of Theorem 2 as follows.

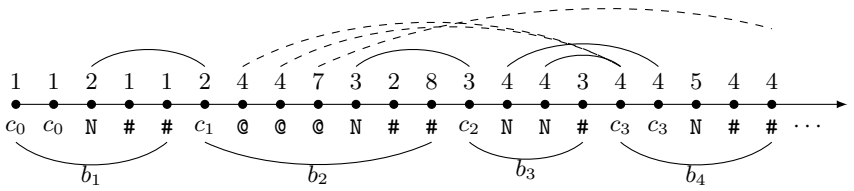

Fig. 2. $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ cannot avoid having repeated consecutive symbols.

(i) The formulas that limit the number of occurrences of symbols in a block are replaced by formulas *limiting the number of groups* in a block.
(ii) The formulas requiring that no two occurrences of a same symbol may have the same data values are replaced by formulas requiring that no two occurrences of a same symbol *in different groups* may have the same data values.
(iii) In all other formulas, $\rightarrow^+$ is replaced by F.
(iv) Finally, we ensure that although we may have repeated symbols inside a block, all symbols from $\hat{\Sigma}$ have the same data value.

$$\mathsf{G}(\hat{\sigma} \Rightarrow \neg \downarrow_1 \mathsf{F}((\hat{\sigma} \vee \#) \wedge \neg \uparrow_1 \wedge \mathsf{F}(\# \wedge \uparrow_1)))$$

Note that this implies that each N of a group must have the same data values as they all point to the next block. However there could still be @ symbols with different data values as depicted in Fig. 2.

The new sentences now imply, for instance, that:

1. Every position from a group of $c \in \hat{\Sigma}$ have the same data value which is later matched by a group of #.
2. Every position from a group of N has the same data value as a position $c \in \hat{\Sigma}$ of the *next* block (and then, it has only one data value).
3. Every position from a group @ has the same data value as a position $c \in \hat{\Sigma}$ of a block to its right. Note that the data values of two @ of the same group may correspond to the data values of symbols in different blocks. This is basically the main conceptual difference with the previous proof.

The proof of correctness of the construction is left to the reader. □

Note that in the previous proof we used the fact that $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$, although it has only one register, can make (in)equality tests several times throughout a path (as used in the formula of item (iv) in the proof) something that simple formulas and XPath can't do.

Two registers. When 2 registers are available the previous result can be adapted to code a (non faulty) Minsky CA. The proof can be found in the appendix.

Theorem 5. *Satisfiability of* $\mathsf{LTL}_2^{\downarrow}(\mathsf{F})$ *is undecidable over data words.*

5 Data trees and XPath

We now turn to data trees. An XML document can be seen as an unranked tree with *attributes* and data values in its nodes. While a data tree has only *one* data value per node, an XML document element may have *several* attributes each of which with an associated data value. All XPath fragments treated in this paper can force all elements of an XML document to have only one attribute. Therefore all hardness results in the present work hold also for the class of XML documents.

The logic XPath. XPath is a two-sorted language, with *path* expressions $(\alpha, \beta, \ldots)$ and *node* expressions $(\varphi, \psi, \ldots)$. These are defined by mutual recursion:

$$\alpha ::= \downarrow \mid \downarrow^+ \mid \uparrow \mid \uparrow^+ \mid \rightarrow \mid \rightarrow^+ \mid \leftarrow \mid {}^+\!\leftarrow \mid \varepsilon \mid \alpha\beta \mid \alpha \cup \beta \mid \alpha[\varphi]$$
$$\varphi ::= \sigma \mid \langle \alpha \rangle \mid \neg\varphi \mid \varphi \wedge \psi \mid \alpha = \beta \mid \alpha \neq \beta \quad (\sigma \in \Sigma)$$

A path expression essentially describes a traversal of the tree by using the axis: `child` ($\downarrow$), `descendant` ($\downarrow^+$), the `next-sibling` ($\rightarrow$), `following-sibling` ($\rightarrow^+$) and their inverses, with the ability to test at any stage for node conditions. Let t be a data tree with domain T and labeling function λ. The semantics of XPath is defined by induction is the usual intuitive way, we only give here some important steps:

$[\![\downarrow]\!]^t = \{(x, xi) \mid xi \in T\}$ $\quad$ $[\![\alpha\beta]\!]^t = \{(x, z) \mid \exists y.(x, y) \in [\![\alpha]\!]^t \wedge (y, z) \in [\![\beta]\!]^t\}$
$[\![\langle\alpha\rangle]\!]^t = \{x \in T \mid \exists y.(x, y) \in [\![\alpha]\!]^t\}$ $\quad$ $[\![\alpha[\varphi]]\!]^t = \{(x, y) \in [\![\alpha]\!]^t \mid y \in [\![\varphi]\!]^t\}$
$[\![\alpha^+]\!]^t =$ the transitive closure of $[\![\alpha]\!]^t$
$[\![\alpha = \beta]\!]^t = \{x \in T \mid \exists y, z.(x, y) \in [\![\alpha]\!]^t, (x, z) \in [\![\beta]\!]^t, \pi_2(\lambda(y)) = \pi_2(\lambda(z))\}$
$[\![\alpha \neq \beta]\!]^t = \{x \in T \mid \exists y, z.(x, y) \in [\![\alpha]\!]^t, (x, z) \in [\![\beta]\!]^t, \pi_2(\lambda(y)) \neq \pi_2(\lambda(z))\}$

A key property for using results of Section 3 is that simple formulas can be translated to XPath and back. The proof of this result is straightforward by induction on the formula.

Proposition 1. *Over data words,* $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ *and* $\mathsf{XPath}(\rightarrow^+, =)$ *have the same expressive power. The same hold for* $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ *and* $\mathsf{XPath}({}^+\!\leftarrow, \rightarrow^+, =)$. *Moreover, in both cases, the transformation from* $\mathsf{sLTL}_1^{\downarrow}$ *to* XPath *takes polynomial time while it takes exponential time in the other direction.*

We remark that there is a big difference between $\mathsf{XPath}(\rightarrow^+, =)$ over data words and $\mathsf{XPath}(\downarrow^+, =)$ over data trees. Indeed $\mathsf{XPath}(\downarrow^+, =)$ is closed under bisimulation and hence it cannot assume that the tree is a vertical path. As the string structure was essential in the proof of Theorem 2, the non primitive recursiveness of $\mathsf{XPath}(\rightarrow^+, =)$ over data words does not lift to $\mathsf{XPath}(\downarrow^+, =)$ over data trees. Actually one can show that satisfiability of $\mathsf{XPath}(\downarrow^+, =)$ is ExpTime-complete [11]. However if one consider the logic $\mathsf{XPath}(\downarrow^+, \rightarrow, =)$ then the axis $\rightarrow$ can be used to enforce a vertical path and therefore it follows from Theorem 2 and Proposition 1 that:

Corollary 1. *Satisfiability of* $\mathsf{XPath}(\downarrow^+, \rightarrow, =)$ *on data trees is not primitive recursive.*

Similarly in $\mathsf{XPath}(\downarrow^+, \uparrow^+, =)$ one can simulate a string by going down to a leaf using $\downarrow^+$ and then use the path from that leaf to the root as a string using $\uparrow^+$. Hence we also have:

Corollary 2. *Satisfiability of* $\mathsf{XPath}(\downarrow^+, \uparrow^+, =)$ *on data trees is not primitive recursive.*

Note that the decidability of $\mathsf{XPath}(\downarrow^+, \rightarrow, =)$ and of $\mathsf{XPath}(\downarrow^+, \uparrow^+, =)$ is still an open problem.

Open problem: Is $\mathsf{XPath}(\downarrow^+, \uparrow^+, =)$ decidable over data trees?

It would be interesting to know whether the strictness of the axis $\downarrow^+$ is necessary in the above two results. This boils down to know whether $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F})$ is already not primitive recursive over data words. Note that the proof of Theorem 4 use in a essential way the possibility to make (in)equality tests several times throughout a path. This is exactly what cannot be expressed in $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F})$.

Open problem: Is satisfiability of $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F})$ primitive recursive over data words?

We conclude with the following result which is a simple consequence of Theorem 3 and Proposition 1:

Corollary 3. *Satisfiability of* $\mathsf{XPath}({}^+\!\leftarrow, \rightarrow^+, =)$ *and of* $\mathsf{XPath}(\downarrow^+, \uparrow^+, \rightarrow, =)$ *over data trees is undecidable.*

6 Discussion

By [9] it is known that satisfiability of $\mathsf{LTL}_1^{\downarrow}(\mathsf{X}, \mathsf{F})$ with infinite data words is undecidable, as well as $\mathsf{LTL}_2^{\downarrow}(\mathsf{X}, \mathsf{F})$. The proof of Theorem 4 can be extended to code runs of ICA over infinite data words, which is known to be undecidable, to show that these results hold in the absence of X.

Theorem 6. *On infinite data words, the satisfiability problem of* $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ *is undecidable.*

Summary of results

In the table below we summarize the main results and some of the consequences we have mentioned. In this table, $\overline{\mathsf{PR}}$ stands for not primitive recursive, decidability unknown while $\mathsf{R} \setminus \mathsf{PR}$ stands for decidable and not primitive recursive.

Logic	Complexity	Details
$\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$	$\mathsf{R} \setminus \mathsf{PR}$	Thm 4 & [9]
$\mathsf{LTL}_1^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$	undecidable	Thm 4
$\mathsf{LTL}_2^{\downarrow}(\mathsf{F})$	undecidable	Thm 5
$\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$	$\mathsf{R} \setminus \mathsf{PR}$	Thm 2 & [9]
$\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$	undecidable	Thm 3
$\mathsf{XPath}(\downarrow^+, \rightarrow, =)$	$\overline{\mathsf{PR}}$	Cor 1
$\mathsf{XPath}(\downarrow^+, \uparrow^+, =)$	$\overline{\mathsf{PR}}$	Cor 2
$\mathsf{XPath}(\downarrow^+, \uparrow^+, \rightarrow, =)$	undecidable	Cor 3

References

1. Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–204, 1994.
2. Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.
3. Henrik Björklund, Wim Martens, and Thomas Schwentick. Optimizing conjunctive queries over trees using schema information. In *MFCS*, pages 132–143, 2008.
4. Mikolaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. In *PODS*, pages 10–19, 2006.
5. Mikolaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.
6. Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic approach to data languages and timed languages. *Inf. Comput.*, 182(2):137–162, 2003.
7. J. Clark and S. DeRose. XML path language (XPath). Website, 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.
8. Claire David. Complexity of data tree patterns over XML documents. In *MFCS*, pages 278–289, 2008.
9. Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. In *LICS*, pages 17–26. IEEE Computer Society Press, 2006.
10. Stéphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL: Decidability and complexity. In *TIME*, pages 113–121, 2005.
11. Diego Figueira. Satisfiability of downward XPath with data equality tests. In *PODS*, Providence, Rhode Island, USA, 2009. ACM Press.
12. Floris Geerts and Wenfei Fan. Satisfiability of XPath queries with sibling axes. In *DBPL*, volume 3774, pages 122–137. Springer, 2005.
13. Marcin Jurdziński and Ranko Lazić. Alternating automata on data trees and XPath satisfiability. *CoRR*, abs/0805.0330, 2008.
14. Alexei Lisitsa and Igor Potapov. Temporal logic with predicate lambda-abstraction. In *TIME*, pages 147–155, 2005.
15. Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.
16. A. Nakamura and H. Ono. On the size of refutation Kripke models for some linear modal and tense logics. *SL*, 39(4):325–333, 1980.
17. J. Ouaknine and J. Worrell. On Metric temporal logic and faulty Turing machines. In *FoSSACS*, volume 3921, pages 217–230, 2006.
18. Philippe Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Information Processing Letters*, 83(5):251–261, September 2002.

A Missing proofs

A.1 Correctness of the construction of the proof of Theorem 2

Given an ICA C, the proof exhibited a formula $\varphi_C \in \mathsf{XPath}(\rightarrow^*, =)$ that is satisfied by a data word iff C accepts the word. We show here that the construction is correct.

From data word to an accepting ICA run Let w be a data word satisfying φ_C. Let $B_1 B_2 \cdots B_m$ be the decomposition of w into blocks as described above. Let $c_i = \langle q_i, a_i, \ell_i, q_{i+1} \rangle$ be the element of B_i in $\hat{\Sigma}$. We show that $w' = a_1 a_2 \cdots a_m$ is accepted by C. We construct a run of C on w' such that C is in state q_i before reading letter a_i of w' and execute the transitions c_i. By `(tran)` each one of these are valid transitions of the ICA, and `(chain)` ensures that the sequence is consistent. `(begin)` takes care of the initialization condition and `(end)` of the accepting condition. It remains to construct the valuation $v_1, \ldots, v_{m+1}$ of the counter at each step such that we have $\langle q_i, v_i \rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1} \rangle$ as desired.

As expected v_1 is the zero valuation. We construct the rest of the valuations by induction, simulating a perfect counter machine and introducing incrementing errors in a lazy way, whenever necessary. At step i, given a transition $\langle q_i, a_i, \ell_i, q_{i+1} \rangle$ and the current valuation v_i we set v_{i+1} such that $\langle q_i, v_i \rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1} \rangle$.

- If ℓ_i is $\mathtt{inc}_k$, then $v_{i+1} := v_i[k \mapsto v_i(k)+1]$ and clearly $\langle q_i, v_i \rangle \overset{a_i,\ell_i}{\rightarrow} \langle q_{i+1}, v_{i+1} \rangle$.
- If ℓ_i is $\mathtt{dec}_k$ then:
  - If $v_i(k) > 0$ then $v_{i+1} := v_i[k \mapsto v_i(k) - 1]$.
  - If $v_i(k) = 0$, then $v_{i+1} := v_i$.

  In both cases one can verify that $\langle q_i, v_i \rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1} \rangle$, the transition being with an incrementing error of 1 in the second case.
- If ℓ_i is $\mathtt{ifzero}_k$, we set $v_{i+1} := v_i$. To ensure that $\langle q_i, v_i \rangle \overset{a_i,\ell_i}{\rightsquigarrow} \langle q_{i+1}, v_{i+1} \rangle$ we must show that necessarily $v_i(k) = 0$. This is a consequence of condition `pair`. This condition enforces that each $\mathtt{inc}_k$ must be paired with a unique $\mathtt{dec}_k$ to its right and before the $\mathtt{ifzero}_k$ test. Hence the lazy strategy above ensures that each increment is decremented later on and before the zero test.

From an accepting ICA run to data word Given an accepting run of the ICA $\langle q_1, v_1 \rangle \overset{a_1,\ell_1}{\rightsquigarrow} \langle q_2, v_2 \rangle \overset{a_2,\ell_2}{\rightsquigarrow} \ldots \overset{a_m,\ell_m}{\rightsquigarrow} \langle q_{m+1}, v_{m+1} \rangle$ we construct a data word w verifying φ_C as follows. Let c_i be $\langle q_i, a_i, \ell_i, q_{i+1} \rangle$. w consists of a concatenation of m blocks $B_1 B_2 \cdots B_m$ each one defined:

- If $i = m$, B_i is the word c_i with data value m.
- If $i < m$, and $\ell_i \in \{\mathtt{dec}_k, \mathtt{ifzero}_k\}$, then set B_i to $c_i\mathtt{N\#}$ with respective data values $i, i+1, i$.

$$\langle c_i, i \rangle \langle \mathtt{N}, i+1 \rangle \langle \mathtt{\#}, i \rangle$$

– If $i < m$ and ℓ_i is $\texttt{inc}_k$. Consider the minimal $d > i$ such that $v_{d+1}(k) = v_i(k)$ and set B_i to
  - c_i `@ N #` with respective data values $i, d, i+1, i$ if d exists
  - c_i `N #` with respective data values $i, i+1, i$ if not.

This construction assigns the unique data value i to each block B_i, and set the data value of each symbol `N` to the data value of the next block B_{i+1}. The constraints `(begin)`, `(end)`, `(tran)`, `(chain)` are obviously true. It remains to verify the constraints on `@`: they must have a unique data value, must point to a corresponding `dec` instruction that occur before any corresponding zero test, and no two `@` may point to the same `dec` instruction. Consider a position i such that ℓ_i is $\texttt{inc}_k$ with a subsequent zero test $\texttt{ifzero}_k$ at position $j > i$. Because the zero test is correct, the increment of counter k made at step i must be decremented at some position between i and j. Hence there is a $i < d < j$ such that $v_{d+1}(k) = v_i(k)$. By construction the data value of the `@` symbol of B_i is the minimal such d. By minimality, c_d must be a $\texttt{dec}_k$ instruction. Assume now that at position $i' < i$ there is also a $\texttt{inc}_k$ instruction. The data value of the `@` symbol of $B_{i'}$ cannot be d. Because if this would be the case then $v_{i'}(k) = v_{d+1}(k) = v_i(k)$ and d would not be minimal for the position i'. Hence `(pair)` is also satisfied and $w \models \varphi_C$.

A.2 Proof of Theorem 4.2

Recall the statement.
Theorem 4.2 Over data words, Satisfiability of $\mathsf{LTL}_1^{\downarrow}(\mathsf{F}, \mathsf{F}^{-1})$ is undecidable.

Proof (sketch). Consider a Minsky Counter Automaton C. We revisit the previous proof. We modify the coding of a run of C by also inserting a group of symbols `@` in each `dec` block. Using F^{-1}, we add formulas ensuring that *every* block containing a `dec` instruction also contains a group of `@` symbols such that their data values refer to previous blocks containing a matching `inc` instruction. We also make sure that two `@` from different blocks have different data values. All this can be done by taking the F^{-1} counterpart of the formulas we constructed with F.

We show that the new formula φ_C is satisfied by a data word iff C accept the word. Constructing a data word that satisfies φ_C from a word accepted by C is done as in the proof of Theorem 2.

For the other direction one needs to show that from a word that satisfies φ_C, the corresponding word with the obvious run and obvious valuations of the counter is accepting for C. One can verify that the new extra conditions do imply that no null counter is ever decreased and each zero test is correct, based on the fact that for every $\texttt{dec}_i$ there must be at least one $\texttt{inc}_i$ to its left, and for every $\texttt{inc}_i$ there must have a related $\texttt{dec}_i$ before any $\texttt{ifzero}_i$ test.

A.3 Proof of Theorem 5

First recall the statement of Theorem 5:

Theorem 5 Satisfiability of $\mathsf{LTL}_2^{\downarrow}(\mathsf{F})$ is undecidable over data words.

Proof. We adapt the proof of part 1. of Theorem 4 using ideas already present in [9,14] for coding Minsky CA with only forward temporal operators. Fix a two counter machine $C = \langle \Sigma, Q, q_0, 2, \delta, F \rangle$. The main idea is to modify the coding of Theorem 4 by adding in each block a symbol h_1 whose data value is supposed to be the one of the last $\mathtt{inc}_1$ that has previously occurred, a symbol l_1 whose data value is supposed to be the one of the last $\mathtt{inc}_1$ that has not been decreased yet and similarly with h_2 and l_2 for the second counter. A zero test of counter 1 can then only occur in a block where the data values of h_1 and l_1 match. The coding is depicted in Figure 3.

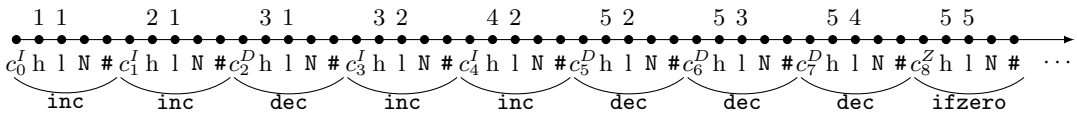

Fig. 3. Coding of a 1-counter machine run. Only data values of the `h` and `l` symbols are represented for the sake of clarity. Only one symbol per group is also represented. Elements depicted as c_i^I, c_i^D and c_i^Z correspond, respectively, to increment, decrement and zero testing symbols of $\hat{\Sigma}$.

Based on these ideas, the formula φ_C that we construct makes sure that:

(i) *The structure.* We enforce a structure that is a sequence of blocks of the form "$\langle q, w, l, q' \rangle \mathrm{h}_1 \mathrm{l}_1 \mathrm{h}_2 \mathrm{l}_2 \mathtt{N\#}$", with the possibility that there are repeated consecutive symbols. This can be done in the same way we did before.
As for the previous codings, we demand that in each block, for every element in the first group of $\hat{\Sigma}$ symbols there is an element in the last group of symbols `#` with the same data value. This is enforce with a formula of $\mathsf{LTL}_1^{\downarrow}(\mathsf{F})$ as before.
With the help of the second register we can now enforce that each group of symbols has the same data value, for all symbol s we have the formula:

$$\mathsf{G}\big(\hat{\sigma} \Rightarrow \downarrow_1 \neg \mathsf{F}\big(s \wedge \downarrow_2 \mathsf{F}\,(s \wedge \neg \uparrow_2 \wedge \mathsf{F}(\# \wedge \uparrow_1))\big)\big)$$

(ii) Each $\langle q, w, l, q' \rangle$ occurring in the string is in δ. For the first one $q = q_0$, and for the last one $q' \in F$. For any $\langle q, w, l, q' \rangle$ and $\langle q'', w', l', q''' \rangle$ in consecutive blocks, $q' = q''$. This can be checked just as in the previous codings.

(iii) In the initial block, the data values of h_1 and l_1 are the same and the data values of h_2 and l_2 are the same (we only show the formula for h_1 and l_1):

$$\downarrow_1 \neg \mathsf{F}\big(\mathrm{h}_1 \wedge \downarrow_2 \mathsf{F}(\mathrm{l}_1 \wedge \neg \uparrow_2 \wedge \mathsf{F}(\# \wedge \uparrow_1))\big)$$

(iv) In any block immediately after a `ifzero` instruction, the data values of h_1,l_1,h_2,l_2 are identical to those of the `ifzero` instruction. We only give the formula for $\texttt{ifzero}_1$ instruction with the h_1 symbol.
$\neg \mathsf{F}\big(\textsc{iz}(1) \wedge \downarrow_1 \mathsf{F}\big(h_1 \wedge \downarrow_2 \mathsf{F}(\psi \wedge \mathsf{F}(\# \wedge \uparrow_1))\big)\big)$
where $\psi \equiv \mathtt{N} \wedge \downarrow_1 \mathsf{F}(h_1 \wedge \neg \uparrow_2 \wedge \mathsf{F}(\# \wedge \uparrow_1))$

(v) In any block immediately after an $\texttt{inc}_1$ instruction, h_1 is not in the same class as any preceding h_1 (stated in the formula below) and l_1, h_2, l_2 are in the same class as in the previous block (this can be stated using a formula similar to the one for for item (iv)).
$\neg \mathsf{F}\big(h_1 \wedge \downarrow_1 \mathsf{F}\big(\textsc{inc}(1) \wedge \downarrow_2 \wedge \mathsf{F}(\varphi \wedge \mathsf{F}(\# \wedge \uparrow_2))\big)\big)$
where $\varphi \equiv \mathtt{N} \wedge \downarrow_2 \mathsf{F}\big(\uparrow_2 \wedge \mathsf{F}(h_1 \wedge \uparrow_1 \wedge \mathsf{F}(\# \wedge \uparrow_2))\big)$
We have of course corresponding formulas for blocks with $\texttt{inc}_2$ instructions.

(vi) In any $\texttt{dec}_1$ block, h_1 and l_1 have different data values.

$$\neg \mathsf{F}\big(\textsc{dec}(1) \wedge \downarrow_1 \mathsf{F}\big(h_1 \wedge \downarrow_2 \mathsf{F}(l_1 \wedge \uparrow_2 \wedge \mathsf{F}(\# \wedge \uparrow_1))\big)\big)$$

We have a corresponding formula for blocks with a $\texttt{dec}_2$ instruction.

(vii) In any $\texttt{ifzero}_1$ block, h_1 and l_1 have the same data value. This can be expressed with a formula as above. We have a corresponding formula for $\texttt{ifzero}_2$ blocks.

(viii) In any block immediately after a $\texttt{dec}_1$ block B,
– h_1 is in the same class as the previous h_1, and
– l_1 is in the same class as the h_1 occurring in a block immediately after the rightmost block (occurring before B) with a h_1 that is in the same class as the l_1 of B.
– h_2 and l_2 are in the same class as their corresponding symbol of B.
The first and third conditions can be expressed using formulas similar to those above. The difficult part is to enforce the second condition. For this we introduce some macros:

$$\begin{aligned}
\mathrm{InBlock}_i(\varphi) &\equiv \downarrow_i \mathsf{F}(\varphi \wedge \mathsf{F}(\# \wedge \uparrow_i)) \\
\mathrm{NextBlock}_i(\varphi) &\equiv \downarrow_i \mathsf{F}(\mathtt{N} \wedge \mathsf{F}(\# \wedge \uparrow_i) \wedge \\
&\qquad \downarrow_i \mathsf{F}(\hat{\sigma} \wedge \varphi \wedge \mathsf{F}(\# \wedge \uparrow_i))) \\
\mathrm{InNextBlock}_i(\varphi) &\equiv \mathrm{NextBlock}_i(\mathrm{InBlock}_i(\varphi))
\end{aligned}$$

The second condition can then be stated:

$$\neg \mathsf{F}\Big(\hat{\sigma} \wedge \downarrow_2 \mathsf{F}\Big(h_1 \wedge \downarrow_1 \mathsf{F}\big(\mathtt{N} \wedge \mathsf{F}(\# \wedge \uparrow_2) \wedge \downarrow_2 \mathsf{F}(\hat{\sigma} \wedge \uparrow_2 \wedge \mathsf{F}(h_1 \wedge \neg \uparrow_1 \wedge \downarrow_2 \varphi))\big)\Big)\Big)$$

where $\varphi \equiv \mathsf{F}\big(\textsc{dec}(i) \wedge \mathrm{InBlock}_2(l_1 \wedge \uparrow_1) \wedge \mathrm{InNextBlock}_1(l_1 \wedge \neg \uparrow_2)\big)$
We have a corresponding formula for blocks following a $\texttt{dec}_2$ instruction.

□

A.4 Proof of Proposition 1

Proposition 1 Over data words, $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s})$ and $\mathsf{XPath}(\rightarrow^+, =)$ have the same expressive power. The same hold for $\mathsf{sLTL}_1^{\downarrow}(\mathsf{F_s}, \mathsf{F_s}^{-1})$ and $\mathsf{XPath}({}^+\!\leftarrow, \rightarrow^+, =)$. Moreover, in both cases, the transformation from $\mathsf{sLTL}_1^{\downarrow}$ to XPath takes polynomial time while it takes exponential time in the other direction.

Proof (sketch). The translation from $\mathsf{sLTL}_1^{\downarrow}$ to XPath is straightforward by induction. One then naturally obtain equality tests of the form $\varepsilon = \alpha$ or $\varepsilon \neq \alpha$ depending on whether the unique possible negation is present or not.

The other translation is also by induction. The non trivial part concerns the translation of node expressions of the form $\alpha = \beta$. Over data words, one of the path expressions α or β must end first, say α. In this case the node expression can be decomposed into a disjunction of (exponentially many) expressions that first start with a path expression that *merge* α with the part β_1 of β that is common to with α, followed by a test of the form $\varepsilon = \beta_2$, where $\beta = \beta_1\beta_2$. Tests of the form $\varepsilon = \beta_2$ are the immediate to translate into $\mathsf{sLTL}_1^{\downarrow}$.

B Semantic of XPath

Recall that XPath is a two-sorted language, with *path* expressions ($\alpha, \beta, \ldots$) and *node* expressions ($\varphi, \psi, \ldots$). These are defined by mutual recursion:

$$\begin{aligned}\alpha &::= \downarrow \mid \downarrow^+ \mid \uparrow \mid \uparrow^+ \mid \rightarrow \mid \rightarrow^+ \mid \leftarrow \mid {}^+\!\leftarrow \mid \varepsilon \mid \alpha\beta \mid \alpha \cup \beta \mid \alpha[\varphi] \\ \varphi &::= \sigma \mid \langle\alpha\rangle \mid \neg\varphi \mid \varphi \wedge \psi \mid \alpha = \beta \mid \alpha \neq \beta \quad (\sigma \in \Sigma)\end{aligned}$$

A path expression essentially describes a traversal of the tree by using the axis: `child` ($\downarrow$), `descendant` ($\downarrow^+$), the `next-sibling` ($\rightarrow$), `following-sibling` ($\rightarrow^+$) and their inverses, with the ability to test at any stage for node conditions. Let t be a data tree with domain T and labeling function λ. The semantics of XPath is defined by induction as follows:

$$\begin{aligned}[\![\downarrow]\!]^t &= \{(x, xi) \mid xi \in T\} \\ [\![\uparrow]\!]^t &= \{(xi, x) \mid xi \in T\} \\ [\![\leftarrow]\!]^t &= \{(xi, x(i-1)) \mid x(i-1) \in T\} \\ [\![\rightarrow]\!]^t &= \{(xi, x(i+1)) \mid x(i+1) \in T\} \\ [\![\alpha^+]\!]^t &= \text{the transitive closure of } [\![\alpha]\!]^t \\ [\![\varepsilon]\!]^t &= \{(x, x) \mid x \in T\} \\ [\![\alpha\beta]\!]^t &= \{(x, z) \mid \exists y.(x, y) \in [\![\alpha]\!]^t \wedge (y, z) \in [\![\beta]\!]^t\} \\ [\![\alpha \cup \beta]\!]^t &= [\![\alpha]\!]^t \cup [\![\beta]\!]^t \\ [\![\alpha[\varphi]]\!]^t &= \{(x, y) \in [\![\alpha]\!]^t \mid y \in [\![\varphi]\!]^t\} \\ [\![\sigma]\!]^t &= \{x \in T \mid \pi_1(\lambda(x)) = \sigma\} \\ [\![\langle\alpha\rangle]\!]^t &= \{x \in T \mid \exists y.(x, y) \in [\![\alpha]\!]^t\}\end{aligned}$$

$$
\begin{aligned}
[\![\neg\varphi]\!]^t &= T \setminus [\![\varphi]\!]^t \\
[\![\varphi \wedge \psi]\!]^t &= [\![\varphi]\!]^t \cap [\![\psi]\!]^t \\
[\![\alpha = \beta]\!]^t &= \{x \in T \mid \exists y, z.(x, y) \in [\![\alpha]\!]^t, \\
&\qquad (x, z) \in [\![\beta]\!]^t, \pi_2(\lambda(y)) = \pi_2(\lambda(z))\} \\
[\![\alpha \neq \beta]\!]^t &= \{x \in T \mid \exists y, z.(x, y) \in [\![\alpha]\!]^t, \\
&\qquad (x, z) \in [\![\beta]\!]^t, \pi_2(\lambda(y)) \neq \pi_2(\lambda(z))\}
\end{aligned}
$$