Learners Based on Transducers*

Sanjay Jain[1], Shao Ning Kuek[2], Eric Martin[3] and Frank Stephan[1,2]

[1] Department of Computer Science, National University of Singapore,
13 Computing Drive, COM1, Singapore 117417, Republic of Singapore,
sanjay@comp.nus.edu.sg and fstephan@comp.nus.edu.sg
[2] Department of Mathematics, National University of Singapore,
10 Lower Kent Ridge Road, S17, Singapore 119076, Republic of Singapore,
shaoning@u.nus.edu
[3] School of Computer Science and Engineering,
The University of New South Wales, Sydney NSW 2052, Australia,
eric.martin@unsw.edu.au

Abstract The learners considered in this paper process an infinite text of data containing all members of the set to be learnt in cycles, one word in each cycle, and maintain as a long term memory a string which provides all internal data the learner can use in the next cycle. Updating of these strings is usually done by either recursive or automatic learners. The present work looks at transduced learners, which sit in-between automatic and recursive learners in terms of computing power. The results include that transduced learners can learn all learnable automatic families with memory exponential in the size of the longest input seen so far. Furthermore, there is a hierarchy based on the memory-allowance: if n is the size of the largest datum seen so far, then for all $k \geq 1$, memory n^{k+1} allows one to learn more automatic families than memory n^k. This result stands in contrast to the situation of automatic learners, as it is unknown whether every learnable automatic family can be learnt by such a learner using word-sized memory, that is, memory not exceeding, by more than a given constant, the length of the longest word seen so far. Further results shed light on which circumstances allow one to impose that transduced learners be consistent, conservative or iterative. The main result of this kind is that all learnable automatic families have a consistent and conservative transduced learner.

1 Introduction

Gold [16] introduced the model of learning in the limit from positive data. Subsequent research in inductive inference [1,5,25,35,40,42,43] studied variations on this model. The basic features of Gold's model are the following. Given a finite alphabet Σ, let L be a subset of Σ^*, that is,

* S. Jain and F. Stephan are supported in part by the Singapore Ministry of Education Academic Research Fund grants R146-000-181-112 and MOE2016-T2-1-019 / R146-000-234-112. Furthermore, S. Jain is supported in part by NUS grant C252-000-087-001. F. Stephan did part of the work while on sabbatical leave to UNSW Sydney. S. N. Kuek worked on topics of this paper for his Final Year Project [33]; furthermore, selected results of this work have been presented at the conference LATA 2018 [22].

a language (over Σ). The learner gets as input a *text for* L, namely, a sequence of strings x_0, x_1, ... that contains all members but no non-member of L. As output, the learner conjectures a sequence of indices e_0, e_1, ... as its hypotheses on what the input language might be. More precisely, the learner works with a *hypothesis space* $\{H_e : e \in I\}$, where I is the set of possible indices, that has representatives for every possible learning task. If the sequence of hypotheses converges to an index e for the language L (that is, $H_e = L$), then the learner is said to have learnt the input language from the text. The learner learns a language L if it learns it from all texts for L. It learns a class $\mathcal{L}$ of languages if it learns all languages in $\mathcal{L}$. To measure the complexity of a learner, it is convenient to consider the learner as operating in cycles: it starts with hypothesis e_0 and in the n-th cycle, it gets the datum x_n and conjectures the hypothesis e_{n+1}. Freivalds, Kinber and Smith [15] and Kinber and Stephan [32] imposed the condition that between two cycles, the learner remembers only part of its previous inputs and works via some (long term) memory, which can be restricted. Thus, the complexity of learners can be measured in terms of two parameters: (a) the computational complexity of mapping the old memory and input datum to the new memory and hypothesis and (b) the length of the memory as a function of the length of the longest example seen so far.

Fundamental choices for (a) are: recursive learners (the mapping can be computed by a Turing machine), transduced learners (the mapping can be computed by a finite transducer) and automatic learners (the mapping is an automatic function). Pitt [43] showed that many complexity-theoretic restrictions — like requiring that the update time in each cycle be carried out in time polynomial in the sum of the lengths of all inputs seen so far — do not give a real restriction for most learning criteria from classical inductive inference. Automatic learners are more severely restricted and offer an interesting object of study [9,23]. In particular, it is natural to investigate the target classes that are represented in an automata-theoretic framework, namely, the automatic families [24]. These offer a representation that an automatic learner can easily handle. It is also natural to impose that hypothesis spaces be themselves automatic families containing the class to be learnt. It turns out that certain such families are learnable by a recursive learner, but not by an automatic learner [23].

The inability to memorise all past data is a major weakness of automatic learners and is exploited by many non-learnability proofs. In the absence of timing constraints for convergence-speed, and since repeated applications of automatic updates allow one to compute any recursive function, it is indeed the inability to memorise all observed data that limits learnability. Therefore the Turing completeness of iterated automatic functions alone does not lead to learnability of all automatic classes. Transducers can overcome this problem simply by appending every new datum to the memory. The interesting question is therefore whether additional constraints, on memory-size for example, can be met in the general case. W.r.t. the learnability of automatic families from fat text (with infinitely many occurrences of each datum), Jain, Luo and Stephan [23] showed that automatic and recursive learners have the same power. Their memory can even be restricted to the length of the longest datum seen so far, the so-called *word length memory limitation*. Thus, the weakness of automatic learners is mainly due to the impossibility to memorise the relevant information, and it is natural to question how much memory a transduced learner needs to be able to learn all automatic or transduced families which can be learnt from text.

In the model of transduced learners, the update mapping of the learner is computed by a non-deterministic transducer which on all accepting runs, produces the same outputs for the same inputs; Berstel [3] and other relevant literature call these functional transducers. Both inputs (old memory and current datum) are read independently, and both outputs (new memory and hypothesis) are written independently. This independence makes transduced learners more powerful than automatic ones.

For (b), Freivalds, Kinber and Smith [15] imposed that learners operate in cycles and only remember, from one cycle to the next, information recorded in a (long term) memory, with restrictions on its length. For automatic and transduced learners, the memory is a string over a fixed alphabet, that may depend on the learner, whose size is measured by the length of the string [15,23,32]. In the subfield of automatic learning, this way of restricting the memory led to fruitful findings, still leaving one major question open: is it truly restrictive to bound the memory by the length of the longest datum seen so far? The present work provides a positive answer for transduced learners, and moreover demonstrates that there is a learning hierarchy based on the memory sizes as a function of the length of the longest datum seen so far. Here, polynomials and exponential functions of various degrees and exponents, respectively, are the most prominent memory bounds.

2 Plan of the paper

Section 3 gives the general notation and concept definitions. Section 4 provides more particularily the background for automatic and transduced functions and relations.

Section 5 is on transduced learners which use wordsize memory. Proposition 3 gives an example of a transduced class which (i) has an automatic conservative and iterative learner using word-sized memory, (ii) has no consistent automatic learner, but (iii) has a conservative and consistent iterative transduced learner with word-sized memory. Proposition 5 provides an example of a transduced class which has a transduced learner with word-sized memory, but has no automatic learner.

Section 6 studies the memory complexity of learning automatic families. For transduced learners, Proposition 7 provides a tool to derive a lower bound on the memory needed to learn automatic families of a certain type. Proposition 8 exhibits an automatic class which can be learnt with quadratic memory by a transduced learner, but which can be learnt by neither an automatic learner nor by any recursive learner using subquadratic memory. Corollaries 9 and 10 establish the memory hierarchy. Proposition 14 shows that every learnable automatic class can be learnt by some transduced learner with exponential memory.

Section 7 generalises Proposition 14 and investigates a general connection between the space needed by recursive learners and that needed by transduced learners for learning uniformly recursive and transduced classes. In particular, it is shown that all learnable transduced classes have a transduced learner whose space usage is, up to an exponential term, the same as a recursive learner for this transduced class; here the memory usage of the recursive learner is the space usage on the Turing machine tape of a Turing machine carrying out all its memory operations and internal computations using this tape. In particular, Corollary 17 summarises the preceding results and states that every explanatorily learnable transduced class can be learnt by a transduced learner with some memory bounded by a recursive function. Note that such a recursive bound does not exist for the class of all cosingleton subsets of the natural numbers plus all sets $\{0, 1, \ldots, x-1\} \cup \{x+1, x+2, \ldots, x+y\}$ where x is in some fixed nonrecursive r.e. set A and y is bounded by the time to enumerate x into A, though this class has a recursive learner. Thus the existence of even a recursive bound for transduced classes is nontrivial.

Section 8 shows that all learnable automatic families have a consistent and conservative learner which uses from time to time a special symbol, ?. Furthermore, if an automatic class has a strong-monotonical learner, one can make this learner also transduced, consistent and conservative. Further results in this section relate these notions to iterative learning.

3 Preliminaries

This section introduces the notation used in the paper as well as the basics of learning theory, starting with the very basic notation for numbers and strings. Let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, \ldots\}$ and Σ denote a finite alphabet (set of symbols), for example, the set $\{0, 1, \ldots, 9\}$ of decimal digits or the set $\{0, 1\}$ of binary digits. Furthermore, ε denotes the empty string and Σ^* denotes the set of all strings (words) over Σ. A language is a subset of Σ^* for some finite alphabet Σ. Concatenation of strings u and v consists of putting v at the end of u, for example, the concatenation of 0011 and 2233 is 00112233; it is denoted by $u \cdot v$, or just uv when the context makes it clear. A string u of length n can be considered as a function from $\{0, 1, \ldots, n-1\}$ to Σ, with $u(i)$ the $(i+1)$-th symbol in u. Given a set of words S and $n \in \mathbb{N}$, let $S_{\leq n}$ and $S_{<n}$ denote $\{x \in S : |x| \leq n\}$ and $\{x \in S : |x| < n\}$, respectively. One assumes Σ comes with an ordering $<$. Then $u <_{lex} v$, read "u is lexicographically before v", means that either v is the concatenation of u with some nonempty string or both strings differ on their common domain and the first index i for which $u(i) \neq v(i)$ satisfies $u(i) < v(i)$. Length lexicographic order between strings is defined using the length of strings and lexicographic order: $u <_{ll} v$ if either $|u| < |v|$ or $|u| = |v|$ and $u <_{lex} v$.

Convolution of two words v and w, denoted $conv(v, w)$, is obtained by first appending at the end of the shorter word a special character, say #, until both words have the same length. In some cases, alignment is at the end of the words (on the right hand side) rather than at the beginning; $endconv(v, w)$ denotes the version of convolution where appending of # is done at the front. One then combines (for either form of convolution) the characters at position i, namely, v_i and w_i, to a new character $u_i = \binom{v(i)}{w(i)}$. For example,

$$conv(010, 10010) = \binom{0}{1}\binom{1}{0}\binom{0}{0}\binom{\#}{1}\binom{\#}{0}$$

and

$$endconv(010, 10010) = \binom{\#}{1}\binom{\#}{0}\binom{0}{0}\binom{1}{1}\binom{0}{0}.$$

Convolutions of several words can be defined similarly.

3.1 Automatic Relations and Functions

The interested reader might consult any standard textbook on automata theory [19,29] or lecture notes on the internet [49] for an introduction to finite automata and regular languages. Automatic structures bring this concept from mere sets to relations and functions by defining it either through convolutions (as done below) or by finite automata reading several inputs at the same speed, feeding a special symbol to the automaton after an input word has been completely read. This field was independently created by Hodgson [17,18] before 1976, by Khoussainov and Nerode in their influential 1995-paper [28] and by Blumensath and Grädel at the end of the last century [6,7]. There are several survey papers and research articles in the field dealing with the possibilities and limitations of this notion [27,31,30,34,44,45,48].

A relation $R = \{(u_1, \dots, u_n) : u_i \in \Sigma^*\}$ is *automatic* iff $\{conv(u_1, \dots, u_n) : (u_1, \dots, u_n) \in R\}$ is recognised by a finite automaton. A function f with m inputs and n outputs is said to be automatic iff the $(m+n)$-ary relation obtained from its graph, that is, $\{(u_1, \dots, u_m, v_1, \dots, v_n) : f(u_1, \dots, u_m) = (v_1, \dots, v_n)\}$, is automatic.

A class of languages $\{L_e : e \in I\}$ defined using indexing I is said to be an *automatic family* or an *automatic class* if I is a regular set of words and $\{(e, x) : x \in L_e, e \in I\}$ is automatic.

3.2 Transducers

A *transducer* processes its inputs at different speeds: it does not necessarily process one symbol from each input component, but possibly none or many. Formally, a transducer is a tuple $(Q, n, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, n is an input arity, Σ is a finite set of symbols, $q_0 \in Q$ is the starting state, $F \subseteq Q$ is the set of final states and δ is a transition relation, that is, a subset of $Q \times (\Sigma^*)^n \times Q$. Transducers are not deterministic, hence δ is not necessarily functional (with $Q \times (\Sigma^*)^n$ as potential domain and Q as potential range). A run of a transducer is a sequence of the form $(p_0, s_{1,1}, \dots, s_{1,n}, p_1), (p_1, s_{2,1}, \dots, s_{2,n}, p_2), \dots (p_{k-1}, s_{k,1}, \dots, s_{k,n}, p_k)$ where p_0 is the starting state and for all $i < k$, $(p_i, s_{i+1,1}, s_{i+1,2}, \dots, s_{i+1,n}, p_{i+1}) \in \delta$. Note that the lengths of the $s_{i,j}$ may differ. The run is *accepting* if $p_k \in F$, and the accepted input is $(w_1, w_2, \dots, w_n)$, where $w_i = s_{1,i} s_{2,i} \dots s_{k,i}$ for $i = 1, 2, \dots, n$. The relation recognised by a transducer is the set of inputs it accepts (in some accepting run). Such relations are called *rational* or *transduced*. The main difference between an automatic relation and a transduced relation is that the transducer can read the inputs independently at different speeds, thanks to which the non-determinism of the transducer may prove useful. Nivat [39] provided a characterisation of a transduced relation based on the notion of a homomorphism over a finite set A, defined as a mapping h from A to $\Sigma^\star$ extended to the mapping $h^\star$ from $A^\star$ to $\Sigma^\star$ such that for all $\sigma \in A^\star$, $h^\star(\sigma)$ is obtained from σ by replacing each occurrence of a symbol a in σ by $h(a)$. Then, an n-ary relation R over $\Sigma^\star$ is transduced iff there exists a finite set A, a regular set L over A, and n homomorphisms $h_1, \dots, h_n$ over A such that the relation R is the set $\{(h_1^\star(u), \dots, h_n^\star(u)) : u \in L\}$.

A function f is said to be *transduced* or *rational* if the relation $\{(u_1, \dots, u_m, v_1, \dots, v_n) :$

$f(u_1, \ldots, u_m) = (v_1, \ldots, v_n)\}$ is rational.

A class of languages $\{L_e : e \in I\}$ indexed by I is said to be a *transduced family* or *transduced class* if I is a regular set of words and the relation $\{(e, x) : e \in I, x \in L_e\}$ is recognised by some transducer. Transduced families have some of the decidability properties of automatic families, in particular those below.

Proposition 1. *If $\{L_e : e \in I\}$ is a transduced class and a transducer $\mathbf{M}$ accepts $\{(e, x) : e \in I, x \in L_e\}$, then one can effectively (from parameter y or e or finite set D), for each of the following sets, find a DFA recognising it:*

(a) $A_y = \{p : y \in L_p\}$;
(b) $A'_y = \{p : y \notin L_p\}$;
(c) $B_y = \{p : \{z \in L_p : |z| > |y|\} \neq \emptyset\}$;
(d) $B'_y = \{p : \{z \in L_p : |z| > |y|\} = \emptyset\}$;
(e) $C_D = \{p : D \subseteq L_p\}$;
(f) $C'_D = \{p : D = L_p\}$;
(g) $F_e = \{x : x \in L_e\}$.

In particular, given d and e, it can be effectively determined whether $L_d \subseteq L_e$ and whether $L_d \subset L_e$.

Proof. (a) Given the word y, one can easily construct a non-deterministic finite state automaton (NFA) which on input e, simulates $\mathbf{M}$ on input (e, y) and accepts e iff the transducer accepts the pair (e, y). This non-deterministic automaton can then be converted to a DFA using standard techniques.

(b) This follows from (a) by using the standard techniques for constructing a DFA for the complement of the language accepted by a given DFA.

(c) One can construct an NFA which accepts an index e iff the transducer $\mathbf{M}$ accepts a pair (e, x) with $|x| > |y|$. This NFA just guesses the symbols of x and uses a counter to make sure that it accepts e only if at least $|y|+1$ symbols have been read for x. From this, a DFA accepting B_y can easily be constructed.

(d) This follows from (c) by using the standard techniques for constructing DFAs for complements.

(e) This follows from (a) by using the techniques for constructing a DFA for the intersection of the languages accepted by given DFAs.

(f) This follows from (a), (b), (d) by using the techniques for constructing a DFA for the intersection of the languages accepted by given DFAs.

(g) Given e, one can construct an NFA which on input x, simulates $\mathbf{M}$ and accepts x iff $\mathbf{M}$ accepts (e, x). This NFA can then be converted to a DFA using standard techniques.

The corollary on the decidability of inclusion and proper inclusion of members of the class follows from (g) and the decidability of the subset relation for languages accepted by DFAs. $\square$

3.3 Learning Theory

Inductive inference has been defined in a statistical but noneffective way by Solomonoff [46,47] and by Gold [16] in a more algorithmic way; the present work follows Gold's approach. Gold's model is to learn recursively enumerable sets from texts which are example sequences containing all members of a set but no nonmember of the set.

More formally, Gold [16] defined a *text* as a mapping T from $\mathbb{N}$ to $\Sigma^* \cup \{\#\}$, whose contents, denoted $content(T)$, is the set $\{T(n) : n \in \mathbb{N} \wedge T(n) \neq \#\}$. It is a text for a language L iff $content(T) = L$. The initial segment of text T of length n is denoted $T[n]$. In this context, # indicates that no element from the underlying set is provided to the learner; it is unrelated to the use of # in the definition of the convolution of two words.

To learn a target class $\mathcal{L} = \{L_e : e \in I\}$, defined using indexing I, a learner uses a hypothesis space $\mathcal{H} = \{H_e : e \in J\}$, defined using indexing J, with $\mathcal{L} \subseteq \mathcal{H}$.

The following notions are adapted from Gold [16]. A learner uses some alphabet Γ for its memory. It starts with an initial memory and hypothesis. On each datum, it updates its memory and hypothesis. That is, a *learner* is a mapping $\mathbf{M}$ from $(\Gamma^* \cup \{?\}) \times (\Sigma^* \cup \{\#\})$ to $(\Gamma^* \cup \{?\}) \times (J \cup \{?\})$, together with an initial memory mem_0 and hypothesis hyp_0. Intuitively, ? denotes both null memory (different from ε) and null hypothesis (when the learner issues no hypothesis). One can extend the definition of a learner to arbitrary initial sequences of texts T, setting $\mathbf{M}(T[0]) = (mem_0, hyp_0)$, and then inductively setting $\mathbf{M}(T[n+1]) = (mem_{n+1}, hyp_{n+1}) = \mathbf{M}(mem_n, T(n))$. Intuitively, mem_n and hyp_n are the memory and conjecture of the learner after having seen the data in $T[n]$, respectively. A learner $\mathbf{M}$ converges on text T to a hypothesis e iff for all but finitely many n, $hyp_n = e$. Note that the memory is not required to converge. A learner $\mathbf{M}$ *explanatorily learns* a language L if for all texts T for L, $\mathbf{M}$ converges on T to a hypothesis e with $H_e = L$.

A learner $\mathbf{M}$ (*explanatorily*) learns a class $\mathcal{L}$ of languages iff it learns each $L \in \mathcal{L}$ [8,13,16]. In this paper, for the sake of brevity, "learning" stands for "explanatorily learning".

Blum and Blum [5] defined a finite sequence σ to be a *locking sequence* for a learner $\mathbf{M}$ on a language L if (a) $content(\sigma) \subseteq L$, (b) the hypothesis e of $\mathbf{M}$ on σ satisfies $H_e = L$ and (c) for all τ with $\sigma \preceq \tau$ and $content(\tau) \subseteq L$, the hypothesis of $\mathbf{M}$ on τ is e. They showed that if $\mathbf{M}$ learns L then such a σ exists.

A learner $\mathbf{M}$ is said to be recursive if the corresponding function F, mapping (old memory, datum) to (new memory, hypothesis), is recursive. Jain, Luo and Stephan [23] defined $\mathbf{M}$ to be an *automatic* learner if F is automatic. Finally, $\mathbf{M}$ is said to be a *transduced* learner if F can be computed by a transducer. See Figures 1 and 2 illustrating the statements of Example 2 and Proposition 3, respectively, for examples of automata associated with transduced learners, where transitions are labeled with triples that consume symbols from old memory, input and new memory, respectively (the fact that for both example and proposition, hypothesis and new memory are the same allows one to use triples rather than quadruples, as one would expect from the general definition of a transduced learner).

In this work, learners are recursive, and can or not be transduced or automatic. The memory limitations of the learner discussed in this paper is based on the length of the memory of the learner in terms of the length of the longest datum seen so far. Thus, for example, a learner $\mathbf{M}$

is word size memory bounded if for some constant c and for all finite sequences σ, if $M(\sigma) = (mem, hyp)$ and $n = \max\{|x| : x \in content(\sigma)\}$, then $|mem| \leq n + c$. Similarly, the learner is $O(n^2)$ memory bounded if $|mem| \leq cn^2 + c$.

Example 2. *For all $e \in \{0,1\}^+$, let $L_e = \{0,1\}^* \setminus (\{0,1\}^* \cdot \{e\})$ and consider the class defined by the transduced family $\{L_e : e \in \{0,1\}^+\}$. This is a transduced family as given input (e, w), a transducer can (i) guess if the length of w is smaller than the length of e and verify it or (ii) guess that length of e is at most the length of w, guess the starting position of last $|e|$ characters of w and verify that the last $|e|$ characters of w do not equal e. The transduced learner for this family has its current memory always the same as the current hypothesis e (initialised to 0). For an input word x, if x ends in e then e is updated to its length-lexicographic successor else e remains unchanged. Thanks to its non-deterministic nature, a transducer can check whether x ends in e and give the corresponding output.*

During the learning process, as long as the current value of e is length-lexicographically strictly below the target, the learner will eventually see an input ending in e, as there are infinitely many such inputs, and then update the hypothesis and memory to the next binary word in length-lexicographical order. Eventually, e reaches the correct value and then no further datum can cause another update of the hypothesis. Hence, one can verify that the transduced learner indeed converges to the correct hypothesis. Proposition 5 provides a more complicated version of this class proved to have a transduced but not an automatic learner.

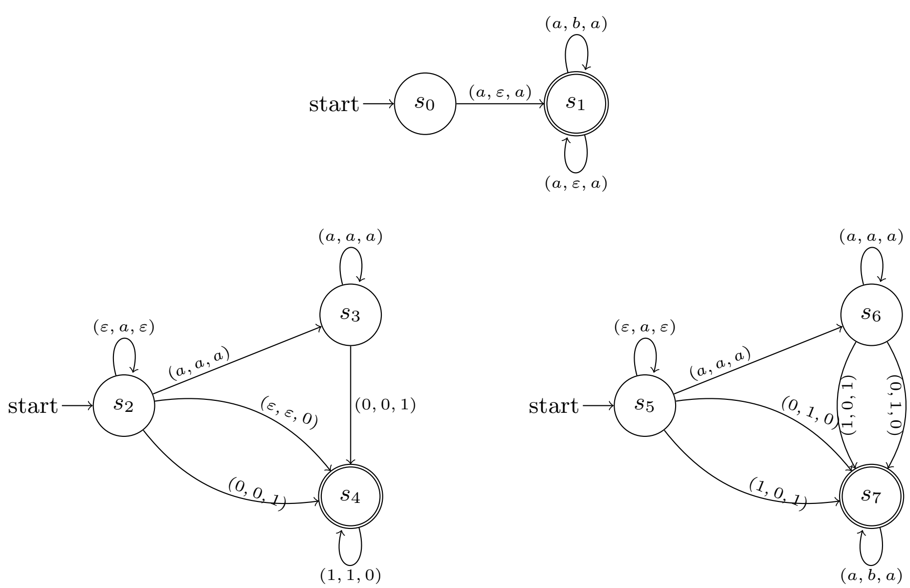

Figure 1. Learner for Example 2

Figure 1 depicts the automaton associated with a transduced learner that achieves the task described in Example 2. The triples that label the edges represent the mapping from old memory,

om (equal to old hypothesis), and datum, w, to new memory, nm, and new hypothesis, nh, the last two being identical and represented as the triple's last element. For ease of notation, one allows multiple start states; if the transducer starts in the wrong state then it gets stuck. Recall that only accepting runs processing all input and output symbols make a tuple (om, w, nm, nh) with $((om, w), (nm, nh))$ in the graph of the function computed by the transducer. The variables a and b stand for arbitrary bits. Triples involving a or b exist for any values that a and b can take. Let old memory (equal to old hypothesis) h and datum w be given.

- The first automaton is for the case where w is shorter than h. The hypothesis should then be kept. The transducer can consume h's first symbol, then possibly some of the following symbols in h and then possibly a symbol from h and a symbol from w, again and again, until the last symbols of h and w are consumed in the last step.
- The second automaton is for the case where w ends in h. Then either h is of the form 1^k with $k > 0$ and has to be changed to 0^{k+1} or h is of the form $\sigma 01^k$ with $k \geq 0$ and σ possibly empty and has to be changed to $\sigma 10^k$. First, all but the last $|h|$ symbols in w, if any, are consumed. The upper transition from s_2 to s_4 is for the case where h is of the form 1^k, producing the new hypothesis's first 0. The lower transition from s_2 to s_4 is for the case where h and what remains of w are of the form 01^k, consuming h's and w's first 0 and producing the new hypothesis's first 1. The transitions from s_2 to s_3 and from s_3 to s_4 are for the case where h and what remains of w are of the form $\sigma 01^k$ for nonempty σ, consuming all symbols in σ and the 0 that follows (in both h and w) and producing the new hypothesis' 1 after σ. Once in state s_4, the 1s that end h and w (in equal numbers), if any, are consumed and give rise to as many 0s in the new hypothesis.
- The third automaton is for the case where w is at least as long as h and does not end in h. The hypothesis should then be kept. First, all but the last $|h|$ symbols in w, if any, are consumed. Then h and what remains of w are of the form $\sigma 0/1\tau$ and $\sigma 1/0\tau'$, respectively, with σ possibly empty and with τ and τ' possibly empty and of the same length. The transition from s_5 to s_6 is for the case where σ is not empty. The transitions from s_5 to s_7 and from s_6 to s_7 consumes the 0 and 1 one of which occurs in h, the other in w, after σ. The loop at s_7 consumes all symbols in τ and τ', if any.

3.4 Additional Constraints on Learning

For a given learner $\mathbf{M}$ and text T, let $hyp_0, hyp_1, \ldots$ and $mem_0, mem_1, \ldots$ be the corresponding sequences of hypothesis and memory, respectively, as defined above. Often, one is interested in learners having additional properties.

A learner $\mathbf{M}$ is *consistent* on T iff every hypothesis hyp_n except ? contains all data observed [4], that is, $content(T[n]) \subseteq H_{hyp_n}$. Consistent learners might indeed not issue a hypothesis, hence hyp_n can be equal to ?; this is useful when they need more data or time to come up with a good hypothesis. An alternative lets $\mathbf{M}$ output Σ^* rather than ?, provided the former is available in the hypothesis space.

A learner is *conservative* on T iff every revision of a hypothesis occurs only when the previous hypothesis is inconsistent, that is, if $hyp_{n+1} \neq hyp_n$ then either $hyp_n = ?$, or $content(T[n+1]) \not\subseteq H_{hyp_n}$ [1,42]. A mind change from a usual hypothesis e to the special symbol ? is "conservative" only when e is inconsistent while a mind change from ? to a normal hypothesis e is always allowed in conservative learning.

Recall that a learner should not converge to ? but to a normal hypothesis e.

Conservativeness and consistency try to hinder Pitt's delaying tricks in learning [43]. However, Case and Kötzing [11] showed that in a polynomial time setting, delaying can be combined with consistency and conservativeness on general assumptions. This work provides similar evidence for learning of automatic families by transduced learners.

Jantke [20] defined a learner to be *strong-monotonic* on T iff for all $m < n$, $H_{hyp_m} \subseteq H_{hyp_n}$ or at least one of hyp_m and hyp_n is ?. Wiehagen [51] considered the weaker notion of *monotonic learning* where monotonicity is with respect to the target set only: for all $m < n$, $H_{hyp_m} \cap content(T) \subseteq H_{hyp_n} \cap content(T)$ or at least one of hyp_m and hyp_n is ?.

Wiehagen [50] called a learner *iterative* if the learner's memory is identical to the current hypothesis, that is, $mem_n = hyp_n$ for all n. Iterativeness is quite sensitive to the hypothesis space being chosen. Indeed, some hypothesis spaces have multiple hypotheses and one can code up some memory items by switching between equivalent hypotheses. This method is called "padding" and looks a bit like "cheating", though it is quite common. Iterative learning which uses one-one hypothesis spaces is much more restrictive. Indeed, when learners are given only small computational power, in which case they often need several rounds to revise a hypothesis, their only strategy is to use an intermediate memory, stored in the hypothesis, in order to save the intermediate values of such computations.

Here, all requirements above are with respect to the texts for the languages in the class to be learnt: a learner is consistent, conservative, etc., iff it is is consistent, conservative, etc., on each text for each language in the target class $\mathcal{L}$, respectively. This is thus the class version of consistency, conservativeness, etc. For global consistency, conservativeness and related criteria, one would impose the constraint on all texts for any language, whether it belongs to the target class or not.

4 Background on Automatic and Transduced Functions

It is well known that a finite automaton recognises a regular language in linear time. One can generalise the notion of automaticity from sets to relations and functions, see Section 3.1, where the reader is referred in particular to the usual introductory papers [7,17,28,31,44] of this notion. Automatic relations have the advantage that relations that are first-order definable from other automatic relations are themselves automatic [7,28]. This simplifies many proofs involving this concept. Furthermore, one can often use the pumping lemma to show that some relation is not automatic. Recall that functions are automatic iff their graph, that is, the set of valid tuples of inputs and corresponding outputs, is automatic. It is not immediately clear that automatic functions are linear time deterministic since recognising a graph and *computing the output of a string from the input* are two different tasks. However, Case, Jain, Seah and Stephan [10] showed that automatic functions coincide with those computed by linear time one-tape Turing machines that let both input and output start at the left end of the tape and that map the convolution of all inputs to the convolution of all outputs, so that only one tape is needed. In other words, a function is automatic iff it is linear-time computable with respect to the most restrictive variant of this notion. Increasing the number of tapes or not restricting the position of the output on the tape results in a larger complexity class.

Transduced (rational) functions and relations have been studied for much longer than automatic functions. If one fixes in a transduced relation all but one input by constants, the resulting set is regular. Classical approaches to transduced functions are the automata of Mealy [37] and Moore [38] to formalise transduction. Today, they are fundamental notions taught in automata theory lectures [3,19,49]. The following table gives a comparison of properties between automatic and tranduced relations, functions and families.

Concept	Automatic Case	Transduced (Rational) Case
Automaton	Reads all inputs at same speed	Reads inputs at different speeds
Type of automaton	NFA or DFA (equivalent)	NFA (non-determinism essential)
Closed under FO-Defn	Yes	No
Decidability of FO-Theory	Yes	No
Membership in Families	Two-Sided Decidable	One-sided Decidable
Closure of Families	Boolean Closure	Only under Union
Members in Families	Regular	Regular

Here, a First-Order Definition (FO-Defn) defines a new automatic function or relation using a formula whose quantifiers range over members of the structures (but not over functions, relations or sets) and which uses as parameters only other automatic functions and relations. The decidability of the first-order theory has been observed by Hodgson [17,18] as well as all researchers who redefined this notion [7,28]; it is known as the Khoussainov and Nerode theorem. The undedidability of the first-order logic of transduced relations can be obtained by coding Post's correspondance problem where two transduced functions compute from a string of indices the two sides of the correspondance and then test for equality. Another example is the theory of strings with concatenation and equality, where the concatenation can be computed by a transducer. While the existential theory is decidable by the algorithm of Makanin [36], it is undecidable for the next level of quantified formulas due to the undecidability of the inclusion problem for pattern languages [26]. For example, the inclusion of the pattern $00x1x0x1x$ in the language of the pattern $0yy$ is equivalent to the first-order formula

$$\forall x\, \exists y\, [00 \cdot x \cdot 1 \cdot x \cdot 0 \cdot x \cdot 1 = 0 \cdot y \cdot y].$$

Given an automatic family, an automaton can explicitly check whether a word is in the family or outside. Therefore, the family $\{L_e : e \in E\}$ is automatic iff $\{\Sigma^* \setminus L_e : e \in E\}$ is automatic. However, in a transduced family, a nondeterministic automaton can recognise whether a word is in the language, while there might be no transducer which recognises whether a word is outside the language. For example, with the transduced family of all $L_e = \{x \cdot e \cdot y : x, y \in \{0,1\}^*\}$ such that $e \in \{0,1\}^*$, the relation $\{(e,z) : z \in L_e\}$ is recognised by a transducer while $\{(e,z) : z \notin L_e\}$ is not. This is the meaning of transduced families being one-sided decidable in the above table.

The same phenomenon is observed with respect to the closure of families under operations to form new families from old ones. Indeed, given automatic families $\{L_e : e \in E\}$, one can make new families with indices of the form $\{H_{conv(i,j)} : i, j \in E\}$ and $H_{conv(i,j)}$ is a Boolean combination of L_i and L_j. In the case of transduced families, one can only do this when $H_{conv(i,j)} = L_i \cup L_j$. Setting $H_{conv(i,j)}$ to either the complement of L_i, to $L_i \setminus L_j$ or to $L_i \cap L_j$, one can see that in all three cases, the family of all sets $L_e = \{0,1\}^* \cdot e \cdot \{0,1\}^*$ with $E = \{0,1\}^*$ provides a counterexample: the resulting collection of sets of the form $H_{conv(i,j)}$ is not a transduced family. However, transduced families are closed under concatenation in the following sense. If 2 is a symbol outside the alphabet of $I \cup J$ and $\{L_i : i \in I\}$ and $\{H_j : j \in J\}$ are transduced families, then so is the family of all $K_{i2j} = L_i \cdot H_j$ with $i \in I$ and $j \in J$.

The following properties of transducers are important and will be used at various places in the present work.

If $f(x_1, x_2, \ldots, x_k)$ is a transduced function with one output and if $g(y_1, y_2, \ldots, y_r)$ is another transduced function, then so is

$$h(y_1, \ldots, y_{i-1}, x_1, x_2, \ldots, x_k, y_{i+1}, \ldots, y_r) = g(y_1, \ldots, y_{i-1}, f(x_1, x_2, \ldots, x_k), y_{i+1}, \ldots, y_r).$$

Similarly, one can use a transduced function to postprocess one of the outputs of g and obtain one or more new outputs. With this type of concatenation, it is important to make sure that none of the inputs or intermediate values is used at two places or that two outputs of one function are fed into another function. Otherwise, one could interchange the order of 0^m and 1^n in 0^m1^n by separating them out in the output of the first transducer, passing them as two independent inputs to the next transducer, which can then put these parts together in the opposite order. It is well-known from automata theory that such an operation cannot be carried out by a finite transducer.

However, one can move around a constant amount of information and the transducer can non-deterministically guess a constant amount of information from the future part of its input and later verify that it guessed correctly. When an incorrect guess is made, the transducer can avoid all accepting states and abort the computation. Also, as two inputs can be read alternately, one can merge one string at certain places into another string so as to achieve more involved storage formats for the learner. An example will be given as the array memory in Description 18.

5 Automatic versus Transduced Learners

This section provides the basics of the differences between automatic and transduced learners. Iterated automatic functions are Turing complete, thus they can, by updating the memory, compute everything in sufficiently many iterations. However, they fail to memorise the data items they observe and in case there are no repetitions of crucial data, learnability fails. This is overcome by transduced learners as they can append the data observed to their long term memory and then process the memory step by step in slow speed; the process is similar to that of certain linear time learners discussed by Case, Jain, Seah and Stephan [10]. However, there will be an excessive memory usage and this as well as subsequent sections will look at the amount of memory used.

The present section will mainly focus on separating the two learning notions by classes where the transduced learner can maintain a memory not exceeding the longest datum observed by more than a constant (word-sized memory). It is shown that these separations can also be obtained by learners satisfying additional qualities. In particular, Proposition 8 in the next section shows that some automatic classes can be learnt by transduced learners but not by automatic ones. Proposition 5 shows that there exists a transduced class which can be learnt by a transduced learner but cannot be learnt by an automatic learner.

The following proposition states some advantages of tranduced learners with respect to memory used/property satisfied as compared to automatic learners.

Proposition 3. *Let I be the index set $\{0,1\}^* \cdot \{1\} \cup \{\varepsilon\}$. For $w \in \{0,1\}^* \cdot \{1\}$, let $L_w = \{0^i 1^j : i \in \mathbb{N}, j < |w|, w(j) = 1\}$. Let $L_\varepsilon = \emptyset$. Let $\mathcal{L} = \{L_w : w \in I\}$. Now the following statements hold:*

(a) *$\mathcal{L}$ is a transduced family;*
(b) *$\mathcal{L}$ is not an automatic family;*
(c) *$\mathcal{L}$ can be conservatively learnt by an iterative automatic learner using word-sized memory;*
(d) *$\mathcal{L}$ cannot be learnt consistently by an automatic learner;*
(e) *$\mathcal{L}$ can be learnt conservatively and consistently by an iterative transduced learner using word-sized memory.*

Proof. Properties (a) – (e) are shown in turn.

(a) The transducer gets as input the two words w (the index) and x. In case $w = \varepsilon$, the transducer rejects the input. Otherwise, for $|w| = n+1$, if x is of the form $0^i 1^j$, $w(n) = 1$ and $w(j) = 1$, then the transducer accepts, else it rejects. Note that the transducer can easily do this by ignoring the part 0^i from x, then reading w and 1^j at the same speed and checking whether $w(j) = 1$ and finally checking that the last symbol in w is 1.
(b) Suppose by way of contradiction that some $\mathcal{L}' = \{L_e : e \in I\} \supseteq \mathcal{L}$ is an automatic family. For j larger than the pumping constant for the automaton accepting the automatic family, there is an e such that $L_e = \{0\}^* \cdot 1^j$. For $i \geq |e|$, it then follows from the pumping lemma that $0^i \cdot 1^{j'}$ is also in L_e for some $j' > j$. Thus, $L_e \neq \{0\}^* \cdot 1^j$, a contradiction.
(c) An automatic learner for $\mathcal{L}$ maintains a memory $w \in \{0,1\}^*$ (which is also its hypothesis) starting with $w = \varepsilon$. The learner ignores all data that start with 0 (that is, memory is not updated on such data). On input 1^j, the memory $w = w(0)w(1)\ldots w(n)$ is updated as follows. In case $|w| \geq j+1$, then the new memory is the same as w except that $w(j)$ is updated to 1. In case $|w| < j+1$, then the new memory is $w0^{j-|w|}1$. Thus, if the memory w before the current input is 011 then w on datum ε is updated to 111, whereas on datum 11111 it is updated to 011001.

Note that the above operation is clearly automatic as the $(j+1)$-th bit of w can be obtained using the input 1^j. As the learner's memory and hypothesis are the same, it is also iterative. The defined learner is easily seen to be conservative, as it only makes updates when the previous hypothesis does not contain the current datum.

(d) For a contradiction, suppose that there is an automatic learner for $\mathcal{L}$ which is consistent and uses hypothesis space $\{H_e : e \in J\}$. Consider the language $L_{1^k} = \{0^i 1^j : j < k\}$, where k is much larger than the pumping constant of the automatic learner. Let σ be a locking sequence for the learner on L_{1^k}. Suppose the hypothesis of the learner after seeing σ is e. Thus, $H_e = L_k$. Consider any $i > \max(|e|, |mem|)$, where mem is the memory of the automatic learner after seeing input σ. The learner, if provided with input string $0^i 1^k$, then has to change its hypothesis, as it would not be consistent otherwise. By pumping down the string $0^i 1^k$, there is also an update of the hypothesis by the learner on input string $0^i 1^j$ for some $j < k$, in contradiction to the fact that σ is a locking sequence for the learner on L_{1^k}.

(e) The transduced learner keeps memory $w \in \{0,1\}^*$ (which is also its hypothesis, thus the learner is iterative), starting with $w = \varepsilon$. It revises its memory w on input $0^i 1^j$, by updating $w(j)$ to 1, where the memory may need to be extended as in part (c). Note that the transducer reads both inputs independently and therefore can skip 0^i before updating w based on 1^j. The learner is easily seen to be consistent. It is also conservative as it revises its hypothesis only when its previous conjecture does not contain the new datum.

The picture of the transducer learner is given in Figure 2; X stands for any digit. The first parameter is the old memory, the second parameter is the input string, and the third parameter is the new memory. Suppose the second parameter, the input string, is $0^i 1^j$, the old memory is w and the new memory is w'. In state q_0, the transducer first consumes the 0s in the input string, and then goes to state q_1. In state q_1, the transducer matches the bits of w and w' one by one while consuming the 1s in the input string. In case $j < |w|$, the transducer learner just makes sure that the bit $w(j)$ of the output memory is 1 (at the end of the input string going from state q_1 to state q_3) and then checks whether the rest of w and the rest of w' are the same before finishing/accepting in state q_4. In case $|w| = j$, the learner just needs to check that the new memory is formed by appending 1 to the old memory (see transition from state q_1 to state q_4 on input $(\varepsilon, \varepsilon, 1)$). In case $j > |w|$, the transducer learner checks whether the new memory is formed by adding $0^{j-|w|}1$ to the old memory. This is done by going to state q_4 via state q_2.

This completes the proof. □

Remark 4. An anonymous referee pointed out that there is a variant of this result where one uses an automatic family to separate conservative automatic learning with word-size memory from consistent automatic learning. Furthermore, by Proposition 22 below, the class is consistently and conservatively learnable by a transduced learner. That variant holds for the class given by Jain, Luo and Stephan [23] which consists of the languages $L_\varepsilon = \{0,1\}^*$, $L_2 = \{0,1,2\}^*$ and $L_y = \{2^{|y|}\} \cup \{x \in \{0,1\}^+ : y \not\preceq x\}$ for $y \in \{0,1\}^+$. This class has a conservative word-sized automatic learner and does not have a consistent automatic learner at all.

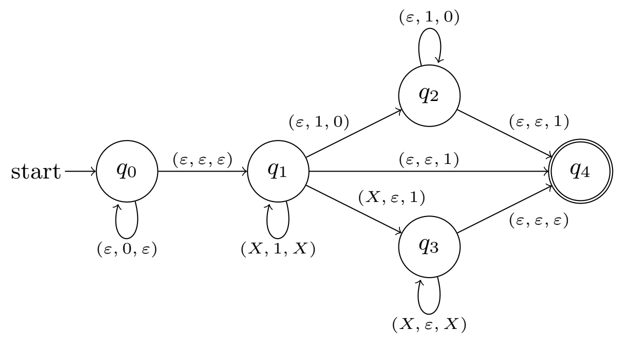

Figure 2. Learner for Proposition 3

Proposition 5. *There is a transduced family which can be learnt by a transducer with word-size memory while it does not have an automatic learner at all.*

Proof. Let $\Sigma = \{0, 1\}$ and

$$I = \big\{ endconv(1^k, 0^n, w10^m) : k \geq \max(n, |w| + 1 + m) \text{ and } n > m \text{ and } w \in \Sigma^* \big\}.$$

For each member d of I of the form $endconv(1^k, 0^n, w10^m)$, let $L_d = \Sigma^* \cdot 0^n{}_{\leq k} \cup \{w10^m\}$. It will be shown that $\mathcal{L} = \{L_d : d \in I\}$ satisfies the statement of the proposition. The transducer learner can use $\mathcal{L}$ itself as the hypothesis space. Except when having seen (possibly many times) at most one string in the input, the memory of the learner would be of the form $endconv(1^k, 0^n, v)$, where $|v|$ and n are at most k and the number of trailing 0s in v is at most n.

Here is the intended interpretation of the memory being $endconv(1^k, 0^n, v)$.

- The longest string seen so far is of length k.
- v is a string with the least number of trailing 0s amongst those seen so far, and the one seen earliest in case there are many such strings.
- n in 0^n is the least number of trailing 0s in the strings seen so far, v excepted.

The hypothesis of the learner would be for L_d in case $d \in I$, and an arbitrary default one otherwise. The transducer learner can easily memorise the first string, convert the memory into the required form when the second string comes, and proceed as described in the next paragraph from the third distinct string onwards.

On a new datum x and an old memory $endconv(1^k, 0^n, v)$, a transducer can easily update its memory to maintain the above properties. Indeed, it can nondeterministically guess which of $|x|$ and k is greater, if any, consume the extra symbols from the greatest of these until the remaining parts are of same length, guess where the trailing 0s start in x and in v, verify whether x needs to replace v, and check whether the number of trailing zeros in x or in v needs to replace the n in 0^n (two numbers which can be easily compared by the transducer). Correspondingly, the updated k, 0^n and v can be verified by the transducer.

However, $\mathcal{L}$ cannot be learnt by any automatic learner using any hypothesis space. Suppose for a contradiction that such an automatic learner $\mathbf{M}$ exists. Suppose the learner uses

Γ as alphabet set for its memory. Let n be such that $2^n > |\Gamma| + 1$. Let G be a function such that $G(w_0, w_1, \ldots, w_n)$ is $\mathbf{M}$'s memory after seeing inputs $w_0, w_1, \ldots, w_n$. By the automaticity of $\mathbf{M}$ applied $n+1$ times, G is automatic and $|G(w_0, w_1, \ldots, w_n)|$ is strictly smaller than $\max(|w_0|, |w_1|, \ldots, |w_n|) + c(n+1)$ for some constant c depending on $\mathbf{M}$. Let p be the number of states in the finite automaton $\mathbf{G}$ that accepts the graph of G. Let m be a number such that $2^m > (|\Gamma| + 1)^{np^2+p^2+c(n+1)}$. Let $w_0, w_1, \ldots, w_n \in \Sigma^m$ be given and let mem be $G(w_0 0^{p^2}, w_1 1^{p^2} 0^{p^2}, \ldots, w_i 1^{ip^2} 0^{p^2}, \ldots, w_n 1^{np^2} 0^{p^2})$. Thus, $|mem|$ is strictly smaller than

$$\max(|w_0 0^{p^2}|, |w_1 1^{p^2} 0^{p^2}|, \ldots, |w_i 1^{ip^2} 0^{p^2}|, \ldots, |w_n 1^{np^2} 0^{p^2}|) + c(n+1),$$

equal to $m + np^2 + p^2 + c(n+1)$. So mem can take at most $(|\Gamma|+1)^{m+np^2+p^2+c(n+1)}$ many values. Observe that $(w_0, w_1, \ldots, w_n)$ can take $2^{(n+1)m}$ many possible values. However,

$$2^{(n+1)m} > 2^m (|\Gamma| + 1)^m > (|\Gamma| + 1)^{m+np^2+p^2+c(n+1)}.$$

By the pigeon-hole principle, there must exist $u_0, u_1, \ldots, u_n, v_0, v_1, \ldots, v_n \in \Sigma^m$ such that $(u_0, u_1, \ldots, u_n) \neq (v_0, v_1, \ldots, v_n)$ and

$$\begin{aligned} G(u_0 0^{p^2}, u_1 1^{p^2} 0^{p^2}, \ldots, u_i 1^{ip^2} 0^{p^2}, \ldots, u_n 1^{np^2} 0^{p^2}) = \\ G(v_0 0^{p^2}, v_1 1^{p^2} 0^{p^2}, \ldots, v_i 1^{ip^2} 0^{p^2}, \ldots, v_n 1^{np^2} 0^{p^2}) = mem'. \end{aligned}$$

Suppose $u_j \neq v_j$ for some j with $0 \leq j \leq n$. By the pigeonhole principle, for at least two distinct i, i' with $m + jp^2 \leq i < i' \leq m + (j+1)p^2$, the states of $\mathbf{G}$ after seeing the first i symbols of

$$conv(u_0 0^{p^2}, u_1 1^{p^2} 0^{p^2}, \ldots, u_n 1^{np^2} 0^{p^2}, mem')$$

and the first i symbols of

$$conv(v_0 0^{p^2}, v_1 1^{p^2} 0^{p^2}, \ldots, v_n 1^{np^2} 0^{p^2}, mem')$$

would be the same as the corresponding states of $\mathbf{G}$ after seeing the first i' symbols of

$$conv(u_0 0^{p^2}, u_1 1^{p^2} 0^{p^2}, \ldots, u_n 1^{np^2} 0^{p^2}, mem')$$

and after seeing the first i' symbols of

$$conv(v_0 0^{p^2}, v_1 1^{p^2} 0^{p^2}, \ldots, v_n 1^{np^2} 0^{p^2}, mem')$$

(at most p^2 pairs of states, but $p^2 + 1$ positions). So the words can be "pumped down" and $\mathbf{G}$ accepts

$$\begin{aligned} conv(u_0 0^{p^2}, u_1 1^{p^2} 0^{p^2}, \ldots, u_{j-1} 1^{(j-1)p^2} 0^{p^2}, u_j 1^{jp^2} 0^{p^2-d}, \\ u_{j+1} 1^{(j+1)p^2-d} 0^{p^2}, \ldots, u_n 1^{np^2-d} 0^{p^2}, mem'') \end{aligned}$$

$$\text{and } conv(v_0 0^{p^2}, v_1 1^{p^2} 0^{p^2}, \ldots, v_{j-1} 1^{(j-1)p^2} 0^{p^2}, v_j 1^{jp^2} 0^{p^2-d}, \\ v_{j+1} 1^{(j+1)p^2-d} 0^{p^2}, \ldots, v_n 1^{np^2-d} 0^{p^2}, mem'')$$

where $d = i' - i$ and mem'' is the corresponding modification of mem' obtained after deleting the corresponding symbols. This implies that the memory of $\mathbf{M}$ after it reads the sequence of words σ defined as

$$u_0 0^{p^2},\ u_1 1^{p^2} 0^{p^2},\ \ldots,\ u_{j-1} 1^{(j-1)p^2} 0^{p^2},\ u_j 1^{jp^2} 0^{p^2-d},\ u_{j+1} 1^{(j+1)p^2-d} 0^{p^2},\ \ldots,\ u_n 1^{np^2-d} 0^{p^2}$$

is the same as the memory of $\mathbf{M}$ after it reads the sequence of words σ' defined as

$$v_0 0^{p^2},\ v_1 1^{p^2} 0^{p^2},\ \ldots,\ v_{j-1} 1^{(j-1)p^2} 0^{p^2},\ v_j 1^{jp^2} 0^{p^2-d},\ v_{j+1} 1^{(j+1)p^2-d} 0^{p^2},\ \ldots,\ v_n 1^{np^2-d} 0^{p^2}.$$

Assume that $\mathbf{M}$ further reads the same sequence of strings τ from the set $\Sigma^{m+np^2} \cdot 0^{p^2}$ with each string in the set appearing at least once in τ. Then $\sigma\tau$ is a text for $L_{endconv(1^{m+np^2} 0^{p^2}, u_j 1^{jp^2} 0^{p^2-d})}$, while $\sigma'\tau$ is a text for $L_{endconv(1^{m+np^2} 0^{p^2}, v_j 1^{jp^2} 0^{p^2-d})}$, which are distinct languages. Still, in either case, $\mathbf{M}$ outputs the same hypothesis after processing each word after the $(n+1)$-th word. Thus, $\mathbf{M}$ can learn at most one of the above languages, a contradiction. □

Remark 6. Fischer and Rosenberg [14] showed that one cannot decide whether a rational relation is of a simpler form, automatic in particular. Starting from this result, an anonymous referee asked whether there are similar undecidable results for transduced families. This can be answered by using the following version of Post's correspondence problem. Given a set of triples $(a, v_a, w_a) \in \Sigma \times \Sigma^* \times \Sigma^*$ over a sufficiently large alphabet Σ, where each $a \in \Sigma$ appears in at most one triple as the first component, a correspondence is a word $a_1 a_2 \ldots a_n \in \Sigma^+$ such that $v_{a_1} v_{a_2} \ldots v_{a_n} = w_{a_1} w_{a_2} \ldots w_{a_n}$. It is known that given a set of triples as above, existence of a correspondence is an undecidable problem. Without loss of generality, one can assume that v_a, w_a have both at least length 2; this will facilitate the proof. Furthermore, if u is a correspondence, so is uu. Thus, if there is a correspondance in an instance of the problem, then there are infinitely many of them. Consider the automatic family

$$L_{a_1 \ldots a_n} = \{x \in \Sigma^* : x \neq v_{a_1} v_{a_2} \ldots v_{a_n} \vee x \neq w_{a_1} w_{a_2} \ldots w_{a_n}\}.$$

Note that $L_{a_1 \ldots a_n} = \Sigma^*$ whenever $a_1 a_2 \ldots a_n$ is not a correspondence; without loss of generality, such words always exist. Furthermore, the family is transduced: a transducer can, on an input $(a_1 \ldots a_n, x)$, nondeterministically check whether $x \neq v_{a_1} v_{a_2} \ldots v_{a_n}$ or $x \neq w_{a_1} w_{a_2} \ldots w_{a_n}$ by simply following the index string $a_1 \ldots a_n$ and accepting iff the corresponding word does not agree with x. The only amount of nondeterminism needed is to recognise the end of the index string, the end of x and whether one has to compare with v or with w. Now there are two cases.

(a) If the instance of the correspondence problem does not admit any solution, then the transduced family consists only of the set Σ^* and every index is for this set. Hence, it is an automatic family, which is finitely and explanatory learnable.

(b) If the instance admits a solution $u = a_1 \ldots a_n$, set $x = v_{a_1} v_{a_2} \ldots v_{a_n}$ and let each set L_{u^m} be equal to $\Sigma^* - \{x^m\}$. By the assumption on the considered correspondence, $|x| \geq 2|u|$ and the family is not automatic: one cannot automatically retrieve x^m from u^m. Furthermore, the family contains the set Σ^* plus infinitely many cosingleton sets. Therefore, the family is not explanatorily learnable from text.

As one cannot determine whether the family satisfies condition (a) or (b), one cannot provide an algorithmic solution to whether a transduced family is equal to an automatic family, nor to whether a transduced family is explanatorily learnable from text.

6 Memory-Size Hierarchies of Transduced Learners for Automatic Classes

This section studies the memory complexity of learning automatic families. While it is not known whether automatic learners without memory restriction can learn more than automatic learners using word-sized memory, the results in this section show that for transduced learners, there is a real hierarchy of the space usage needed, even when learning automatic classes. Here space usage is measured as a function of the length of the largest datum seen so far. Proposition 7 provides a tool to show that for automatic families of certain type, one can provide a lower bound on the memory needed to learn them, independently on the type of learner. Proposition 8 exhibits an automatic class which can be learnt with quadratic memory by a transduced learner, but which can be learnt neither by an automatic learner nor by any recursive learner using subquadratic memory. Corollaries 9 and 10 establish the memory hierarchy: they show that for transduced learners, memory bounded by polynomials of degree $k+1$ lets one learn more classes than memory bounded by polynomials of degree k. Proposition 14 shows that every learnable automatic class can be learnt by some transduced learner with exponential memory.

The following result provides lower bounds for the long term memory in terms of the longest example seen so far.

Proposition 7. *Let S be a regular set of words. Let $\mathcal{L} = \bigcup_{n\in\mathbb{N}}\big(\{S_{\le n}\} \cup \big\{S_{\le n} \setminus \{z\} : z \in S\big\}\big)$. After it has seen a word of length n, any learner for $\mathcal{L}$, whether automatic, transduced or recursive, needs in the worst case memory of length at least $|S_{\le n}|/c$ for some constant c.*

Proof. Consider a learner $\mathbf{M}$ for $\mathcal{L}$ which uses alphabet Γ for its memory.

Let $n \in \mathbb{N}$ be given. Suppose there are two finite sequences σ and τ with $content(\sigma) \neq content(\tau)$, $content(\sigma) \cup content(\tau) \subseteq S_{\le n}$, and the memories of $\mathbf{M}$ after having seen the inputs σ and τ are the same. Let z be in the symmetric difference of $content(\sigma)$ and $content(\tau)$, say in $content(\sigma) \setminus content(\tau)$. Let T be a text for $S_{\le n} \setminus \{z\}$. Then either $\mathbf{M}$ converges to the same hypothesis on σT and on τT, or it fails to converge on both. As σT and τT are texts for $S_{\le n}$ and $S_{\le n} \setminus \{z\}$, respectively, which are different languages in $\mathcal{L}$, $\mathbf{M}$ fails to learn at least one of these languages.

It follows that $\mathbf{M}$ has at least $2^{|S_{\le n}|}$ different memory values on different finite sequences with elements from S of length at most n. Thus, at least one of these memory values must be of length at least $|S_{\le n}|/\log_2(|\Gamma|)$. □

If $|S_{\le n}| = n^k$ for some constant k, the lower bound is $\Omega(n^k)$. If $|S_{\le n}| = c^n$ for some constant c, the lower bound is $\Omega(c^n)$ on the maximum length of the memory on input sequences containing words of length up to n. It will be shown that for some regular languages S, similar upper bounds are obtained.

Proposition 8. *Let $S = \{0\}^* \cdot \{1\}^*$. So $|S_{\leq n}| = (n^2+3n+2)/2$. Let $\mathcal{L} = \bigcup_{n\in\mathbb{N}}\big(\{S_{\leq n}\} \cup \big\{S_{\leq n} \setminus \{z\} : z \in S\big\}\big)$. Then $\mathcal{L}$ can be learnt by a transduced learner with memory size $O(n^2)$, but $\mathcal{L}$ cannot be learnt by any automatic learner, even without explicit memory bounds.*

Proof. Suppose for a contradiction that an automatic learner $\mathbf{M}$ learns $\mathcal{L}$. Fixing $n \in \mathbb{N} \setminus \{0\}$, consider a sequence σ containing exactly n elements of length at most n from S. As $\mathbf{M}$ is automatic, its memory on σ can be of length at most cn for some constant c, see Jain, Luo and Stephan [23, Proposition 15], and thus the number of possible memories of $\mathbf{M}$ after seeing σ is bounded by d^n for some constant d. On the other hand, there are at least $\binom{n^2/2}{n}$ possible contents of such sequences, and for large enough n, this is larger than d^n. Thus, for large enough n, there exist two sequences σ and σ', with different contents, each containing exactly n elements of length at most n, such that the memory of $\mathbf{M}$ is the same after seeing σ and after seeing σ'. Let $x \in \mathit{content}(\sigma) \setminus \mathit{content}(\sigma')$ be given and let T be a text for $S_{\leq n} \setminus \{x\}$. Then $\mathbf{M}$, on texts σT and $\sigma' T$, either does not converge or converges to the same conjecture even though they are texts for different languages in $\mathcal{L}$. Thus, $\mathbf{M}$ cannot learn $\mathcal{L}$.

It is now shown that a transduced learner can learn $\mathcal{L}$. To represent languages in $\mathcal{L}$, one uses indices of the form $0^i1^j2^k$ and 3^{k+1}. If $w = 0^i1^j2^k$ then L_w contains all words in S of length up to $i+j+k$ except for 0^i1^j. If $w = 3^{k+1}$ then L_w contains all members of S of length up to k.

The transduced learner $\mathbf{M}'$ uses as memory a string in $\{0,1\}^*$ where certain positions are marked. Intuitively, for memory $w = w(0)w(1)\ldots w(n-1)$, each position in the memory string represents a string of the form 0^i1^j, the marked positions representing the strings that have been seen in the input. More precisely, position p in the string represents the concatenation of 0^m where m is the number of 0s in $w(0)w(1)\ldots w(p)$, with the longest final segment of $w(0)w(1)\ldots w(p)$ consisting of nothing but 1s. For example, if the value of the memory data structure is $011'0'1'$, then the strings represented by the positions of the marked (primed) letters as ordered in the memory word are 011, 00 and 001 (indicating that they are precisely the words that have been observed, in any order). Note that having to take all 0s before the third mark, 0111 is not represented in the memory. The overall goal of updating the memory is to let the current input word 0^i1^j be represented in the memory by a position and the symbol at this position be marked. The beginning of the memory can be marked in order to record that ε has been observed in the input.

$\mathbf{M}'$ starts with memory ε and no mark. Suppose that at any time, (1) the new input word is 0^i1^j, (2) i' is the number of 0s in the current memory word and (3) j' is the number of 1s in the current memory word which have exactly i 0s in the memory before their position. Note that j' is 0 if either $i' < i$ or $i' \geq i$ and after the first i 0s, either the memory word ends or another 0 follows. The non-deterministic transducer operates as follows.

1. First, while there is a 0 to be read in both the current memory and the input datum d, read from the current memory and copy each read symbol to the new memory. Whenever a 0 is read, read it also on d until at least one of current memory or d has only 1s left. Thus, the

current memory is copied until $\min(i, i')$ 0s (along with intermediate 1s in the memory) are copied from the current memory and d has the first $\min(i, i')$ symbols read.

2. If $i' \geq i$ (this is determined by the transducer by guessing, and verifying when reading the rest of the words) then read j 1s from the current datum and j' 1s from the current memory and write $\max(j, j')$ 1s to the new memory. Then copy the remaining part of the current memory to the new memory.
3. If $i' < i$ then first copy all remaining 1s from the current memory to the new memory and then copy the remaining symbols $0^{i-i'}1^j$ from the current datum to the new memory.
4. Besides this, all symbols copied from the current memory to the new memory keep their marks in case they have some already, and the symbol at the position representing 0^i1^j in the new memory also receives a mark.

What follows demonstrates how the hypothesis is written when writing to the new memory, as both outputs are independently written into different words. Still, the workings of the transducer is best understood when the transducer is thought of as non-deterministically extracting a hypothesis from the new memory.

Without loss of generality, assume that the input language contains an element of length at least 3, as otherwise the input language can be easily learnt. Recall that the memory keeps track of all words seen in the input using the marks. It can also be used to indicate missing words: if a position representing 0^i1^j is unmarked, then 0^i1^j has not been seen in the input. If there is no 1 in the memory just before the i-th 0, then $0^{i-1}1$ is not seen in the input. Call a word v stored in the memory *maximal* iff $v1$ is not represented in the memory. Suppose the longest input seen so far is of length n, all positions in the memory are marked and at most one datum is missing in the input data from $0^*1^*_{\leq n}$. So at least one maximal word in the memory is of length n. Assume there exists a maximal word v in the memory whose length is of parity different to the parity of n. Since the length of v is necessarily smaller than n, $v1$ is of length at most n, and therefore one datum is missing in the input data from $0^*1^*_{\leq n}$ indeed, $v1$ being that datum.

When writing the new hypothesis, the learner $\mathbf{M}'$ can verify and act according to the first of which of the following cases applies, where i' still denotes the total number of 0s in the memory.

1. If the position representing ε is not marked, then the hypothesis is $2^{i'}$.
2. If the memory ends in 1, then the hypothesis is $0^{i'+1}$.
3. If some 0 in the memory is not preceded by a 1, then the hypothesis is $0^{i-1}1^{i'-i+1}$ for the least i such that the i-th 0 is not preceded by a 1.
4. If some 0 in the memory is not marked then the hypothesis is $0^i2^{i'-i}$ for the least i such that the position representing 0^i is not marked.
5. If some 1 is not marked then the hypothesis is $0^i1^j2^{j'-j}$ where i and j are such that 0^i1^j is represented by the position of the leftmost unmarked 1 and j' is the number of 1s between the i-th and $(i+1)$-th 0s in memory.
6. If all positions are marked and the lengths of all maximal words represented, except for the maximal word v, share the same parity, then the hypothesis is $v1$.
7. If none of the above conditions applies then the hypothesis is $3^{i'+1}$ (denoting the set of all words in S of length up to i').

Note that in order to check the sixth case, the transducer can always count the number of 0s up to the current position modulo 2, and then count the number of 1s following this 0 modulo 2. Also, $\mathbf{M}'$ can easily verify for any particular case that none of the earlier cases applies. Thus, $\mathbf{M}'$ can generate the hypothesis based on the above.

The memory keeps track of all data seen in the above described data structure. Also, the hypothesis is computed in such a way that it is correct whenever all except perhaps one member of S, of length at most n, have been seen, but no data of length longer than n have been observed. Indeed, for n being the maximum length of an input datum, the following holds:

(i) if the only missing datum from $S_{\leq n}$ is ε then clause 1 will give the correct hypothesis;
(ii) if the only missing datum from $S_{\leq n}$ is 0^n then clause 2 will give the correct hypothesis;
(iii) if a 0 is not preceded by a 1 in the memory, then clause 3 will give the correct hypothesis;
(iv) if there is an unmarked position in the memory then clauses 4 or 5 will give the correct hypothesis.

The only remaining case for missing data is then handled by clause 6, and gives the correct hypothesis by the observation preceding the listing of the 7 cases. In case no datum from $S_{\leq n}$ is missing from the input, then clearly clause 7 outputs the correct hypothesis.

As an example to illustrate parts of the proof, the table that follows shows a possible sequence of data, memory and hypothesis (denoted by w) updates of $\mathbf{M}'$. The initial memory is ε and the old memory at any step is the new memory of the previous step.

Datum	New Memory	w	Words in L_w
1	$1'$	ε	None
01	$1'01'$	2	$0, 1$
ε	$'1'01'$	00	$\varepsilon, 0, 1, 01, 11$
00	$'1'01'0'$	02	$\varepsilon, 1, 00, 01, 11$
0	$'1'0'1'0'$	11	$\varepsilon, 0, 1, 00, 01$
11	$'1'1'0'1'0'$	333	$\varepsilon, 0, 1, 00, 01, 11$

This completes the proof. □

The previous proof can be generalised to larger alphabet sizes, provided one fixes the alphabet size and the exponent of the polynomial. Thus, both preceding results give the following corollary, in which the bound $\Theta(n^k)$ for the family given by $\mathcal{L}_k$ indicates that one can learn with memory size $O(n^k)$, but every learner needs at least memory size $\Omega(n^k)$.

Corollary 9. *Suppose Σ has k symbols $0, 1, \ldots, k-1$ and let S^k be $\{0\}^* \cdot \{1\}^* \cdot \ldots \cdot \{k-1\}^*$. So $|{S^k}_{\leq n}| = \binom{n+k}{k} = \sum_{m \leq n} \binom{m+k-1}{k-1}$. Let $\mathcal{L}_k$ be the class of all $L_{vw} = {S^k}_{\leq |vw|} \setminus \{v\}$ for all $w \in \{k\}^*$ and $v \in \{0\}^* \cdot \{1\}^* \cdot \ldots \cdot \{k-1\}^*$, and L_w be ${S^k}_{<|w|}$ for all $w \in \{k+1\}^+$. Then $\mathcal{L}_k$ can be learnt by a transduced learner with a memory length bound of $\Theta(n^k)$.*

An additional corollary to Proposition 8 can be obtained with respect to target-sized learners. The result uses the fact, proved by Jain, Ong, Pu and Stephan [24], that for all automatic families, there is a constant c such that for each language L_e, if words in the language L_e have at most length n then the shortest index of L_e has at most length $n + c$. Stephan [49] defined a learner to have a *target-sized* memory bound if the length of the memory is never longer than the length of the shortest index of the language being learnt plus a constant. If one relaxes this bound by just requiring the existence of a function f such that the memory is never longer than $f(n)$ with n being the size of the shortest index of the target, then one can get the following corollary.

Corollary 10. *The class $\mathcal{L}_k$ (from Corollary 9) of subsets of $\{0\}^* \cdot \{1\}^* \cdot \ldots \cdot \{k-1\}^*$ of all words except perhaps one of length up to n, $n \in \mathbb{N}$, can be learnt by a transduced learner with target-sized memory of size $O(\binom{n+k}{k})$, but not with any better memory constraint, except for a multiplicative constant.*

Proposition 11. (*a*) *The class of languages of all binary words except perhaps one of length up to n, $n \in \mathbb{N}$, can be learnt by a transduced learner with exponential target-sized memory.*
(*b*) *The class $\mathcal{L}$ consisting of all $L_w = \{v \in \Sigma^* : \varepsilon <_{ll} v \leq_{ll} w\}$ with $w \neq \varepsilon$ and $L_\varepsilon = \Sigma^*$ cannot be learnt with any type of target-sized memory.*

Proof. Part (a) follows from Proposition 14 below. For part (b), suppose by way of contradiction that learner $\mathbf{M}$ learns the class $\mathcal{L}$ using some target-size memory bound $f(\cdot)$. Suppose σ is a locking sequence for $\mathbf{M}$ on L_ε. There are only finitely many possibilities for the memory of $\mathbf{M}$ on $\sigma\tau$ for any τ such that $content(\tau) \subseteq L_\varepsilon$. Let $\max_{ll}(S)$ denote the length-lexicographically maximum element in S. Thus, there are τ, τ' such that $content(\tau) \subseteq L_\varepsilon, content(\tau') \subseteq L_\varepsilon$, $\max_{ll}(content(\sigma\tau))$ and $\max_{ll}(content(\sigma\tau'))$ are different, but the memory of $\mathbf{M}$ after seeing $\sigma\tau$ and the memory of $\mathbf{M}$ after seeing $\sigma\tau'$ are the same. So $\mathbf{M}$ does not identify at least one of the languages $L_{\max_{ll}(content(\sigma\tau))}$ and $L_{\max_{ll}(content(\sigma\tau'))}$ from the texts $\sigma\tau\varepsilon^\infty$ and $\sigma\tau'\varepsilon^\infty$, respectively. □

Lange, Zeugmann and Zilles [35] provide an overview of the studies that investigate what role hypothesis spaces play in learning. Here, a hypothesis space is said to be class-comprising if $\{H_e : e \in J\} \supseteq \{L_e : e \in I\}$ and class-preserving if $\{H_e : e \in J\} = \{L_e : e \in I\}$. Note that the classes $\mathcal{L}_k$ from Corollary 9 have a consistent transduced learner which uses a class-comprising transduced hypothesis space, namely, the space of all finite subsets of S_k as defined in the preceding corollary. The hypotheses would just be the memory contents. A word w is in the hypothesis given by a memory content v iff the word w is represented by a marked position in the memory. Therefore, one has the following corollary.

Corollary 12. *The class of all finite subsets of S_k as defined in Corollary 9 can be learnt by a consistent, conservative and iterative learner which uses the current memory content as a hypothesis. This learner is also learning the subclass $\mathcal{L}_k$ of this class using a transduced class-comprising hypothesis space.*

Sometimes, for consistency, one needs to remember all observed data in order to know whether a certain element, namely v in the example below, has been observed or not. However, an inconsistent automatic learner only needs to track the one or two longest words seen so far and can therefore work with much less memory. For this class, even a consistent learner with class-comprising hypothesis space needs a quadratic-sized memory, as it needs to memorise all observed data from $\{0\}^* \cdot \{1\}^*$ of length up to $|v| + k$; see Proposition 7.

Corollary 13. *Let $\mathcal{L}$ consist of the following sets, for all $v \in \{0\}^* \cdot \{1\}^*$:*

- $L_{v2^{k+1}} = \{v2^{k+1}\} \cup (\{0\}^* \cdot \{1\}^* \cdot \{2\}^* \cdot \{3\}^*)_{\le |v|+k} \setminus \{v\}$;
- $L_{v3^{k+1}} = \{v2^{k+1}, v3^{k+1}\} \cup (\{0\}^* \cdot \{1\}^* \cdot \{2\}^* \cdot \{3\}^*)_{\le |v|+k}$.

$\mathcal{L}$ has an automatic learner. However, there is no consistent automatic learner for $\mathcal{L}$ and every consistent transduced learner for $\mathcal{L}$ needs at least a quadratic-sized memory.

Proposition 14. *If an automatic class can be explanatorily learnt from text then there is a transduced learner for the same class which learns it with $O(c^n)$ sized memory for some constant c that depends only on the class.*

Proof Sketch. The idea is to simulate an automatic learner $\mathbf{M}'$ which learns the class from fat texts (that is, texts in which each appearing item appears infinitely often). Jain, Luo and Stephan [23] showed that such automatic learners exist for all learnable automatic classes. The transduced learner $\mathbf{M}$ will work in phases. In each phase, it will archive the new data received during this phase and update a data structure (called binary list below) to record the data archived in the previous phase (and then delete it from the archive). Additionally, the learner $\mathbf{M}$ will simulate $\mathbf{M}'$ on the previously recorded data in the binary list. The memory of $\mathbf{M}$ is of the form

$$conv(d, mem, other\ info)@b_0b_1 \dots b_r@PL@CL,$$

where one of the b_is is marked, and some items in PL may be marked (PL is for "previous list" and CL for "current list"). The description of the various items in the memory of $\mathbf{M}$ is as follows.

1. A convolution of a data-counter $d \in \Sigma^*$, current memory mem of $\mathbf{M}'$ in the simulation and some other information as needed by $\mathbf{M}$ as described below.
   From now on, suppose the data counter value d is lexicographically the $(n+1)$-th element of Σ^*.
2. A binary list $b_0b_1 \dots b_r$, for some $r \ge n$. Additionally, there is a mark on b_n to denote the current value of the data-counter.
   The intention is that b_i is 1 iff the $(i+1)$-th string in length-lexicographic order has been observed in the input. Initially, the list contains just $b_0 = 0$ with a mark on b_0 (thus, $d = \varepsilon$ and $n = 0$).
   This binary list may however lag behind the actual data archivation. All data items observed in phases before the current and previous phases would have the corresponding bit appropriately set to 1. The data items archived in the previous phase would be updated during the current phase and then deleted from the archive.
3. PL: The list of data-items archived in the previous phase which may not have been recorded in the binary list.
4. CL: The list of data-items being archived in the current phase.

In each phase, the learner **M** does several rounds of the following steps until the next phase starts.

1. Keep simulating $\mathbf{M}'$ as follows. If the current bit b_n corresponding to the data-counter d is 1, then feed d to $\mathbf{M}'$, else feed a pause symbol $\#$. Correspondingly, update the memory of $\mathbf{M}'$ and output the hypothesis of $\mathbf{M}'$.
2. If there are copies of d in the list PL, then remove all these copies and set $b_n = 1$, else leave b_n and PL unchanged (this update might take several rounds, see details below).
3. This step is done only if step 2 is completed in this round.
   If b_n is the last bit in the binary list and PL is empty
   Then
   - set $d = \varepsilon$,
   - change n to 0, that is set the marker on the binary list to b_0 and
   - change @@ surrounding the empty PL to @ and insert @ at the end of CL. That is, PL takes the value of old CL and CL becomes empty. This also means that the next phase starts after the archival of the new data item in step 4 below.

   Else change n to $n+1$ (by shifting the marker in the binary list and updating d to its length lexicographic successor), and if b_{n+1} does not exist then insert $b_{n+1} = 0$ in the binary list.
4. Archive the current datum at the end of CL.

The above is a general outline. Steps 1 and 4 are done in every round, step 2 spans several rounds and whenever step 2 is completed then step 3 starts.

For step 2, the following is done in general. If $d = \varepsilon$ then delete all occurrences of ε in the list PL. Also, if ε occurred in PL then b_0 is set to 1, else b_0 retains its old value. Otherwise, if $d \neq \varepsilon$, the following is done over several rounds. At the first round of cleaning, the first symbol d_0 of d is marked and every word in the memory PL which starts with d_0 has its first symbol (d_0) marked. In the next round, the following is done.

- In case all symbols of $d = d_0 d_1 \ldots d_m$ are marked, then the transducer unmarks them all, deletes all elements in PL which end after $d'_0 d'_1 \ldots d'_m$ (thus these elements are equal to d), and unmarks all elements starting with $d'_0 d'_1 \ldots d'_m$ which have further symbols after d'_m (thus these elements are not equal to d but are proper extensions of d). Furthermore, if a copy of d is found in the archived list PL or if $b_n = 1$ then b_n is set to 1, else b_n is set to 0.
- Otherwise, if $d'_0 d'_1 \ldots d'_m$ is marked and d_{m+1} is not marked, (i) each subword of the form $d'_m d_{m+1}$ is transformed to $d'_m d'_{m+1}$ in both the memory for d and the elements archived in PL and (ii) each element in PL starting with $d'_0 \ldots d'_m$ which does not continue with d_{m+1} gets the markers removed. The transducer knows non-deterministically which of the two cases applies and verifies the case when it sees d'_m and the unmarked symbol after it.

It is now easy to verify that **M** simulates the automatic learner $\mathbf{M}'$ on a fat text (as it repeats the elements seen from the beginning at the end of every phase). Thus, **M** learns each language learnt by $\mathbf{M}'$ on fat texts.

To determine the complexity of memory of **M**, assume that **M** has seen up to some point only words of length at most n. Then, in the current phase, d has taken at most $O(c^n)$ different values where c is the size of the alphabet. Furthermore, each processing of a value of d takes $O(n)$ rounds, as one has to send d symbols, one by one, through the memory. Therefore, $O(n \cdot c^n)$ words are archived in the current phase and the overall length of CL and PL is $O(n^2 \cdot c^n)$. If one makes c larger, one can absorb the polynomial factor before the c^n. □

7 A Space Bound for Learning all Learnable Transduced Classes

In inductive inference, there are cases where the amount of memory to learn finite members of a recursively enumerable class cannot be bounded by a recursive function of the largest element of the set to be learnt. In contrast, it will be shown that for transduced classes, some recursive function bounds memory usage. More precisely, one can use set-driven recursive learners and translate them into transduced learners whose memory bound is a function stemming from the short-term memomry needed to iteratively simulate the computation of the values of the set-driven learner plus an expression exponential in the size of the largest datum seen so far, needed for memory archivation and management.

Here a learner $\mathbf{M}$ is *set-driven* [41] if for all sequences σ and τ such that $content(\sigma)$ is equal to $content(\tau)$, $\mathbf{M}$'s memory and hypothesis are the same after seeing either σ or τ. It is first shown that every learnable transduced family can be learnt by a set-driven recursive learner, which can be obtained uniformly from a transducer learner for the family. This learner is defined for all transduced families. However, for unlearnable families, the learner will fail on some input texts. The learner employs the properties of transduced families listed in Proposition 1.

A *tell-tale* set for a language L with respect to a class $\mathcal{L}$ of languages is a finite subset D of L such that for all $L' \in \mathcal{L}$, if $D \subseteq L' \subseteq L$ then $L' = L$. Angluin [1] has shown that for any learnable family of languages $\mathcal{L}$, every language in $\mathcal{L}$ has a tell-tale with respect to $\mathcal{L}$.

Proposition 15. *Let a learnable transduced family $\mathcal{L} = \{L_e : e \in I\}$ be given. Then some set-driven recursive learner learns $\mathcal{L}$. Furthermore, such a learner can be effectively obtained from the transducer describing the transduced family.*

Hence the total space needed by the recursive learner (*that is, the long term memory as well as the short term memory needed for computation*) *is bounded by a recursive function in the length of the longest datum seen so far.*

Proof. The learner $\mathbf{M}$ is given by the following algorithm. Let $D = \{a_1, a_2, \ldots, a_m\}$ be the set of distinct words observed in the input so far. Note that $\mathbf{M}$ can keep track of D in its long term memory. Let $f(D)$ be the length-lexicographically least index e with $D \subseteq L_e$ (by Proposition 1, e can be found effectively). Let n be the maximum of

- the length of the description of the DFA accepting I,
- the length of the description of the transducer accepting $\{(e, x) : e \in I, x \in L_e\}$,
- the alphabet size for the transduced family and
- the lengths $|a_1|$, $|a_2|$, …, $|a_m|$ of all words in D.

The learner $\mathbf{M}$ then chooses the first of the following options which applies:

1. If there is $e \in I$ with $L_e = D$ then **M** outputs the length-lexicographically least such e;
2. If $f(D)$ is defined, because there is e with $D \subseteq L_e$, then **M** selects the length-lexicographically least index e of length at most $|f(D)|+n$ such that $D \subseteq L_e$ and there is no index d of length at most $|f(D)|+n$ with $D \subseteq L_d \subset L_e$;
3. Otherwise, D is not consistent with any hypothesis in the family and **M** outputs ? to signal that there is no valid conjecture.

By definition, **M** is set-driven. Furthermore, if the input language L_e is finite then **M** will converge to its length-lexicographically least index after having seen all elements. If L_e is infinite then there must exist a finite subset D of L (a tell-tale set) such that there is no language L_d in $\mathcal{L}$ such that $D \subseteq L_d \subset L_e$. Thus, for large enough n, after large enough data have been received, step 2 would let **M** output the least index for L_e. So **M** learns all languages in $\mathcal{L}$.

Let $n \in \mathbb{N}$ be given. Note that there are only finitely many data sets D with size of the maximum element bounded by n. Let $g(n)$ be the maximum of the long term and short term space used by the algorithm when run with the given parameterisation describing the transduced family and the data set D as input. The number $g(n)$ can be algorithmically computed from n and is an upper bound on the total space needed by the recursive learner for n denoting the length of the longest datum seen so far. □

Proposition 16. *If a class has a set-driven recursive learner using space bound $g(n)$, then it also has a transduced learner using space bound $g(n) + c^n$ for its memory, for some constant c when n denotes the length of the longest datum seen so far.*

Proof. Assume without loss of generality that c is at least equal to 2, and also that $g(n)$ is at least d^n, where d is greater than the alphabet size as this can be absorbed using the c^n bound, where n is the length of the longest datum seen so far.

From a recursive set-driven Turing Machine learner $\mathbf{Q}'$, one can construct another Turing Machine **Q**, which reads the inputs from a read only one way input tape containing the whole text, using a special symbol to separate the words. This Turing Machine **Q** has only one work tape and always has the current hypothesis at the left end of the tape (with an appropriate separator between the hypothesis and the rest of the work tape). **Q** may take several steps to update the hypothesis (and thus it may have garbage in the hypothesis portion of its work tape during some steps). However, if the hypothesis is not being changed after some new datum has been read, then **Q** does not visit the hypothesis portion of the work tape. Note that the above modification of $\mathbf{Q}'$ to **Q** can be done without increasing the space needed beyond $g(n)$ as long as it is at least d^n (note that constant multiplicative factors in the memory size can be absorbed by increasing the alphabet size of **Q**).

One can construct a transduced learner **M** for the class using a memory bound $g(n) + c^n$ as follows. The transduced learner **M** has in its memory two parts, next to each other except for a separator, as follows.

- The instantaneous description of learner **Q** except for the input tape, i.e., the content / head location of the work tape of the learner **Q** along with its state. Note that this takes space $g(n)$.

- After a separator, a list of data words with three pointers in them where symbols of data words may be marked. The words are separated using some separator. Here the first pointer is the reading position of the Turing machine learner mentioned above, the second pointer is the position of the memory clean-up and the third pointer is a position up to which the memory clean-up runs before restarting from the beginning of the data memory. This cleanup operation is described below.

The transduced learner **M** using non-determinism does the following updates to its memory in each round on reading a datum (by copying the updated version of the old memory into the new memory).

1. Simulate one step of **Q** (using the instantaneous description part of the memory).
2. Copy the beginning of the tape of **Q** (the hypothesis) into the hypothesis output.
3. In the above simulation of one step of **Q**, guess the input symbol read and verify it using the symbol below the first pointer. If needed, the first pointer is then moved right.
4. Do one step of the following algorithm which is distributed over several rounds to modify the word list part of the memory.
   4.1. Move the second pointer one step towards the right.
   4.2. In case that pointer moves from a separator between words onto a symbol $b(0)$ of a word b, mark all symbols $b(0)$ in the list of data words which are the first symbol of a word with a prime.
   4.3. In case that pointer moves from a symbol $b(m)$ in a word b to symbol $b(m+1)$, mark all symbols $b(m+1)$ which are unmarked and are to the right of a marked $b(m)$ with a prime and unmark all marked words $b(0)' \dots b(m)'$ which are not followed by $b(m+1)$.
   4.4. In case that pointer moves from a symbol $b(m)$ onto a word separator, all copies of $b(0) \dots b(m)$ but the first one and possibly the one with the first pointer on it are removed and all marking is undone. The algorithm for this runs as follows. Unmark that occurrence of a fully marked string $b(0) \dots b(m)$ between two separators in the list which either has the first pointer on it or is the first of all such strings and remove all further strings $b(0) \dots b(m)$ which are between two separators while unmarking all further marked strings which do not have a separator following them and which are therefore of the form $b(0)' \dots b(m)'b(m+1) \dots$ as they represent some longer data items than $b(0) \dots b(m)$.
5. Append the current datum to the end of the list of data words.
6. If the second pointer meets the third pointer, move the second pointer to the beginning and the third pointer to the end of the data sequence.

To verify that **M** learns the class, suppose the longest data item seen so far is of length n. Note that the memory part due to instantaneous description of **Q** is at most $g(n)$ symbols long, as **Q** has only processed data items up to length n. Furthermore, between two pointer updates in step 6, **M** goes over each word of length up to n at most twice, as after processing a word once all duplicates except for the one with the first pointer on are removed. Thus, the overall number of words processed by **M** can be bounded by $4 \cdot c^n$ where c is the size of the alphabet. Thus, there are at most $4 \cdot (n+1) \cdot c^n$ rounds between two executions of step 6. Therefore, the overall amount of data items archived by $\mathbf{M}$ is at most $(8n+12) \cdot c^n$, since the amount of data-items before the third pointer (just after it has moved for the last time) is at most $(4n+8) \cdot c^n$ and the amount of data-items after the third pointer (just before it moves the next time) is at most $(4n+4) \cdot c^n$. Taking into account that each word has at most n symbols plus a separation symbol, the overall amount of space used in the list can be bounded by $(8n^2 + 20n + 12) \cdot c^n + g(n)$ symbols. This memory size is bounded by $(c')^n + g(n)$ for some constant c'.

As all data-items will eventually be read by $\mathbf{Q}$ in the simulation (only copies of items which are already processed are omitted from memory), $\mathbf{Q}$ receives a text of the language to be learnt and will converge to a correct hypothesis. Thus, $\mathbf{M}$ will also converge to a correct hypothesis. This completes the verification of the algorithm. □

Corollary 17. *If a transduced class can be explanatorily learnt from text then there is a transduced learner for the same class which learns the class with $g(n)$ memory for some total recursive function g which depends only on the class.*

8 Consistency, Conservativeness and Iterativeness

Relations between consistency, conservativity and iterativeness in learning have been investigated for learning classes of recursively enumerable sets, with recent contributions from Case and Kötzing [12] as well as Jain, Kötzing, Ma and Stephan [21]. The results obtained there are a bit different from those for automatic classes where strong-monotonically learnable automatic classes have an iterative and conservative recursive learner. However, for transduced learners, strong monotonically learnable automatic classes have learners which are either iterative or conservative, but not necessarily both; see Proposition 23 below. These results are preceded by a description of general archiving techniques and results about the learnability of some widely studied classes in inductive inference.

The class of co-singleton subsets of the set of all binary words (that is, those sets which miss out exactly one word) does neither have an automatic learner nor an iterative one. However, it has a consistent and conservative transduced learner. An important tool is given by the *array memory*, introduced in the following description.

Description 18. The array memory is stored as one string, however, it is more easily visualised when it is written as an array, the string being the concatenation of the non-empty symbols of each row. Before going into the details of the array memory, here are two examples. They both store words that are all length-lexicographically above 011. The first one stores the words 0001, 0011, 001111, 1100, 1001001 and 111. The second one stores the words 111 and 0111.

```
(#        .1,#,#)
(#    .1,#;0,#,#)
(#    ;1,#;0,#,#)
(#.1,1;1,0;1,#,#)       (#  .1,#)
(1;0,1;1,0;0,1,1)       (1.1;1,1)
(1;0,0;0,1;0,1,1)       (1;1;1,1)
(0;0,0;0,1;1,1,0)       (0;1;0,0)
```

The words in the storage are stored bottom-up and are surrounded on both sides by the boundary word, 011. The symbol `#` stands for a placeholder (not for an empty space that would have been omitted), to be part of the string. If needed, it is used to extend a word and make it as long as the longest of the preceding words; it is also used to extend both occurrences of the boundary word and make them as long as the longest word in the array. Dots are used to precede the last symbol of any word that is longer than all preceding words, with semicolons preceding all the symbols below, whereas the symbols in all other words are preceded with commas. Finally, an initial column of `(`s and a final column of `)`s are added to the array.

The rows are concatenated from bottom to top, with blanks omitted, to give the string stored in memory. Thus, for the second example, the resulting string stored in memory is

```
(0;1;0,0)(1;1;1,1)(1.1;1,1)(#.1,#)
```

This order is chosen consistently with the reading and writing directions of the transducer, from bottom to top in the array memory. To check membership of a word in the array, one needs several rounds. More precisely, the word moves from left to right through the array and compares in each round with the words in two columns, as shown in the following picture. The word passed through the first example is 1100; it is prefixed by a colon. The word travels at double speed in order to keep up with the new items added into memory that come at single speed.

```
(#:#        .1,#,#) (#    :#    .1,#,#) (#        :#.1,#,#)
(#:#    .1,#;0,#,#) (#    :#.1,#;0,#,#) (#    .1,#:#;0,#,#)
(#:#    ;1,#;0,#,#) (#    :#;1,#;0,#,#) (#    ;1,#:#;0,#,#)
(#:0.1,1;1,0;1,#,#) (#.1,1:0;1,0;1,#,#) (#.1,1;1,0:0;1,#,#)
(1:0;0,1;1,0;0,1,1) (1;0,1:0;1,0;0,1,1) (1;0,1;1,0:0;0,1,1)
(1:1;0,0;0,1;0,1,1) (1;0,0:1;0,1;0,1,1) (1;0,0;0,1:1;0,1,1)
(0:1;0,0;0,1;1,1,0) (0;0,0:1;0,1;1,1,0) (0;0,0;0,1:1;1,1,0)
```

Some bit in memory (perhaps after the array) is maintained to see whether a copy of the word has been seen in memory. This would be the case in the round going from second to third diagramme above, where the word 1100 has moved over the words 001111 and 1100 and noted that the second of these words is a copy of itself. Note that when moving over a word of a new length, it is needed that the length be known as new, which is possible thanks to the semicolons up to the dot. While the diagramme above has clear rows, the storage format has not, as the spaces are omitted and therefore the word moves over all symbols from the bottom which are prefixed by a semicolon until it reaches the one which is prefixed by a dot. New word are added at the end, and extended if they are longer than the previous words. The following picture refines the previous one by adding 0111 and 00000000 at the end of the memory while looking for 1100.

```
                                            (#        :#      .0,#)
  (#:#        .1,#,#) (#    :#    .1,#,#,#) (#        :#.1,#,#;0,#)
  (#:#    .1,#;0,#,#) (#    :#.1,#;0,#,#,#) (#    .1,#:#;0,#,#;0,#)
  (#:#    ;1,#;0,#,#) (#    :#;1,#;0,#,#,#) (#    ;1,#:#;0,#,#;0,#)
  (#:0.1,1;1,0;1,#,#) (#.1,1:0;1,0;1,#,1,#) (#.1,1;1,0:0;1,#,1;0,#)
  (1:0;0,1;1,0;0,1,1) (1;0,1:0;1,0;0,1,1,1) (1;0,1;1,0:0;0,1,1;0,1)
  (1:1;0,0;0,1;0,1,1) (1;0,0:1;0,1;0,1,1,1) (1;0,0;0,1:1;0,1,1;0,1)
  (0:1;0,0;0,1;1,1,0) (0;0,0:1;0,1;1,1,0,0) (0;0,0;0,1:1;1,1,0;0,0)
```

One can also clean up. With the running example, this means checking whether the successor 100 of 011 occurs in the array and removing it from the array if it is found. If that is the case, the learner then has to record somewhere that it has been seen indeed. This is done with the concept of *list array memory.* It is a list of all elements in length-lexicographic order up to a certain string, using `+` and `-` to indicate which of those elements have been observed and which haven't, respectively, followed by an array whose elements are lower bounded by the last member of the list or its predecessor. For example, if the empty string, 00 and 11 are in the list (whereas 0, 1, 01 and 10 aren't) and the array contains 000 and 0001, two strings that are lower bounded by 11, the last element in the list, then the list array memory would be

```
+ 0- 1- 00+ 01- 10- 11+
(#  .1,#)
(#.0;0,#)
(1;0;0,1)
(1;0;0,1)
```

and the transducer would store it as the string

```
+0-1-00+01-10-11+(1;0;0,1)(1;0;0,1)(#.0;0,#)(#.1,#)
```

Updating a list array memory is possible as follows. To record a new word w upto the lower bound of the array memory, a transducer just has to check whether it is followed by `+` in the list and if not, replace `-` by `+`. If w is lexicographically above the lower bound of the array memory, the transducer would append w to the end and the learner would perform some regularisation work. More precisely, for as long as the array memory is non-empty, the learner would extend the last member of the list with its length-lexicographic successor, assuming it has not been observed and so letting `-` follow it. It would then check whether it is actually in the array memory and in case it is, delete all of its occurrences, update the lower bound of the array, and change `-` to `+` in the list.

Recall that an iterative learner [50] uses as memory the current hypothesis and nothing else. This is a strong constraint on the memory, first because it eventually has to stop growing, second because it cannot be updated from the moment the learner converges to the final hypothesis. This explains why the easily learnable class consisting of the language $\{0\}^+$ and all finite languages containing ε has an automatic learner, but no iterative learner.

Example 19. *The class of all finite sets has an iterative transduced learner using some transduced family as a hypothesis space.*

Proof. Note that an iterative transduced learner of the class of all finite sets could just use the list array memory as its hypothesis space. The members of the hypothesis are the words in the list part of the memory which are marked with +. The words in the array part are not yet integrated to the hypothesis because a transducer that decides membership cannot access them in one go. However, subsequent updates of the hypothesis in several rounds will end up in a situation where all members of the finite set to be learnt are in the list part and where the array part is empty. From then onwards, the memory is no longer revised. Therefore, the scheme just described gives rise to an iterative transduced learner for the class of all finite sets, together with the corresponding hypothesis space. □

For the following result, it seems indispensable to use the '?' conjectures, though that remains to be proven.

Example 20. *The class of finite sets can be learnt consistently and conservatively by a transduced learner using some transduced family as hypothesis space.*

Proof. The proof is very similar to that of the preceding result. The idea is to use a list array memory as memory for the learner with the same updating rules as before. However, the hypothesis space consists only of hypotheses formulated as explicit lists of members of the finite set. These are only updated when a new datum arrives. Therefore, one has to be able to check whether the datum is not already there before appending it to the current list of elements. So the learner does the following.

1. Maintain as a memory the same data structure as when iteratively learning finite sets.
2. In case the array memory part of the data structure is empty and all elements are in the explicit list memory, the hypothesis is the sublist of all members of the list memory which are marked with a +. Note that this list is in length-lexicographic ascending order without repetitions.
3. In case the array memory part of the data structure is nonempty, the hypothesis is ?.

Whenever the learner outputs a hypothesis e, the array memory part of the data structure is empty and all observed data are in the list memory part. Therefore, the hypothesis e contains all observed data and is consistent. Furthermore, an update from e to another hypothesis e' only occurs when some - is changed to + in the list memory part of the data structure, indicating that a new datum has been observed. The array memory part of the data structure becomes nonempty only when a new datum has been observed, and conservative learners can afford to issue

? when they make a mind change. The mind change from ? to a hypothesis e is always allowed for a conservative learner and occurs after sufficient clean-up operations on the memory. Thus, the learner that has been described is consistent and conservative, and will eventually converge to an explicit list of the finite set to be learnt, with elements given in length-lexicographic order. Thanks to the accompanying memory, all updates are properly done. □

Example 21. *Let L be an infinite regular set and $L_v = L \backslash \{v\}$ for $v \in L$. The class $\{L_v : v \in L\}$ of all co-singleton subsets of L has a consistent and conservative transduced learner M who does not use the special hypothesis ? and can always produce a real conjecture.*

Proof. Without loss of generality, one assumes $\{0\}^* \subseteq L$. This is not a big assumption anyway, as the pumping lemma implies that $u \cdot v^* \cdot w \subseteq L$ for some words u, v, w with v being non-empty. Fixing v to 0 and u, w to ε simplifies the proof. The running examples will assume that $L = \{0,1\}^*$, although this assumption is not needed for the proof.

Here also, the idea is to use a list array memory. The only modification is: (i) either one replaces `-` by `@` to indicate that a word, say v, is the index of the current conjecture L_v or (ii) one appends `@` after the k-th row from the bottom to indicate that 0^k is the current conjecture for L_{0^k}. Here are example memories for both cases, with $L = \{0,1\}^*$.

```
+ 0@ 1- 00+ 01- 10- 11+          + 0- 1- 00+ 01- 10- 11+
(#  .1,#)                        (#  .1,#)@
(#.0;0,#)                        (#.0;0,#)
(1;0;0,1)                        (1;0;0,1)
(1;0;0,1)                        (1;0;0,1)
```

In both cases, the words ε, 00, 11, 000 and 0001 have been observed, but the two hypotheses are different: they are L_0 for the first example and L_{0000} for the second one. The initial memory of the learner is `@(#.#)` which says that the conjecture is L_ε, that the bound of the array memory is ε and that this memory contains no word. The invariant is: (i) the list array memory records all data observed so far and (ii) exactly one `@` exists which marks the current hypothesis so that the transducer can compare the hypothesis with the current datum and at the same time, record the current datum in memory. For that reason, the `@` must sit at the position where the current datum is recorded in case it is equal to the index of the hypothesis.

The learner updates the memory in rounds and also carries out in each round one step of the memory management. More precisely, the learner does in each round the following.

1. Let x denote the input word.
2. If x has an entry in the list memory then proceed in accordance with the following cases.
   - 2.1. x is marked with `+`: no change.
   - 2.2. x is marked with `-`: change the mark to `+`.
   - 2.3. x is marked with `@` and there is another entry with `-`: change the mark of x to `+` and for the first y carrying the mark `-`, change this mark to `@`.

2.4. x is marked with `@` and no other entry is marked with `-`: introduce a new row in the array memory and append `@` behind it. Furthermore, the new row is of the form `(#.#)@` and the sign in the penultimate column of all other rows is a semicolon.

3. If x does not have an entry in the list memory, then it is verified that it is length-lexicographically above the border and it is appended to the end of the array memory.
4. In case `@` sits in row k from below and $x = 0^k$, that of both cases below which applies is performed.

4.1. There is an entry with mark `-` in the list memory. Then for the first y in the list memory which carries mark `-`, change the mark to `@` and remove `@` from the old position.

4.2. All marks in the list memory are `+` and the sign in the penultimate column of all other rows is a semicolon. Then introduce a new row in the array memory of the form `(#.#)` and append `@` to it. Furthermore, 0^k is added to the array memory.

5. Finally, one performs one step of the current clean-up round for the list array memory with the following additional constraint. When the lower bound happens to advance from a previous value to 0^k and the k-th row from below is marked with `@`, then this marker `@` is added after the new entry 0^k in the list memory and it is removed from the array memory.

Similarly to the previous examples, the following illustration deals with the updates done after v has been read while the current conjecture is L_v. For the first diagramme, ε, 00, 11, 000 and 0001 have been previously observed, L_0 is the current hypothesis and 0 is next seen in the input. For the second diagramme, ε, 0, 1, 00, 01, 10, 11, 000 and 0001 have been previously observed, L_{0000} is the current hypothesis and 0000 is next seen in the input. The situations after observing 0 and 0000 are shown below, respectively.

```
After observing 0                After observing 0000
+ 0+ 1@ 00+ 01- 10- 11+          + 0+ 1+ 00+ 01+ 10+ 11+
                                 (#      .#)@
(#  .1,#)                        (#  .1,0;#)
(#.0;0,#)                        (#.0;0,0;#)
(1;0;0,1)                        (1;0;0,0;1)
(1;0;0,1)                        (1;0;0,0;1)
New hypothesis is 1              New hypothesis is 00000
```

Again, all spacing is only for the reader's convenience and not in the actual memory. With the first example, the actual memory would be the string

```
+0+1@00+01-10-11+(1;0;0,1)(1;0;0,1)(#.0;0,#)(#.1,#)
```

and there could be other additional symbols from an on-going clean-up process, not reproduced in this example in order to keep it simple.

Eventually, after observing enough data and performing enough bookkeeping, the index w of the language L_w to be learnt will have an entry in the list memory. After some time, it will be the first entry in the list memory which is marked with either `-` or `@`. In case it is marked with

- and the current hypothesis is L_v, then v will be observed eventually and the hypothesis will be updated to the first entry in the list memory marked with - (that is, w).

Furthermore, the learner is consistent, as a hypothesis w for L_w is immediately updated when w occurs in the input and the new conjecture is a v for an L_v with v not yet observed, as either the entry v- is in the list memory or a new row is created in the array memory to make sure that v is longer than all data observed. □

Proposition 22. *Every learnable automatic class has a consistent and conservative transduced learner which uses the given automatic hypothesis space but also employs ? when a current conjecture cannot be computed due to lack of data or computation time.*

Proof. Let $\{L_e : e \in D\}$ be an automatic and learnable family. Without loss of generality, it can be assumed that D is infinite and $\varepsilon \in D$. Let $Succ_D$ denote the length-lexicographic successor inside D.

As the automatic class is learnable, following arguments of Angluin [1], each set L_e has a tell-tale set. In the case of an automatic learnable family, there is a constant c such that a tell-tale set for L_e can be $H_e = \{x \in L_e : |x| \leq |e| + c\}$ [23]. The tell-tale set thus satisfies the following two properties:

- For all d, e, if $H_e \subseteq L_d \subseteq L_e$ then $L_d = L_e$;
- Each set H_e is finite and can be enumerated by an algorithm from the index e.

Note that the second condition follows from the way H_e is defined; H_e can even be decided by an automaton given e, as automatic families have better effectiveness properties than uniformly recursive ones. Angluin [1] constructed a learner which makes use of the tell-tale sets. For automatic families, this learner has one bound d and then searches for all indices e up to the bound d whether they satisfy the following two properties:

- All data in H_e have been observed;
- No datum outside L_e has been observed.

When no e up to d qualifies, the bound d will be increased and all e will be checked again. Wherever it performs more than a constant amount of steps in the following algorithm, it issues a conjecture.

The transduced learner **M** runs the algorithm below in rounds and archives incoming data at the end of an array memory (an operation that is not explicitly mentioned in the description of the algorithm), in parallel, one incoming datum per round. In every round, it resumes execution from some line of the algorithm and performs at most a constant number steps before it conjectures a hypothesis. The upper bound in the number of steps allows **M** to reach an instruction that lets it output a hypothesis (thereby ending the round), unless it executes instruction 3 that captures a situation where a search is performed in chunks that end when the maximum authorised number of steps have been performed, with ? then being hypothesised.

1. Initialise variables d, e as ε and goto Step 2.
2. For all $y \in L_e$ with $|y| \leq |e| + c$ do Begin
3. Check whether y has occurred in the observed data and keep conjecturing ? during the search.
4. If it is found that y has not been observed then break out of the loop and goto Step 9 End.
5. For all observed data x (including new incoming data) do Begin
6. Check whether $x \in L_e$.
7. If all data archived are in L_e then conjecture e else conjecture ?.
8. If an $x \notin L_e$ is found then break out of the Loop and goto Step 9 End.
9. If $d = e$ then update $e = \varepsilon$ and $d = Succ_D(d)$, else update $e = Succ_D(e)$. Goto Step 2.

M uses the array memory to both store the observed data and perform checks as mentioned; the memory has no list part because membership checks for L_e are better performed using the array memory.

The array memory has, besides the usual entries for the observed data, additional entries that represent d, e and y, where y is a variable used to check membership. The lower bound of the array memory described when the data structure was introduced is not needed here bcause the kind of memory that is used here has no list part. Recall that for an automatic family, a single automaton reads, at the same speed, index e and datum x in order to decide whether $x \in L_e$. For that reason, one will keep d and e and y distributed over the rows in the same way as done with the data. For the checks, one moves copies of e and of y over the data and advances in each round by two columns in order to eventually reach the end of the array memory for completing the checks.

The learner **M** is consistent and conservative, as (i) it conjectures a hypothesis e different to ? only when it has been verified that all data observed are in L_e and (ii) it abandons e only for ? when some datum outside L_e has been observed. As **M** always verifies that all data in H_e have been observed prior to conjecturing e of L_e, the hypothesis L_e can only be false if the correct hypothesis contains elements outside L_e. Therefore, the false hypothesis will indeed be eventually abandoned. In case **M** would only go through false hypotheses which all get infinitely often abandoned, it would then check for the correct e infinitely often whether all data from H_e are observed and all observed data are in L_e. This check will turn out to be true eventually and therefore **M** would, in contradiction to the assumption, converge to e. Thus, **M** is indeed learning the class.

The end of the proof illustrates memory management. The illustration is simplified; it would have to be modified to make the learner transduced. However, the authors believe that it is sufficiently complex to give an insight into the mechanisms of the memory management of the learning algorithm. The example class consists of $\{2\}^+$ and all finite subsets of $\{2\}^*$ which contain ε. An index for a finite language L is a sequence of bits $e_0 e_1 \ldots e_k$ such that $e_0 = e_k = 1$ and for all $h \leq k$, $2^h \in L_e$ iff $e_h = 1$. for instance, 10101 is the index for $\{\varepsilon, 22, 2222\}$. Furthermore, ε is the index for $\{2\}^+$. One will keep as data for the algorithm d, e, y at the beginning of the array memory marked with the ! symbol. Furthermore, a copy of e or y might be sent through the array memory (advancing by two columns per cycle) in order to check whether all members of the memory array are in L_e and whether all elements of H_e are in the memory array, respectively.

If both checks receive the answer "yes" then **M** will conjecture e until it becomes inconsistent, that is, a new datum that is not in L_e is received (and recorded). The entry e will then be changed to the next possible value as described in the algorithm above. The figure that follows depicts a situation where the data ε, 222, 222, 22 and 22222 have been observed, the current hypothesis e is 101 and the bound d is 111. The array memory while checking whether $y = 22$ is in the memory would after two steps look like the following plus the update after processing two columns and reading 2222.

```
(!#!#!#    :#    .2)     (!#!#!#        :#.2,#)
(!#!#!#    :#    ;2)     (!#!#!#        :#;2,2)
(!1!1!#  .2:#,2,#;2)     (!1!1!#  .2,2,#:#;2,2)
(!1!0!2  ;2:2,2,2;2)     (!1!0!2  ;2,2,2:2;2,2)
(!1!1!2.#;2:2,2,2;2)     (!1!1!2.#;2,2,2:2;2,2)
```

M would have noted that the datum 22 has been crossed by the searching copy of y, concluding that y is in the array memory. While this was an advancement within the first loop, the following is a round inside the second loop. Assume that $d = 1111$ and $e = 1011$ so that $L_e = \{\varepsilon, 22, 222\}$. The data in the array memory is assumed to be the same observed examples. During this round, it is verified that L_{1011} contains 222 and 22.

```
(!#!#!2    :#    .2)     (!#!#!2        :#.2,#)
(!1!1!2    :1    ;2)     (!#!#!2        :1;2,2)
(!1!1!2  .2:1,2,#;2)     (!1!1!2  .2,2,#:1;2,2)
(!1!0!2  ;2:0,2,2;2)     (!1!0!2  ;2,2,2:0;2,2)
(!1!1!2.#;2:1,2,2;2)     (!1!1!2.#;2,2,2:1;2,2)
```

However, in the next round, the tests whether 22222 and 2222 are in L_e will both fail. Currently the hypothesis is ? anyway, as the copy of $e = 1011$ for checking has not yet reached the end of the array memory. $\square$

Proposition 23. *Let $\mathcal{L}$ be a strong-monotonically learnable automatic family. Now the following statements hold:*

(a) *$\mathcal{L}$ has a transduced iterative learner;*
(b) *$\mathcal{L}$ has a transduced learner which is consistent, conservative and strong-monotonic (the learner uses the special conjecture ?).*

Proof. The basic idea is to take an arbitrary recursive learner for the family and exploit the fact that whenever it conjectures a set L_e in the family then it has seen a finite subset H_e of L_e with the property: for all $L_d \supseteq H_e$, $L_d \supseteq L_e$. One can first-order define such a set H_e by a cut-off word (all words of L_e length-lexicographically up to the cut-off word are in H_e and no others) and chose the cut-off word as the minimal word for which all supersets in the family of the corresponding set H_e are also supersets of L_e. Thus, the mapping from e to the cut-off word is automatic, so again one can find a constant so that H_e can be chosen as the set of all words of L_e which overshoot the length of e by at most this constant. This permits to define the following basic learning algorithm which will then be translated into the corresponding settings. The learner maintains the current hypothesis c, an array memory which contains, besides the observed examples outside L_c, the bound d for searching and a candidate hypothesis e. The algorithm starts in step 2 with $e = d = \varepsilon$, where initially c is ? and $L_?$ is taken as being contained in every L_e.

1. If there are words in the array memory which are not in L_c then goto Step 2, else stay in Step 1 and monitor further data-items and do no update on those which are in L_c.
2. Check index e as outlined in Step 3 for the indices below d in length-lexicographical order and if none of them qualifies, then restart the search with a larger bound d.
3. An index e qualifies iff all data in H_e have been observed, all data in the array memory are in L_e and $L_c \subset L_e$.
4. If e qualifies then make the array memory empty and replace c by e and reset e to the initial value. Goto Step 1.

The breakdown of this algorithm into rounds of a transduced learner is very similar to what has been done for the preceding results.

For part (a), the hypothesis space consists of all possible memory values. An index in the hypothesis space accepts what the index c coded into memory accepts. Note that the learner will eventually find the correct index c and then not leave Step 1, as no new examples outside L_c are observed. The hypothesis is then stable and consists of some coding of c, d and e and an empty array memory.

For part (b), the hypothesis space is the given automatic family. The learner conjectures the index c when it is in Step 1, and ? otherwise. The conditions on revision and bookkeeping enforce that the learner is consistent. Furthermore, the learner is conservative, since it leaves Step 1 and conjectures ? only when a datum outside L_c has been observed. Strong-monotonicity is enforced by the way updates are realised. □

Note that monotonic learnability does not imply iterative learnability, as witnessed by the class consisting of the set $L_\varepsilon = \{0\}^*$ and for all $i, j \in \mathbb{N}$, the set $L_{1^{i+1}0^j} = \{\varepsilon, 0, 00, \ldots, 0^i\} \cup \{1^{i+1}, 0^{i+j+1}\}$. It is easy to see that this class is an automatic family. It has a monotonic learner, that can initially conjecture $\{0\}^*$, memorise the length n of the longest member of 0^* seen so far, and when receiving 1^{i+1} output a conjecture for $L_{1^{i+1}0^{n-i-1}}$. However, the class does not have an iterative learner: when first seeing only examples from $\{0\}^*$, the learner eventually stops memorising these examples and is then unable to recover the value of 0^{i+j+1} once it sees 1^{i+1}.

Proposition 24. *Every conservatively and iteratively learnable automatic family that is learnable using an automatic family as hypothesis space, has a conservative and iterative transduced learner.*

Proof. Suppose a recursive, conservative and iterative learner learns the automatic family $\mathcal{L} = \{L_e : e \in I\}$ using automatic family $\{H_e : i \in J\}$ as hypothesis space. Without loss of generality, assume that the learner is modified as follows. The input text is given to the learner on a one way read only tape. The learner keeps its current hypothesis at the left end of the work tape followed by a separator and then the rest of the tape. After it has read a word, the learner erases the work tape except for the new hypothesis and ends up in a special state. Call this modified learner $\mathbf{M}$.

Let T be a text. Note that if (i) the hypothesis of $\mathbf{M}$ on $T[n]$ is d, (ii) H_d contains $content(T[n])$ and (iii) either $T(n) = \#$ or $T(n)$ is in H_d, then $\mathbf{M}$ does not change its hypothesis when receiving $T(n)$. Furthermore, the limiting conjecture output by $\mathbf{M}$ on T is the same as the limiting conjecture output by $\mathbf{M}$ on the text obtained from T by dropping $T(n)$. This is utilised by the following transduced learner $\mathbf{N}$ that does not record $T(n)$ in its memory.

The hypothesis of $\mathbf{N}$ is its memory; the corresponding language will be described below. Here is the description of how $\mathbf{N}$ maintains its memory.

(a) Besides the work tape configuration and the state of $\mathbf{M}$, $\mathbf{N}$ keeps track of data to simulate $\mathbf{M}$ (as indicated above, some data will be omitted).
(b) The array memory also contains a pointer to the datum t which the simulation of $\mathbf{M}$ is currently reading. This pointer has an entry in every row of the array memory. One of the entries is marked as the symbol from t that is currently processed (indeed, only one symbol is read per round).
(c) The array memory has either only one hypothesis e at the end of its memory or two hypotheses d and e. In the second case, the hypothesis H_e is known not to contain the input data and d is the latest hypothesis in the simulation of $\mathbf{M}$.
(d) The current hypothesis of $\mathbf{N}$ is its memory, the language it represents is H_e.
(e) The array memory also contains a bit b_e such that $b_e = 1$ iff all data in the array memory are in H_e.
(f) Hypothesis d, if present, is moved in the array memory from left to right in order to check whether all stored data are in H_d.

The main idea behind the workings of $\mathbf{N}$ is as follows.

($*$) While the current hypothesis of $\mathbf{M}$ is for a set H_e not containing all data archived so far (as indicated by b_e), do the following two actions in parallel.

- Archive all data observed in the array memory.
- Continue simulation of $\mathbf{M}$ until a new hypothesis d is found. Then check whether H_d contains all data archived so far and if so, replace e by d. Otherwise, drop d and continue with the simulation of $\mathbf{M}$. Again, checking whether L_d contains all data in the archive is done at double speed to eventually catch up with the input data.

Once the latest hypothesis has been found out to contain all archived data, the learner keeps observing incoming data and leaves its memory unchanged. The simulation of $\mathbf{M}$ is halted until a datum x not in the latest hypothesis L_e is observed; when this happens, x is archived, b_e is set to 0, and the simulation of $\mathbf{M}$ is resumed as described by ($*$) above.

More precisely, in each round, on a new data item x, $\mathbf{N}$ updates its array memory and the configuration of $\mathbf{M}$ as follows.

1. If $b_e = 1$ and $x \in L_e$, then x is discarded and goto the next round.
   Else $b_e = 0$ or $x \notin L_e$, x is appended to the end of the array memory, b_e is set to 0 and the following steps are executed (over several rounds).
2. If there is currently only one hypothesis e in the array memory, then $b_e = 0$ and the following is done.
   - If the simulation of **M** has currently not finished with its round of computation, then one more step of the simulation of **M** is done.
   - If the simulation of **M** has concluded a round of computation (by moving to a special state as mentioned above), then the hypothesis d of **M** is copied (during several rounds) from the beginning of the Turing tape to the first column of the array memory. Then goto step 3, suspending the simulation of **M** until restarted below.

   If during the above simulation of **M**, a data symbol needs to be read, then the symbol currently pointed to by the pointer in the array memory is read, and the pointer is adjusted accordingly. If the pointer indicates that the word is read completely then it is moved over the data item to the next data item.
3. If two hypotheses e and d exist and d is inside the array memory, then d is moved over the next two data items. In case at least one of the data items is not in L_d, the hypothesis d is dropped from the array memory (and the simulation of **M** is continued as in 2 above).
4. If d reaches the end of the array memory and is next to e (thus, **N** has verified that all data items in the array memory are contained in L_d), then e is replaced by d and b_e is replaced by 1.
5. The hypothesis of **N** is taken to be the full memory content (as **N** has to be iterative) and the set conjectured by this padded hypothesis is the set H_e of the (possibly updated) e at the end of the array memory.

As long as **M** does not find the correct hypothesis in the limit, it conjectures some intermediate hypotheses H_e. As long as this hypothesis is consistent with the observed data, **M** ignores the incoming data due to conservativeness. This is exploited by the algorithm and justifies that at some stages, **N** does not archive incoming data consistent with the current hypothesis L_e. However, if **M** has not yet converged to the correct hypothesis, a datum outside L_e will eventually come up and then the simulation of **M** resumes until **M** outputs a new hypothesis which becomes d in the array memory. In the next stages, it is checked whether L_d contains all data stored in the array memory. If L_d is not a correct hypothesis, then the above process repeats until a correct final hypothesis is reached.

The learner **N** is conservative because whenever it conjectures L_e, either some datum outside L_e has already been archived and thus it can go on with simulation of **M** / checking of the latest hypothesis d, or it ignores all members of L_e until a non-member is observed. **N** is also iterative since it conjectures its full memory, equivalent to the hypothesis e.

For illustration purposes, here are some diagrammes that depict how **N** updates its memory. With this example, the words archived in the array memory are 33, 222, 333, 2223, 22 and 22333 and the languages learnt are the finite unions of sets of the form $\{2\}^i \cdot \{3\}^*$. These finite unions are represented by hypotheses from $\{\varepsilon\} \cup \{0,1\}^* \cdot \{1\}$ where the set $L_{a_0 a_1 \ldots a_n}$ is the union of all sets $\{2\}^i \cdot \{3\}^*$ with $a_i = 1$. The symbol preceding the memory is b_e for the hypothesis $e = 11$.

The table shows the memory before and after a round where 23333 is read, illustrating a kind of update described in the proof. The read head is a column of `+` and `-` where the `+` are below and the `-` are above the current read position. One symbol is read by **M** during this round.

```
 (    -:#      .3:#)       (    -    :#  .3,3:#)
 (    -:1  .3,#;3:#)       (    -  .3:1,#;3,3:#)
 (  .2-:1,3;2,#;3:#)       (  .2-,3;2:1,#;3,3:#)
 (.3;2-:0,3;2,2;2:1)       (.3;2+,3;2:0,2;2,3:1)
0(;3;2+:1,3;2,2;2:1)      0(;3;2+,3;2:1,2;2,2:1)
```

This completes the proof. □

9 Conclusion and Open Problems

It was demonstrated that whereas many questions on memory usage remain open for automatic learners, transduced learners that learn a transduced family can always bound the memory size as a function of the longest datum seen so far. When learning automatic families, concrete bounds have been found and a hierarchy of polynomial and exponential bounds has emerged. It has been shown that every learnable family can be learnt using some exponential bound.

Furthermore, additional properties such as iterativeness, consistency and conservativeness were considered. Many proofs required using the special symbol ? for intermediate revisions, because computations need to be delayed and cannot commit in one round to a new valid hypothesis. This somehow parallels Pitt's delaying tricks in standard inductive inference [43]. In order to get the full power of iterative learning, a strongly padded hypothesis space had to be used. Still, the existence of iterative or consistent and conservative learners was mainly proved for automatic families. For transduced families, proof of existence is still open.

Open Problem 25. *The following problems on transduced learners are open.*

(a) *Is every learnable transduced family consistently, conservatively, transducedly learnable?*
(b) *Is every strong-monotonically learnable transduced family iteratively, transducedly learnable? In particular, is the transduced family* $\{L_e = \{0,1\}^* \cdot e \cdot \{0,1\}^* : e \in \{0,1\}^*\}$ *iteratively, transducedly learnable?*

The particular case of Problem 25 (b) is a concrete example where a strong-monotonic learner exists, but an iterative transduced learner might not exist. This is because the methods that convert strong-monotonic learners into iterative transduced ones need the learner to be able to verify whether a datum is not in the current hypothesis L_e. While the subword property can be captured by a non-deterministic transducer, its complement cannot.

Besides the question of the extent to which results for the learning of automatic families by transduced learners carry over to transduced families, two open problems naturally present themselves, that challenge the understanding of iterative learnability of automatic families. In particular, if one could find a good characterisation of iterative learnability of automatic families by recursive learners, one might get a handle towards the solution of both questions.

Open Problem 26. *The following problems about learning automatic families are open.*

(a) *Is every iteratively learnable automatic family recursively, iteratively, conservatively learnable?*
(b) *Does every iteratively learnable automatic family have a transduced iterative learner?*

Note that an affirmative answer to Open Problem 26 (a) using an automatic family as hypothesis space implies a positive answer to Open Problem 26 (b).

10 Acknowledgements

The authors would like to thank the anonymous referees for several helpful comments which improved the presentation of the paper.

References

1. Dana Angluin. Inductive inference of formal languages from positive data. Information and Control 45:117–135, 1980.
2. Dana Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.
3. Jean Berstel. *Transductions and Context-Free Languages.* Springer Vieweg, Heidelberg, 1979.
4. Janis Bārzdiņš. Inductive inference of automata, functions and programs. *Proceedings of the 20th International Congress of Mathematicians, Vancouver*, pages 455–460, 1974. In Russian. English translation in American Mathematical Society Translations: Series 2, 109:107-112, 1977.
5. Lenore Blum and Manuel Blum. Toward a mathematical theory of inductive inference. *Information and Control*, 28:125–155, 1975.
6. Achim Blumensath. *Automatic structures.* Diploma thesis, Department of Computer Science, RWTH Aachen, 1999.
7. Achim Blumensath and Erich Grädel. Automatic structures. *15th Annual IEEE Symposium on Logic in Computer Science*, LICS 2000, pages 51–62, 2000.
8. John Case and Chris Lynes. Machine inductive inference and language identification. *Proceedings of the 9th International Colloquium on Automata, Languages and Programming*, ICALP 1982. Springer LNCS 140:107–115, 1982.
9. John Case, Sanjay Jain, Trong Dao Le, Yuh Shin Ong, Pavel Semukhin and Frank Stephan. Automatic learning of subclasses of pattern languages. *Language and Automata Theory and Applications,* LATA 2011. Proceedings. Springer LNCS 6638:192–203, 2011.
10. John Case, Sanjay Jain, Samuel Seah and Frank Stephan. Automatic Functions, Linear Time and Learning. *Logical Methods in Computer Science*, 9(3), 2013.

11. John Case and Timo Kötzing. Difficulties in forcing fairness of polynomial time inductive inference. *Algorithmic Learning Theory*, Twentieth International Conference, ALT 2009, Proceedings. Springer LNAI, 5809:263–277, 2009.
12. John Case and Timo Kötzing. Strongly non-U-shaped learning results by general techniques. Proceedings of the Twenty third International Conference on *Computational Learning Theory*, COLT 2010, Proceedings. Pages 181-193, Omnipress 2010.
13. John Case and Carl H. Smith. Comparison of identification criteria for machine inductive inference. *Theoretical Computer Science*, 25:193–220, 1983.
14. Patrick C. Fischer and Arnold L. Rosenberg. Multitape one-way nonwriting automata. *Journal of Computer and System Sciences*, 2(1):88–101, 1968.
15. Rūsiņš Freivalds, Efim Kinber and Carl H. Smith. On the impact of forgetting on learning machines. *Journal of the ACM*, 42:1146–1168, 1995.
16. E. Mark Gold. Language identification in the limit. *Information and Control* 10:447–474, 1967.
17. Bernard R. Hodgson. *Théories décidables par automate fini.* Ph.D. thesis, Département de mathématiques et de statistique, Université de Montréal, 1976.
18. Bernard R. Hodgson. Décidabilité par automate fini. *Annales des sciences mathématiques du Québec*, 7(1):39–57, 1983.
19. John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Second Edition, Addison-Wesley, 2001.
20. Klaus-Peter Jantke. Monotonic and non-monotonic inductive inference. *New Generation Computing*, 8:349–360, 1991.
21. Sanjay Jain, Timo Kötzing, Junqi Ma and Frank Stephan. On the role of update constraints and text-types in iterative learning. *Information and Computation*, 247:152-168, 2016.
22. Sanjay Jain, Shao Ning Kuek, Eric Martin and Frank Stephan. Learners based on transducers. *Languages and Automata Theory and Applications*, LATA 2018, 12 pages, Springer, to appear, 2018.
23. Sanjay Jain, Qinglong Luo and Frank Stephan. Learnability of automatic classes. *Journal of Computer and System Sciences*, 78:1910–1927, 2012. Special issue on LATA 2010.
24. Sanjay Jain, Yuh Shin Ong, Shi Pu and Frank Stephan. On automatic families. *Proceedings of the eleventh Asian Logic Conference* in honour of Professor Chong Chitat on his sixtieth birthday, pages 94–113, World Scientific, 2012.
25. Sanjay Jain, Daniel N. Osherson, James S. Royer and Arun Sharma. *Systems That Learn.* MIT Press, Second Edition, 1999.
26. Tao Jiang, Arto Salomaa, Kai Salomaa and Sheng Yu. Inclusion is undecidable for pattern languages. *International Colloquium on Automata, Languages and Programming*, ICALP 1993, *Springer LNCS*, 700:301–312, 1993.
27. Bakhadyr Khoussainov and Mia Minnes. Three Lectures on Automatic Structures. *Proceedings of the ASL Logic Colloquium 2007*, *Lecture Notes in Logic*, 35:132–176, 2010.

28. Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. *Logical and Computational Complexity*, International Workshop LCC 1994. Springer LNCS 960:367–392, 1995.
29. Bakhadyr Khoussainov and Anil Nerode. *Automtata Theory and its Applications.* Birkhäuser, 2001.
30. Bakhadyr Khoussainov and Anil Nerode. Open Questions in the Theory of Automatic Structures. *Bulletin of the European Association of Theoretical Computer Science* (EATCS), no. 94, pp. 181–204, 2008.
31. Bakhadyr Khoussainov, Andre Nies, Sasha Rubin and Frank Stephan. Automatic structures: richness and limitations. *Logical Methods in Computer Science*, 3(2:2)1–18, 2007.
32. Efim Kinber and Frank Stephan. Language learning from texts: mind changes, limited memory and monotonicity. *Information and Computation*, 123:224–241, 1995.
33. Shao Ning Kuek. *Transduced Learners.* Honours Year Project, National University of Singapore, Submitted, March 2017.
34. Dietrich Kuske. Is Cantor's theorem automatic? *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 22 September 2003, pp. 332–345. Springer, Berlin, Heidelberg, 2003.
35. Steffen Lange, Thomas Zeugmann and Sandra Zilles. Learning indexed families of recursive languages from positive data: a survey. *Theoretical Computer Science*, 397:194–232, 2008.
36. Gennadiy Semenovich Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR*, Sbornik 32:129–198, 1977.
37. George H. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, 34(5):1045-1079, 1955.
38. Edward F. Moore. Gedanken Experiments on sequential machines. *Automata Studies*, edited by C.E. Shannon and John McCarthy, *Princeton University Press*, Princeton, New Jersey, 1956.
39. Maurice Nivat. Transductions des langages de Chomsky. *Annales de l'institut Fourier*, Grenoble, 18:339–455, 1968.
40. Piergiorgio Odifreddi. *Classical Recursion Theory*, Volume II. Studies in Logic and the Foundations of Mathematics, 143. Elsevier, 1999.
41. Daniel Osherson, Michael Stob, and Scott Weinstein. Learning strategies. *Information and Control*, 53:32–51, 1982.
42. Daniel Osherson, Michael Stob and Scott Weinstein. *Systems That Learn, An Introduction to Learning Theory for Cognitive and Computer Scientists.* Bradford — The MIT Press, Cambridge, Massachusetts, 1986.
43. Leonard Pitt. Inductive inference, DFAs, and computational complexity. *Analogical and Inductive Inference, Proceedings of the Second International Workshop*, AII 1989. Springer LNAI 397:18–44, 1989.
44. Sasha Rubin. Automata presenting structures: a survey of the finite string case. *The Bulletin of Symbolic Logic*, 14:169–209, 2008.

45. Sasha Rubin. Automatic structures. In Jean-Éric Pin (ed), *Handbook of Automata Theory*. To appear.
46. Ray Solomonoff. A formal theory of inductive inference, part I. *Information and Control*, 7(1):1–22, 1964.
47. Ray Solomonoff. A formal theory of inductive inference, part II. *Information and Control*, 7(2):224–254, 1964.
48. Frank Stephan. Automatic structures – recent results and open questions. *Third International Conference on Science and Engineering in Mathematics, Chemistry and Physics*, ScieTech 2015 (pages 121–130 in proceedings booklet), *Journal of Physics: Conference Series*, 622/1 (Paper 012013) 2015.
49. Frank Stephan. *Methods and Theory of Automata and Languages*. School of Computing, National University of Singapore, 2016.
50. Rolf Wiehagen. Limes-Erkennung rekursiver Funktionen durch spezielle Strategien. *Journal of Information Processing and Cybernetics (EIK)*, 12(1–2):93–99, 1976.
51. Rolf Wiehagen. A thesis in inductive inference. *Proceedings of the 1st Workshop on Nonmonotonic and Inductive Logic*. Springer LNCS 543:184–207, 1990.