What You Must Remember When Transforming Datawords

M. Praveen
Chennai Mathematical Institute, India
UMI ReLaX, Indo-French joint research unit

Abstract

Streaming Data String Transducers (SDSTs) were introduced to model a class of imperative and a class of functional programs, manipulating lists of data items. These can be used to write commonly used routines such as insert, delete and reverse. SDSTs can handle data values from a potentially infinite data domain. The model of Streaming String Transducers (SSTs) is the fragment of SDSTs where the infinite data domain is dropped and only finite alphabets are considered. SSTs have been much studied from a language theoretical point of view. We introduce data back into SSTs, just like data was introduced to finite state automata to get register automata. The result is Streaming String Register Transducers (SSRTs), which is a subclass of SDSTs.

We use origin semantics for SSRTs and give a machine independent characterization, along the lines of Myhill-Nerode theorem. Machine independent characterizations for similar models are the basis of learning algorithms and enable us to understand fragments of the models. Origin semantics of transducers track which positions of the output originate from which positions of the input. Although a restriction, using origin semantics is well justified and is known to simplify many problems related to transducers. We use origin semantics as a technical building block, in addition to characterizations of deterministic register automata. However, we need to build more on top of these to overcome some challenges unique to SSRTs.

2012 ACM Subject Classification Theory of computation → Transducers; Theory of computation → Automata over infinite objects

Keywords and phrases Streaming String Transducers, Data words, Machine independent characterization

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2020.55

Related Version A full version of the extended abstract is available at `https://arxiv.org/abs/2005.02596`.

Funding *M. Praveen*: Partially supported by a grant from the Infosys foundation.

Acknowledgements The author thanks C. Aiswarya, Kamal Lodaya, K. Narayan Kumar and anonymous reviewers for suggestions to improve the presentation and pointers to related works.

1 Introduction

Transductions are in general relations among words. Transducers are theoretical models that implement transductions. Transducers are used in a variety of applications, such as analysis of web sanitization frameworks, host based intrusion detection, natural language processing, modeling some classes of programming languages and constructing programming language tools like evaluators, type checkers and translators. Streaming Data String Transducers (SDSTs) were introduced in [2] to model a class of imperative and a class of functional programs, manipulating lists of data items. Transducers have been used in [16] to infer semantic interfaces of data structures such as stacks. Such applications use Angluin style learning, which involves constructing transducers by looking at example operations of the object under study. Since the transducer is still under construction, we need to make inferences about the transduction without having access to a transducer which implements it. Theoretical bases for doing this are machine independent characterizations, which identify what kind of transductions can be implemented by what kind of transducers and give a

template for constructing transducers. Indeed the seminal Myhill-Nerode theorem gives a machine independent characterization for regular languages over finite alphabets, which form the basis of Angluin style learning of regular languages [3]. A similar characterization for a fragment of SDSTs is given in [5] and is used as a basis to design a learning algorithm.

Programs deal with data from an infinite domain and transducers modeling the programs should also treat data as such. For example in [16], the state space reduced from 10^9 to 800 and the number of learning queries reduced from billions to 4000 by switching to a transducer model that can deal with data from an infinite domain. We give a machine independent characterization for a fragment of SDSTs more powerful than those in [16, 5]. The additional power comes from significant conceptual differences. The transducers used in [16] produce the output in a linear fashion without remembering what was output before. For example, they cannot output the reverse of the input strings, which can be done by our model. The model studied in [5] are called Streaming String Transducers (SSTs), the fragment obtained from SDSTs by dropping the ability to deal with data values from an infinite domain. We retain this ability in our model, called Streaming String Register Transducers (SSRTs). It is obtained from SDSTs by dropping the ability to deal with linear orders in the data domain. Apart from Angluin style learning algorithms, machine independent characterizations are also useful for studying fragments of transducer models. E.g. in [5], machine independent characterization of SSTs is used to study fragments such as non-deterministic automata with output and transductions definable in First Order logic.

We use origin semantics of transducers, which are used in [5] to take into account how positions of the output originate from the positions of the input. Using origin semantics is known to ease some of the problems related to transducers, e.g., [7]. Origin semantics is a restriction, but a reasonable one and is used extensively in this paper.

Contributions

Machine independent characterizations are known for automata over data values from an infinite domain [15, 4] and for streaming transducers over finite alphabets [5], but not for streaming transducers over data values, which is what we develop here. This involves both conceptual and technical challenges. In [15, 4], data values that must be remembered by an automaton while reading a word from left to right are identified using a machine independent definition. We lift this to transducers and identify that the concept of factored outputs from [5] is necessary for this. Factored outputs can let us ignore some parts of transduction outputs, which is necessary to define when two words behave similarly. However, [5] does not deal with data values from an infinite domain and it takes quite a bit of manipulation with permutations on data values to make ideas from there work here. In transductions, suffixes can influence how prefixes are transformed. This is elegantly handled in [5] using two way transducer models known to be equivalent to SSTs. There are no such models known when data values are present. To handle it in a one way transducer model, we introduce data structures based on trees that keep track of all possible suffixes. This does raise the question of whether there are interesting two way transducer models with data values. Recent work [6] has made progress in this direction, which we discuss at the end of this extended abstract. We concentrate here on SDSTs and its fragments, which are known to be equivalent to classes of imperative and functional programming languages. In [2], it is explained in detail which features of programming languages correspond to which features of the transducer. Over finite alphabets, streaming string transducers are expressively equivalent to regular transductions, which are also defined by two way deterministic finite-state transducers and by monadic second order logic [1].

Related Works

Studying transducer models capable of handling data values from an infinite domain is an active area of research [13, 14]. Streaming transducers like SDSTs have the distinctive feature of using variables to store intermediate values while computing transductions; this idea appears in an earlier work [11] that introduced *simple programs on strings*, which implement the same class of transductions as those implemented by SSTs. An Angluin style learning algorithm for deterministic automata with memory is given in [17]. A machine independent characterization of automata with finite memory is given in [8], which is further extended to data domains with arbitrary binary relations in [9]. The learning algorithm of [17] is extended to Mealy machines with data in [16]. However, Mealy machines are not as powerful as SSRTs that we consider here. Using a more abstract approach of nominal automata, [19] presents a learning algorithm for automata over infinite alphabets. Logical characterizations of transducers that can handle data are considered in [12]. However, the transducers in that paper cannot use data values to make decisions, although they are part of the output. Register automata with linear arithmetic introduced in [10] shares some of the features of the transducer model used here. Here, data words stored in variables can be concatenated, while in register automata with linear arithmetic, numbers stored in variables can be operated upon by linear operators.

Most proofs and some technical details in this extended abstract are skipped due to space constraints. All the proofs and technical details can be found in the full version.

2 Preliminaries

Let $\mathbb{I}$ be the set of integers, $\mathbb{N}$ be the set of non-negative integers and D be an infinite set of data values. We will refer to D as the *data domain*. For $i, j \in \mathbb{I}$, we denote by $[i, j]$ the set $\{k \mid i \leq k \leq j\}$. For any set S, S^* denotes the set of all finite sequences of elements from S. The empty sequence is denoted by ϵ. Given $u, v \in S^*$, v is a *prefix* (resp. suffix) of u if there exists $w \in S^*$ such that $u = vw$ (resp. $u = wv$). The sequence v is an *infix* of u if there are sequences w_1, w_2 such that $u = w_1 v w_2$.

Let Σ, Γ be finite alphabets. We will use Σ for input alphabet and Γ for output alphabet. A *data word* over Σ is a word in $(\Sigma \times D)^*$. A *data word with origin information* over Γ is a word in $(\Gamma \times D \times \mathbb{N})^*$. Suppose $\Sigma = \{\text{title}, \text{firstName}, \text{lastName}\}$ and $\Gamma = \{\text{givenName}, \text{surName}\}$. An example data word over Σ is $(\text{title}, \text{Mr.})(\text{firstName}, \text{Harry})(\text{lastName}, \text{Tom})$. If we were to give this as input to a device that reverses the order of names, the output would be the data word with origin information $(\text{surName}, \text{Tom}, 3)(\text{givenName}, \text{Harry}, 2)$, over Γ. In the triple $(\text{givenName}, \text{Harry}, 2)$, the third component 2 indicates that the pair (givenName, Harry) originates from the second position of the input data word. We call the third component *origin* and it indicates the position in the input that is responsible for producing the output triple. If a transduction is being implemented by a transducer, the origin of an output position is the position of the input that the transducer was reading when it produced the output. The data value at some position of the output may come from any position (not necessarily the origin) of the input data word. We write *transduction* for any function from data words over Σ to data words with origin information over Γ.

For a data word w, $|w|$ is its length. For a position $i \in [1, |w|]$, we denote by $\texttt{data}(w, i)$ (resp. $\texttt{letter}(w, i)$) the data value (resp. the letter from the finite alphabet) at the i^{th} position of w. We denote by $\texttt{data}(w, *)$ the set of all data values that appear in w. For positions $i \leq j$, we denote by $w[i, j]$ the infix of w starting at position i and ending at position j. Note that $w[1, |w|] = w$. Two data words w_1, w_2 are *isomorphic* (denoted by

$w_1 \simeq w_2$) if $|w_1| = |w_2|$, $\texttt{letter}(w_1, i) = \texttt{letter}(w_2, i)$ and $\texttt{data}(w_1, i) = \texttt{data}(w_1, j)$ iff $\texttt{data}(w_2, i) = \texttt{data}(w_2, j)$ for all positions $i, j \in [1, |w_1|]$. For data values d, d', we denote by $w[d/d']$ the data word obtained from w by replacing all occurrences of d by d'. We say that d' is a *safe replacement* for d in w if $w[d/d'] \simeq w$. Intuitively, replacing d by d' doesn't introduce new equalities/inequalities among the positions of w. For example, d_1 is a safe replacement for d_2 in $(a, d_3)(b, d_2)$, but not in $(a, d_1)(b, d_2)$.

A permutation on data values is any bijection $\pi : D \to D$. For a data word u, $\pi(u)$ is obtained from u by replacing all its data values by their respective images under π. A transduction f is *invariant under permutations* if for every data word u and every permutation π, $f(\pi(u)) = \pi(f(u))$ (permutation can be applied before or after the transduction).

Suppose a transduction f has the property that for any triple (γ, d, o) in any output $f(w)$, there is a position $i \leq o$ in w such that $\texttt{data}(w, i) = d$. If the data value d is output from the origin o, then d should have already occurred in the input on or before o. Such transductions are said to be *without data peeking*. We say that a transduction has *linear blow up* if there is a constant K such that for any position o of any input, there are at most K positions in the output whose origin is o.

Streaming String Register Transducers

We present an extension of SSTs to handle data values, just like finite state automata were extended to finite memory automata [18]. Our model is a subclass of SDSTs, which can store intermediate values (which can be long words) in variables. E.g., reversing an input word can be achieved as follows: as each input symbol is read, concatenate it to the back of a variable maintained for this purpose. At the end, the variable will have the reverse of the input. There are also registers in these models, which can store single data values. Transitions can be enabled/disabled based on whether the currently read data value is equal/unequal to the one stored in one of the registers.

▶ **Definition 1.** *A* Streaming String Register Transducer *(SSRT) is an eight tuple* $S = (\Sigma, \Gamma, Q, q_0, R, X, O, \Delta)$*, where*

- *the finite alphabets* Σ, Γ *are used for input, output respectively,*
- Q *is a finite set of states,* q_0 *is the initial state,*
- R *is a finite set of registers and* X *is a finite set of data word variables,*
- $O : Q \rightharpoonup ((\Gamma \times \hat{R}) \cup X)^*$ *is a partial output function, where* $\hat{R} = R \cup \{curr\}$*, with* $curr$ *being a special symbol used to denote the current data value being read and*
- $\Delta \subseteq (Q \times \Sigma \times \Phi \times Q \times 2^R \times U)$ *is a finite set of transitions. The set* Φ *consists of all Boolean combinations of atomic constraints of the form* $r^{=}$ *or* $r^{\neq}$ *for* $r \in R$*. The set* U *is the set of all functions from the set* X *of data word variables to* $((\Gamma \times \hat{R}) \cup X)^*$*.*

It is required that

- *For every* $q \in Q$ *and* $x \in X$*, there is at most one occurrence of* x *in* $O(q)$ *and*
- *for every transition* $(q, \sigma, \phi, q', R', ud)$ *and for every* $x \in X$*,* x *appears at most once in the set* $\{ud(y) \mid y \in X\}$*.*

We say that the last two conditions above enforce a SSRT to be *copyless*, since it prevents multiple copies of contents being made.

A *valuation* val for a transducer S is a partial function over registers and data word variables such that for every register $r \in R$, either $val(r)$ is undefined or is a data value in D, and for every data word variable $x \in X$, $val(x)$ is a data word with origin information over Γ. The valuation val and data value d satisfies the atomic constraint $r^{=}$ (resp. $r^{\neq}$) if $val(r)$ is defined and $d = val(r)$ (resp. undefined or $d \neq val(r)$). Satisfaction is extended to

Boolean combinations in the standard way. We say that a SSRT is *deterministic* if for every two transitions $(q, \sigma, \phi, q', R', u)$ and $(q, \sigma, \phi', q'', R'', u')$ with the same source state q and input symbol σ, the formulas ϕ and ϕ' are mutually exclusive (i.e., $\phi \wedge \phi'$ is unsatisfiable). We consider only deterministic SSRTs here.

A configuration is a triple (q, val, i) where $q \in Q$ is a state, val is a valuation and i is the number of symbols read so far. The transducer starts in the configuration $(q_0, val_\epsilon, 0)$ where q_0 is the initial state and val_ϵ is the valuation such that $val_\epsilon(r)$ is undefined for every register $r \in R$ and $val_\epsilon(x) = \epsilon$ for every data word variable $x \in X$. From a configuration (q, val, i), the transducer can read a pair $(\sigma, d) \in \Sigma \times D$ and go to the configuration $(q', val', i+1)$ if there is a transition $(q, \sigma, \phi, q', R', ud)$ and 1) d and val satisfies ϕ and 2) val' is obtained from val by assigning d to all the registers in R' and for every $x \in X$, setting $val'(x)$ to $ud(x)[y \mapsto val(y), (\gamma, curr) \mapsto (\gamma, d, i+1), (\gamma, r) \mapsto (\gamma, val(r), i+1)]$ (in $ud(x)$, replace every occurrence of y by $val(y)$ for every data word variable $y \in X$, replace every occurrence of $(\gamma, curr)$ by $(\gamma, d, i+1)$ for every output letter $\gamma \in \Gamma$ and replace every occurrence of (γ, r) by $(\gamma, val(r), i+1)$ for every output letter $\gamma \in \Gamma$ and every register $r \in R$). After reading a data word w, if the transducer reaches some configuration (q, val, n) and $O(q)$ is not defined, then the transducer's output $[\![S]\!](w)$ is undefined for the input w. Otherwise, the transducer's output is defined as $[\![S]\!](w) = O(q)[y \mapsto val(y), (\gamma, curr) \mapsto (\gamma, d, n), (\gamma, r) \mapsto (\gamma, val(r), n)]$, where d is the last data value in w.

Intuitively, the transition $(q, \sigma, \phi, q', R', ud)$ checks that the current valuation val and the data value d being read satisfies ϕ, goes to the state q', stores d into the registers in R' and updates data word variables according to the update function ud. The condition that x appears at most once in the set $\{ud(y) \mid y \in X\}$ ensures that the contents of any data word variable are not duplicated into more than one variable. This ensures, among other things, that the length of the output is linear in the length of the input. The condition that for every two transitions $(q, \sigma, \phi, q', R', ud)$ and $(q, \sigma, \phi', q'', R'', ud')$ with the same source state and input symbol, the formulas ϕ and ϕ' are mutually exclusive ensures that the transducer cannot reach multiple configurations after reading a data word (i.e., the transducer is deterministic).

▶ **Example 2.** Consider the transduction that is the identity on inputs in which the first and last data values are equal. On the remaining inputs, the output is the reverse of the input. This can be implemented by a SSRT using two data word variables. As each input symbol is read, it is appended to the front of the first variable and to the back of the second variable. The first variable stores the input and the second one stores the reverse. At the end, either the first or the second variable is output, depending on whether the last data value is equal or unequal to the first data value (which is stored in a register).

In Section 3, we define an equivalence relation on data words and state our main result in terms of the finiteness of the index of the equivalence relation and a few other properties. In Section 4, we prove that transductions satisfying certain properties can be implemented by SSRTs (the backward direction of the main result) and we prove the converse in Section 5.

3 How Prefixes and Suffixes Influence Each Other

As is usual in many machine independent characterizations (like the classic Myhill-Nerode theorem for regular languages), we define an equivalence relation on the set of data words to identify similar ones. If the equivalence relation has finite index, it can be used to construct finite state models. We start by looking at what "similar data words" mean in the context of transductions.

Suppose L is the set of all even length words over some finite alphabet. The words a and aaa do the same thing to any suffix v: $a \cdot v \in L$ iff $aaa \cdot v \in L$. So, a and aaa are identified to be similar with respect to L in the classic machine independent characterization. Instead of a language L, suppose we have a transduction f and we are trying to identify words u_1, u_2 that do the same thing to any suffix v. The naive approach would be to check if $f(u_1 \cdot v) = f(u_2 \cdot v)$, but this does not work. Suppose a transduction f is such that $f(a \cdot b) = (a,1)(b,2)$, $f(aaa \cdot b) = (a,1)(a,2)(a,3) \cdot (b,4)$ and $f(c \cdot b) = (c,1)(b,2)(b,2)$ (we have ignored data values in this transduction). The words a and aaa do the same thing to the suffix b (the suffix is copied as it is to the output), as opposed to c (which copies the suffix twice to the output). But $f(a \cdot b) \neq f(aaa \cdot b)$. The problem is that we are not only comparing what a and aaa do to the suffix b, but also comparing what they do to themselves. We want to indicate in some way that we want to ignore the parts of the output that come from a or aaa: $f(\underline{a} \mid v) = \texttt{left} \cdot (b,2)$ and $f(\underline{aaa} \mid b) = \texttt{left} \cdot (b,4)$. We have underlined a and aaa on the input side to indicate that we want to ignore them; we have replaced a and aaa in the output by `left` to indicate that they are coming from ignored parts of the input. This has been formalized as factored outputs in [5]. This is still not enough for our purpose, since the outputs $(b,2)$ and $(b,4)$ indicate that a and aaa have different lengths. This can be resolved by offsetting one of the outputs by the difference in the lengths: $f(\underline{a} \mid v) = \texttt{left} \cdot (b,2) = f_{-2}(\underline{aaa} \mid b)$. The subscript -2 in $f_{-2}(\underline{aaa} \mid b)$ indicates that we want to offset the origins by -2. We have formalized this in the definition below, in which we have borrowed the basic definition from [5] and added data values and offsets.

▶ **Definition 3** (Offset factored outputs). *Suppose f is a transduction and uvw is a data word over Σ. For a triple (γ, d, o) in $f(uvw)$, the* abstract origin $\texttt{abs}(o)$ *of o is* `left` *(resp.* `middle`*,* `right`*) if o is in u (resp. v, w). The factored output $f(\underline{u} \mid v \mid w)$ is obtained from $f(uvw)$ by first replacing every triple (γ, d, o) by $(*, *, \texttt{abs}(o))$ if $\texttt{abs}(o) = \texttt{left}$ (the other triples are retained without change). Then all consecutive occurrences of $(*, *, \texttt{left})$ are replaced by a single triple $(*, *, \texttt{left})$ to get $f(\underline{u} \mid v \mid w)$. Similarly we get $f(u \mid \underline{v} \mid w)$ and $f(u \mid v \mid \underline{w})$ by using $(*, *, \texttt{middle})$ and $(*, *, \texttt{right})$ respectively. We get $f(\underline{u} \mid v)$ and $f(u \mid \underline{v})$ similarly, except that there is no middle part. For an integer z, we obtain $f_z(\underline{u} \mid v)$ by replacing every triple (γ, d, o) by $(\gamma, d, o + z)$ (triples $(*, *, \texttt{left})$ are retained without change).*

Let $w = (a,d_1)(a,d_2)(b,d_3)(c,d_4)$ and f be the transduction in Example 2. Then $f(w) = (c,d_4,4)(b,d_3,3)(a,d_2,2)(a,d_1,1)$ (assuming $d_4 \neq d_1$). The factored output $f(\underline{(a,d_1)(a,d_2)} \mid (b,d_3) \mid (c,d_4))$ is $(c,d_4,4)(b,d_3,3)(*,*,\texttt{left})$.

It is tempting to say that two data words u_1 and u_2 are equivalent if for all v, $f(\underline{u_1} \mid v) = f_z(\underline{u_2} \mid v)$, where $z = |u_1| - |u_2|$. But this does not work; continuing with the transduction f from Example 2, no two data words from the infinite set $\{(a,d_i) \mid i \geq 1\}$ would be equivalent: $f(\underline{(a,d_i)} \mid (a,d_i)) \neq f(\underline{(a,d_j)} \mid (a,d_i))$ for $i \neq j$. To get an equivalence relation with finite index, we need to realize that the important thing is not the first data value, but its (dis)equality with the last one. So we can say that for every i, there is a permutation π_i on data values mapping d_i to d_1 such that $f(\underline{\pi_i(a,d_i)} \mid v) = f(\underline{(a,d_1)} \mid v)$. This will get us an equivalence relation with finite index but it is not enough, since the transducer model we build must satisfy another property: it must use only finitely many registers to remember data values. Next we examine which data values must be remembered.

Suppose L is the set of all data words in which the first and last data values are equal. Suppose a device is reading the word $d_1d_2d_3d_1$ from left to right and trying to determine whether the word belongs to L (we are ignoring letters from the finite alphabet here). The device must remember d_1 when it is read first, so that it can be compared to the last data value. A machine independent characterization of what must be remembered is given in [4, Definition 2]; it says that the first occurrence of d_1 in $d_1d_2d_3d_1$ is *L-memorable* because

replacing it with some fresh data value d_4 (which doesn't occur in the word) makes a difference: $d_1d_2d_3d_1 \in L$ but $d_4d_2d_3d_1 \notin L$. We adapt this concept to transductions, by suitably modifying the definition of "making a difference".

▶ **Definition 4** (memorable values). *Suppose f is a transduction. A data value d is* f-memorable *in a data word u if there exists a data word v and a safe replacement d' for d in u such that $f(\underline{u[d/d']} \mid v) \neq f(\underline{u} \mid v)$.*

Let f be the transduction of Example 2 and d_1, d_2, d_3, d_1' be distinct data values. We have $f(\underline{d_1d_2d_3} \mid d_1) = (*, *, \texttt{left})(d_1, 4)$ and $f(\underline{d_1'd_2d_3} \mid d_1) = (d_1, 4)(*, *, \texttt{left})$. Hence, d_1 is f-memorable in $d_1d_2d_3$.

We have to consider one more phenomenon in transductions. Consider the transduction f whose output is ϵ for inputs of length less than five. For other inputs, the output is the third (resp. fourth) data value if the first and fifth are equal (resp. unequal). Let $d_1, d_2, d_3, d_4, d_5, d_1'$ be distinct data values. We have $f(\underline{d_1d_2d_3d_4} \mid v) = \epsilon = f(\underline{d_1'd_2d_3d_4} \mid v)$ if $v = \epsilon$ and $f(\underline{d_1d_2d_3d_4} \mid v) = (*, *, \texttt{left}) = f(\underline{d_1'd_2d_3d_4} \mid v)$ otherwise. Hence, d_1 is not f-memorable in $d_1d_2d_3d_4$. However, any device implementing f must remember d_1 after reading $d_1d_2d_3d_4$, so that it can be compared to the fifth data value. Replacing d_1 by d_1' does make a difference but we cannot detect it by comparing $f(\underline{d_1d_2d_3d_4} \mid v)$ and $f(\underline{d_1'd_2d_3d_4} \mid v)$. We can detect it as follows: $f(d_1d_2d_3d_4 \mid \underline{d_1}) = (d_3, 3) \neq (d_4, 4) = f(d_1d_2d_3d_4 \mid \underline{d_5})$. Changing the suffix from d_1 to d_5 influences how the prefix $d_1d_2d_3d_4$ is transformed (in transductions, prefixes are vulnerable to the influence of suffixes). The value d_1 is also contained in the prefix d_1d_2, but $f(d_1d_2 \mid \underline{v}) = f(d_1d_2 \mid \underline{v[d_1/d_5]})$ for all v. To detect that d_1d_2 is vulnerable, we first need to append d_3d_4 to d_1d_2 and then have a suffix in which we substitute d_1 with something else. We formalize this in the definition below; it can be related to the example above by setting $u = d_1d_2$, $u' = d_3d_4$ and $v = d_1$.

▶ **Definition 5** (vulnerable values). *A data value d is* f-vulnerable *in a data word u if there exist data words u', v and a data value d' such that d does not occur in u', d' is a safe replacement for d in $u \cdot u' \cdot v$ and $f(u \cdot u' \mid \underline{v[d/d']}) \neq f(u \cdot u' \mid \underline{v})$.*

Consider the transduction f defined as $f(u) = f_1(u) \cdot f_2(u)$; for $i \in [1, 2]$, f_i reverses its input if the i^{th} and last data values are distinct. On other inputs, f_i is the identity (f_1 is the transduction given in Example 2). In the two words $d_1d_2d_3d_1d_2d_3$ and $d_1d_2d_3d_2d_1d_3$, d_1 and d_2 are f-memorable. For every data word v, $f(\underline{d_1d_2d_3d_1d_2d_3} \mid v) = f(\underline{d_1d_2d_3d_2d_1d_3} \mid v)$, so it is tempting to say that the two words are equivalent. But after reading $d_1d_2d_3d_1d_2d_3$, a transducer would remember that d_2 is the latest f-memorable value it has seen. After reading $d_1d_2d_3d_2d_1d_3$, the transducer would remember that d_1 is the latest f-memorable value it has seen. Different f-memorable values play different roles and one way to distinguish which is which is to remember the order in which they occurred last. So we distinguish between $d_1d_2d_3d_1d_2d_3$ and $d_1d_2d_3d_2d_1d_3$. Suppose d_2, d_1 are two data values in some data word u. We say that d_1 is *fresher* than d_2 in u if the last occurrence of d_1 in u is to the right of the last occurrence of d_2 in u.

▶ **Definition 6.** *Suppose f is a transduction and u is a data word. We say that a data value d is f-influencing in u if it is either f-memorable or f-vulnerable in u. We denote by $\texttt{ifl}_f(u)$ the sequence $d_m \cdots d_1$, where $\{d_m, \ldots, d_1\}$ is the set of all f-influencing values in u and for all $i \in [1, m-1]$, d_i is fresher than d_{i+1} in u. We call d_i the i^{th} f-influencing data value in u. If a data value d is both f-vulnerable and f-memorable in u, we say that d is of* type `vm`*. If d is f-memorable but not f-vulnerable (resp. f-vulnerable but not f-memorable) in u, we say that d is of type* `m` *(resp.* `v`*). We denote by $\texttt{aifl}_f(u)$ the sequence $(d_m, t(d_m)) \cdots (d_1, t(d_1))$, where $t(d_i)$ is the type of d_i for all $i \in [1, m]$.*

To consider two data words u_1 and u_2 to be equivalent, we can insist that $\texttt{aifl}_f(u_1) = \texttt{aifl}_f(u_2)$. But as before, this may result in some infinite set of pairwise non-equivalent data words. We will relax the condition by saying that there must be a permutation π on data values such that $\texttt{aifl}_f(\pi(u_2)) = \texttt{aifl}_f(u_1)$. This is still not enough; we have overlooked one more thing that must be considered in such an equivalence. Recall that in transductions, prefixes are vulnerable to the influence of suffixes. So if u_1 is vulnerable to changing the suffix from v_1 to v_2, then $\pi(u_2)$ must also have the same vulnerability. This is covered by the third condition in the definition below.

▶ **Definition 7.** *For a transduction f, we define the relation $\equiv_f$ on data words as $u_1 \equiv_f u_2$ if there exists a permutation π on data values satisfying the following conditions:*

- $\lambda v.f_z(\pi(u_2) \mid v) = \lambda v.f(u_1 \mid v)$, *where* $z = |u_1| - |u_2|$,
- $\texttt{aifl}_f(\pi(u_2)) = \texttt{aifl}_f(u_1)$ *and*
- *for all* u, v_1, v_2, $f(u_1 \cdot u \mid v_1) = f(u_1 \cdot u \mid v_2)$ *iff* $f(\pi(u_2) \cdot u \mid v_1) = f(\pi(u_2) \cdot u \mid v_2)$.

As in the standard lambda calculus notation, $\lambda v.f_z(u \mid v)$ denotes the function that maps each input v to $f_z(u \mid v)$. It is routine to verify that for any data word u and permutation π, $\pi(u) \equiv_f u$, since π itself satisfies all the conditions above. We denote by $[u]_f$ the equivalence class of $\equiv_f$ containing u.

▶ **Lemma 8.** *If f is invariant under permutations, then $\equiv_f$ is an equivalence relation.*

Following is the main result of this extended abstract.

▶ **Theorem 9.** *A transduction f is implemented by a SSRT iff f satisfies the following properties: 1)f is invariant under permutations, 2) f is without data peeking, 3) f has linear blow up and 4) $\equiv_f$ has finite index.*

4 Constructing a SSRT from a Transduction

In this section, we prove the reverse direction of Theorem 9, by showing how to construct a SSRT that implements a transduction, if it satisfies the four conditions in the theorem. SSRTs read their input from left to right. Our first task is to get SSRTs to identify influencing data values as they are read one by one. Suppose a transducer that is intended to implement a transduction f has read a data word u and has stored in its registers the data values that are f-influencing in u. Suppose the transducer reads the next symbol (σ, e). To identify the data values that are f-influencing in $u \cdot (\sigma, e)$, will the transducer need to read the whole data word $u \cdot (\sigma, e)$ again? The answer turns out to be no, as the following result shows. The only data values that can possibly be f-influencing in $u \cdot (\sigma, e)$ are e and the data values that are f-influencing in u.

▶ **Lemma 10.** *Let f be a transduction, u be a data word, $\sigma \in \Sigma$ and d, e be distinct data values. If d is not f-memorable (resp. f-vulnerable) in u, then d is not f-memorable (resp. f-vulnerable) in $u \cdot (\sigma, e)$.*

Next, suppose that d is f-influencing in u. How will we get the transducer to detect whether d continues to be f-influencing in $u \cdot (\sigma, e)$? The following result provides a partial answer. If $u_1 \equiv_f u_2$ and the i^{th} f-influencing value in u_1 continues to be f-influencing in $u_1 \cdot (\sigma, e)$, then the i^{th} f-influencing value in u_2 continues to be f-influencing in $u_2 \cdot (\sigma, e)$. The following result combines many such similar results into a single one.

▶ **Lemma 11.** *Suppose f is a transduction that is invariant under permutations and without data peeking. Suppose u_1, u_2 are data words such that $u_1 \equiv_f u_2$, $\texttt{ifl}_f(u_1) = d_1^m d_1^{m-1} \cdots d_1^1$*

and $\texttt{ifl}_f(u_2) = d_2^m d_2^{m-1} \cdots d_2^1$. *Suppose* $d_1^0 \in D$ *is not* f*-influencing in* u_1, $d_2^0 \in D$ *is not* f*-influencing in* u_2 *and* $\sigma \in \Sigma$. *For all* $i, j \in [0, m]$, *the following are true:*

1. d_1^i *is* f*-memorable (resp.* f*-vulnerable) in* $u_1 \cdot (\sigma, d_1^j)$ *iff* d_2^i *is* f*-memorable (resp.* f*-vulnerable) in* $u_2 \cdot (\sigma, d_2^j)$.
2. $u_1 \cdot (\sigma, d_1^j) \equiv_f u_2 \cdot (\sigma, d_2^j)$.

If $u_1 \equiv_f u_2$, there exists a permutation π such that $\texttt{aifl}_f(u_1) = \texttt{aifl}_f(\pi(u_2))$. Hence, all data words in the same equivalence class of $\equiv_f$ have the same number of f-influencing values. If $\equiv_f$ has finite index, then there is a bound (say I) such that any data word has at most I f-influencing data values. Consider a SSRT $S_f^{\texttt{ifl}}$ with the set of registers $R = \{r_1, \ldots, r_I\}$. The states are of the form $([u]_f, ptr)$, where u is some data word and $ptr : [1, |\texttt{ifl}_f(u)|] \to R$ is a pointer function. Let $ptr_\perp$ be the trivial function from $\emptyset$ to R. The transitions can be designed to satisfy the following.

▶ **Lemma 12.** *Suppose the SSRT* $S_f^{\texttt{ifl}}$ *starts in the configuration* $(([\epsilon]_f, ptr_\perp), val_\epsilon, 0)$ *and reads some data word* u. *It reaches the configuration* $(([u]_f, ptr), val, |u|)$ *such that* $val(ptr(i))$ *is the* i^{th} f*-influencing value in* u *for all* $i \in [1, |\texttt{ifl}_f(u)|]$.

The details of constructing $S_f^{\texttt{ifl}}$ can be found in the full version. In short, the idea is that we can hard code rules such as "if the data value just read is the i^{th} f-influencing value in u, it continues to be f-influencing in the new data word". Lemma 11 implies that the validity of such rules depend only on the equivalence class $[u]_f$ containing u and does not depend on u itself. So the SSRT need not remember the entire word u; it just remembers the equivalence class $[u]_f$ in its control state. The SSRT can check whether the new data value is the i^{th} f-influencing value in u, by comparing it with the register $ptr(i)$.

Next we will extend the transducer to compute the output of a transduction. Suppose the transducer has read the data word u so far. The transducer doesn't know what is the suffix that is going to come, so whatever computation it does has to cover all possibilities. The idea is to compute $\{f(u \mid \underline{v}) \mid v \in (\Sigma \times D)^*\}$ and store them in data word variables, so that when it has to output $f(u)$ at the end, it can output $f(u \mid \underline{\epsilon})$. However, this set can be infinite. If $\equiv_f$ has finite index, we can reduce it to a finite set. Recall the transduction f from Example 2 and the infinite set of data words $\{(a, d_i) \mid i \geq 1\}$. For any $i \neq j$, $f(\underline{(a, d_i)} \mid (a, d_i)) \neq f(\underline{(a, d_j)} \mid (a, d_i))$ for $i \neq j$. But for every i, there is a permutation π_i on data values mapping d_i to d_1 so that $f(\underline{\pi_i(a, d_i)} \mid v) = f(\underline{(a, d_1)} \mid v)$ for any data word v. We have revealed that all data words in $\{(a, d_i) \mid i \geq 1\}$ are equivalent by applying a permutation to each data word, so that they all have the same f-influencing data values. We formalize this idea below.

▶ **Definition 13.** *Let* f *be a transduction and* Π *be the set of all permutations on the set of data values* D. *An* equalizing scheme *for* f *is a function* $E : (\Sigma \times D)^* \to \Pi$ *such that there exists a sequence* $\delta_1 \delta_2 \cdots$ *of data values satisfying the following condition: for every data word* u *and every* $i \in [1, |\texttt{ifl}_f(u)|]$, *the* i^{th} f*-influencing data value of* $E(u)(u)$ *is* δ_i.

Note that $E(u)(u)$ denotes the application of the permutation $E(u)$ to the data word u. We will write $E(u)(u)$ as u_q for short (intended to be read as "equalized u"). Note that $E(u)^{-1}(u_q) = u$. Left parts that have been equalized like this will not have arbitrary influencing data values – they will be from the sequence $\delta_1 \delta_2 \cdots$. For the transduction in Example 2, the first data value is the only influencing value in any data word. An equalizing scheme will map the first data value of all data words to δ_1.

The relation $\equiv_f$ identifies two prefixes when they behave similarly. We now define a relation that serves a similar propose, but for suffixes.

▶ **Definition 14.** *For a transduction f and equalizing scheme E, we define the relation $\equiv_f^E$ on data words as $v_1 \equiv_f^E v_2$ if for every data word u, $f(u_q \mid \underline{v_1}) = f(u_q \mid \underline{v_2})$.*

It is routine to verify that $\equiv_f^E$ is an equivalence relation. Saying that v_1 and v_2 are similar suffixes if $f(u \mid \underline{v_1}) = f(u \mid \underline{v_2})$ for all u doesn't work; this may result in infinitely many pairwise unequivalent suffixes (just like $\equiv_f$ may have infinite index if we don't apply permutations to prefixes). So we "equalize" the prefixes so that they have the same f-influencing data values, before checking how suffixes influence them.

▶ **Lemma 15.** *Suppose f is a transduction satisfying all the conditions of Theorem 9. If E is an equalizing scheme for f, then $\equiv_f^E$ has finite index.*

Suppose we are trying to design a SSRT to implement a transduction f, which has the property that $\equiv_f^E$ has finite index. The SSRT can compute the set $\{f(u_q \mid \underline{v}) \mid v \in (\Sigma \times D)^*\}$, which is finite (it is enough to consider one representative v from every equivalence class of $\equiv_f^E$). At the end when the SSRT has to output $f(u)$, it can output $E(u)^{-1}(f(u_q \mid \underline{\epsilon})) = f(u)$. The SSRT never knows what is the next suffix; at any point of time, the next suffix could be ϵ. So the SSRT has to apply the permutation $E(u)^{-1}$ at each step. Letting V be a finite set of representatives from every equivalence class of $\equiv_f^E$, the SSRT computes $\{f(u \mid \underline{E(u)^{-1}(v)}) \mid v \in V\}$ at every step.

Now suppose the SSRT has computed $\{f(u \mid \underline{E(u)^{-1}(v)}) \mid v \in V\}$, stored them in data word variables and it reads the next symbol (σ, d). The SSRT has to compute $\{f(u \cdot (\sigma, d) \mid \underline{E(u \cdot (\sigma, d))^{-1}(v)}) \mid v \in V\}$ from whatever it had computed for u.

To explain how the above computation is done, we use some terminology. In factored outputs of the form $f(u \mid \underline{v})$, $f(\underline{u} \mid \underline{v})$, $f(\underline{u} \mid v \mid \underline{w})$ or $f(\underline{u} \mid \underline{v} \mid \underline{w})$, a triple is said to come from u if it has origin in u or it is the triple $(*, *, \texttt{left})$. A left block in such a factored output is a maximal infix of triples, all coming from the left part u. Similarly, a non-right block is a maximal infix of triples, none coming from the right part. Middle blocks are defined similarly. For the transduction f in Example 2, $f((a, d_1)(b, d_2)(c, d_3))$ is $(c, d_3, 3)(b, d_2, 2)(a, d_1, 1)$. In $f((a, d_1)(b, d_2) \mid \underline{(c, d_3)})$, $(b, d_2, 2)(a, d_1, 1)$ is a left block. In $f(\underline{(a, d_1)} \mid (b, d_2) \mid \underline{(c, d_3)})$, $(b, d_2, 2)$ is a middle block. In $f(\underline{(a, d_1)} \mid \underline{(b, d_2)} \mid \underline{(c, d_3)})$, $(*, *, \texttt{middle})(*, *, \texttt{left})$ is a non-right block, consisting of one middle and one left block.

The concretization of the i^{th} left block (resp. middle block) in $f(\underline{u} \mid \underline{v} \mid \underline{w})$ is defined to be the i^{th} left block in $f(u \mid \underline{vw})$ (resp. the i^{th} middle block in $f(\underline{u} \mid v \mid \underline{w})$). The concretization of the i^{th} non-right block in $f(\underline{u} \mid \underline{v} \mid \underline{w})$ is obtained by concatenating the concretizations of the left and middle blocks that occur in the i^{th} non-right block. The following is a direct consequence of the definitions.

▶ **Proposition 16.** *The i^{th} left block of $f(u \cdot (\sigma, d) \mid \underline{v})$ is the concretization of the i^{th} non-right block of $f(\underline{u} \mid \underline{(\sigma, d)} \mid \underline{v})$.*

For the transduction f from Example 2, the first left block of $f((a, d_1)(b, d_2) \mid \underline{(c, d_3)})$ is $(b, d_2, 2)(a, d_1, 1)$, which is the concretization of $(*, *, \texttt{middle})(*, *, \texttt{left})$, the first non-right block of $f(\underline{(a, d_1)} \mid \underline{(b, d_2)} \mid \underline{(c, d_3)})$.

From Proposition 16, we deduce that the i^{th} left block of $f(u \cdot (\sigma, d) \mid \underline{E(u \cdot (\sigma, d))^{-1}(v)})$ is the concretization of the i^{th} non-right block of $f(\underline{u} \mid \underline{(\sigma, d)} \mid \underline{E(u \cdot (\sigma, d))^{-1}(v)})$. The concretizations come from the left blocks of $f(u \mid \underline{(\sigma, d) \cdot E(u \cdot (\sigma, d))^{-1}(v)})$ and the middle blocks of $f(\underline{u} \mid (\sigma, d) \mid \underline{E(u \cdot (\sigma, d))^{-1}(v)})$. In the absence of data values, the above two statements would be as follows: The i^{th} left block of $f(u \cdot \sigma \mid \underline{v})$ is the concretization of the i^{th} non-right block of $f(\underline{u} \mid \underline{\sigma} \mid \underline{v})$. The concretizations come from the left blocks of $f(u \mid \underline{\sigma \cdot v})$ and the middle blocks of $f(\underline{u} \mid \sigma \mid \underline{v})$. This technique of incrementally computing factored

outputs was introduced in [5] for SSTs. In SSTs, $f(u \mid \underline{\sigma \cdot v})$ would have been computed as $f(u \mid \underline{v'})$ when u was read, where v' is some word that influences prefixes in the same way as $\sigma \cdot v$. But in SSRTs, only $f(u \mid \underline{E(u)^{-1}(v')})$ would have been computed for various v'; what we need is $f(u \mid \underline{(\sigma,d) \cdot E(u \cdot (\sigma,d))^{-1}(v)})$. We work around this by proving that a v' can be computed such that $f(u \mid \underline{(\sigma,d) \cdot E(u \cdot (\sigma,d))^{-1}(v)}) = f(u \mid \underline{E(u)^{-1}(v')})$. This needs some technical work, which can be found in the full version. The end result is summarized below.

▶ **Lemma 17.** *Suppose f is a transduction satisfying all the conditions in Theorem 9, E is an equalizing scheme for f, u, v are data words and $(\sigma, d) \in \Sigma \times D$. There are functions g_1 and g_2 such that $f(u \cdot (\sigma,d) \mid \underline{E(u \cdot (\sigma,d))^{-1}(v)}) = g_1([u]_f, \mathtt{ifl}_f(u), d, v, f(u \mid \underline{E(u)^{-1}(v')}))$, where $v' = g_2([u]_f, \mathtt{ifl}_f(u), d, v)$.*

The functions g_1 and g_2 need to be applied by the SSRT and that is possible. For g_2, it only needs $[u]_f$ (stored in the control state), $\mathtt{ifl}_f(u)$ (stored in the registers), d (this is the latest data value that has been read) and v (which is from a finite set and can be hardcoded). For g_1, it additionally needs $f(u \mid \underline{E(u)^{-1}(v')})$, which would have been be stored in one of the data word variables when u was read.

Suppose $v_1, v_2 \in V$ and $v' = g_2([u]_f, \mathtt{ifl}_f(u), d, v_1) = g_2([u]_f, \mathtt{ifl}_f(u), d, v_2)$. We have $f(u \cdot (\sigma,d) \mid \underline{E(u \cdot (\sigma,d))^{-1}(v_1)}) = g_1([u]_f, \mathtt{ifl}_f(u), d, v_1, f(u \mid \underline{E(u)^{-1}(v')}))$ and $f(u \cdot (\sigma,d) \mid \underline{E(u \cdot (\sigma,d))^{-1}(v_2)})$ is equal to $g_1([u]_f, \mathtt{ifl}_f(u), d, v_2, f(u \mid \underline{E(u)^{-1}(v')}))$. The SSRT would have stored $f(u \mid \underline{E(u)^{-1}(v')})$ in a data word variable and now it is needed for two computations. But in SSRTs, the contents of one data word variable cannot be used in two computations, since SSRTs are copyless. This problem is solved in [5] for SSTs using a two way transducer model equivalent to SSTs. In this two way model, the suffix can be read and there is no need to perform computations for multiple suffixes. We cannot use that technique here, since there are no known two way models equivalent to SSRTs.

We solve this problem by not performing the two computations of g_1 immediately. Instead, we remember the fact that there is a multiple dependency on a single data word variable. The actual computation is delayed until the SSRT reads more symbols from the input and gathers enough information about the suffix to discard all but one of the dependencies. Suppose we have delayed computing $f(u \cdot (\sigma,d) \mid \underline{E(u \cdot (\sigma,d))^{-1}(v_1)})$ due to some dependency. After reading the next symbol, $f(u \cdot (\sigma,d) \mid \underline{E(u \cdot (\sigma,d))^{-1}(v_1)})$ itself might be needed for multiple computations. We keep track of such nested dependencies in a tree data structure called dependency tree. Dependency trees can grow unboundedly, but if $\equiv_f^E$ has finite index, it can be shown that some parts can be discarded from time to time to keep their size bounded. We store such reduced dependency trees as part of the control states of the SSRT. The details of this construction can be found in the full version and the end result is summarized below.

Proof sketch of reverse direction of Theorem 9. Let f be a transduction that satisfies all the properties stated in Theorem 9. We extend the SSRT $S_f^{\mathtt{ifl}}$. The states will be of the form $([u]_f, ptr, T)$ where $[u]_f$ and ptr are as before and T is a reduced dependency tree. The SSRT will have a finite set of data word variables X. After reading any data word u, the SSRT will reach the configuration $(([u]_f, ptr, T), val, |u|)$ that satisfies the following property. For any equivalence class $[v]_f^E$ of $\equiv_f^E$, there is a leaf node θ of T such that the path from the root of T to θ will determine a sequence z in X^* (z is a sequence of data word variables) and $val(z) = f(u \mid \underline{E(u)^{-1}(v)})$ ($val(z)$ is the concatenation of the contents of data word variables according to the sequence z). After reading the entire input u, the SSRT outputs $f(u) = f(u \mid \underline{E(u)^{-1}(\epsilon)})$ using the leaf of T corresponding to $[\epsilon]_f^E$. ◀

5 Properties of Transductions Implemented by SSRTs

In this section, we prove the forward direction of our main result (Theorem 9). We begin by identifying data words after reading which, a SSRT reaches similar configurations

▶ **Definition 18.** *For a SSRT S, we define a binary relation $\equiv_S$ on data words as follows: $u_1 \equiv_S u_2$ if they satisfy the following conditions. Suppose f is the transduction implemented by S, which reaches the configuration $(q_1, val_1, |u_1|)$ after reading u_1 and reaches $(q_2, val_2, |u_2|)$ after reading u_2.*

1. *$q_1 = q_2$,*
2. *for any two registers r_1, r_2, we have $val_1(r_1) = val_1(r_2)$ iff $val_2(r_1) = val_2(r_2)$,*
3. *for any register r, $val_1(r)$ is the i^{th} f-memorable value (resp. f-vulnerable value) for some i in u_1 iff $val_2(r)$ is the i^{th} f-memorable value (resp. f-vulnerable value) in u_2,*
4. *for any data word variable x, we have $val_1(x) = \epsilon$ iff $val_2(x) = \epsilon$ and*
5. *for any two subsets $X_1, X_2 \subseteq X$ and any arrangements χ_1, χ_2 of X_1, X_2 respectively, $val_1(\chi_1) = val_1(\chi_2)$ iff $val_2(\chi_1) = val_2(\chi_2)$.*

An arrangement of a finite set X_1 is a sequence in X_1^* in which every element of X_1 occurs exactly once. It is routine to verify that $\equiv_S$ is an equivalence relation of finite index.

Suppose a SSRT S reads a data word u, reaches the configuration $(q, val, |u|)$ and from there, continues to read a data word v. For some data word variable $x \in X$, if $val(x)$ is some data word z, then none of the transitions executed while reading v will split z – it might be appended or prepended with other data words and may be moved to other variables but never split. Suppose $X = \{x_1, \ldots, x_m\}$. The transitions executed while reading v can arrange $val(x_1), \ldots, val(x_m)$ in various ways, possibly inserting other data words (whose origin is in v, so they will be replaced by $(*, *, \texttt{right})$ in $[\![S]\!](u \mid \underline{v})$) in between. Hence, any left block of $[\![S]\!](u \mid \underline{v})$ is $val(\chi)$, where χ is some arrangement of some subset $X' \subseteq X$. Recall that a left block of $[\![S]\!](u \mid \underline{v})$ is a maximal infix that doesn't contain $(*, *, \texttt{right})$).

Proof sketch of forward direction of Theorem 9. Suppose a SSRT S implements a transduction f. It can be shown that $\equiv_S$ refines $\equiv_f$, so $\equiv_f$ has finite index. The most difficult part of this proof is to prove that if $u_1 \equiv_S u_2$, then there exists a permutation π such that for all data words u, v_1, v_2, $f(u_1 \cdot u \mid \underline{v_1}) = f(u_1 \cdot u \mid \underline{v_2})$ iff $f(\pi(u_2) \cdot u \mid \underline{v_1}) = f(\pi(u_2) \cdot u \mid \underline{v_2})$. The idea is to show that if $f(u_1 \cdot u \mid \underline{v_1}) \neq f(u_1 \cdot u \mid \underline{v_2})$, then for some arrangements χ_1, χ_2 of some subsets $X_1, X_2 \subseteq X$, $val_1(\chi_1) \neq val_1(\chi_2)$ (val_1 (resp. val_2) is the valuation reached by S after reading u_1 (resp. u_2)). Since $u_1 \equiv_S u_2$, this implies that $val_2(\chi_1) \neq val_2(\chi_2)$, which in turn implies that $f(\pi(u_2) \cdot u \mid \underline{v_1}) \neq f(\pi(u_2) \cdot u \mid \underline{v_2})$. ◀

6 Future Work

One direction to explore is whether there is a notion of minimal canonical SSRT and if a given SSRT can be reduced to an equivalent minimal one. Adding a linear order on the data domain, logical characterization of SSRTs and studying two way transducer models with data are some more interesting studies.

Using nominal automata, techniques for finite alphabets can often be elegantly carried over to infinite alphabets, as done in [19], for example. It would be interesting to see if the same can be done for streaming transducers over infinite alphabets. Using concepts from the theory of nominal automata, recent work [6] has shown that an atom extension of streaming string transducers is equivalent to a certain class of two way transducers. This model of

transducers is a restriction of SSRTs and is robust like regular languages over finite alphabets. It would also be interesting to see how can techniques in this extended abstract be simplified to work on the transducer model presented in [6].

References

1. R. Alur and P. Černý. Expressiveness of streaming string transducers. In *FSTTCS 2010*, volume 8 of *LIPIcs*, pages 1–12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPIcs.FSTTCS.2010.1.
2. R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL 2011*, POPL, pages 1–12. ACM, 2011.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
4. M. Benedikt, C. Ley, and G. Puppis. What you must remember when processing data words. In *Proceedings of the 4th Alberto Mendelzon International Workshop on Foundations of Data Management, Argentina*, volume 619 of *CEUR Workshop Proceedings*, 2010.
5. M. Bojańczyk. Transducers with origin information. In *ICALP*, volume 8573 of *LNCS*, pages 26–37, Berlin, Heidelberg, 2014. Springer.
6. M. Bojańczyk and R. Stefański. Single-Use Automata and Transducers for Infinite Alphabets. In *ICALP 2020*, volume 168 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113:1–113:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2020.113.
7. Sougata Bose, Anca Muscholl, Vincent Penelle, and Gabriele Puppis. Origin-equivalence of two-way word transducers is in PSPACE. In Sumit Ganguly and Paritosh K. Pandya, editors, *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, volume 122 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FSTTCS.2018.22.
8. S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. *Journal of Logical and Algebraic Methods in Programming*, 84(1):54–66, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.
9. S. Cassel, B. Jonsson, F. Howar, and B. Steffen. A succinct canonical register automaton model for data domains with binary relations. In *Automated Technology for Verification and Analysis - 10th International Symposium, 2012, Proceedings*, pages 57–71, 2012. doi:10.1007/978-3-642-33386-6_6.
10. Y-F Chen, O. Lengál, T. Tan, and Z. Wu. Register automata with linear arithmetic. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.
11. M. Chytil and V. Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *Automata, Languages and Programming, Fourth Colloquium, University of Turku, Finland, July 18-22, 1977, Proceedings*, pages 135–147. Springer Berlin Heidelberg, 1977.
12. A. Durand-Gasselin and P. Habermehl. Regular transformations of data words through origin information. In B. Jacobs and C. Löding, editors, *Foundations of Software Science and Computation Structures*, pages 285–300, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
13. Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. Synthesis of data word transducers. In Wan J. Fokkink and Rob van Glabbeek, editors, *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*, volume 140 of *LIPIcs*, pages 24:1–24:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CONCUR.2019.24.

14 Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. On computability of data word functions defined by transducers. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 217–236. Springer, 2020. doi:10.1007/978-3-030-45231-5_12.

15 N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theoretical Computer Science*, 306:155–175, 2003.

16 F. Howar, M. Isberner, B. Steffen, O. Bauer, and B. Jonsson. Inferring semantic interfaces of data structures. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 554–571, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

17 F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 251–266, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

18 M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

19 J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 613–625, 2017.