Expressiveness of streaming string transducers

Rajeev Alur[1] and Pavol Černý[2]

1 University of Pennsylvania
2 IST Austria

Abstract

Streaming string transducers [1] define (partial) functions from input strings to output strings. A streaming string transducer makes a single pass through the input string and uses a finite set of variables that range over strings from the output alphabet. At every step, the transducer processes an input symbol, and updates all the variables in parallel using assignments whose right-hand-sides are concatenations of output symbols and variables with the restriction that a variable can be used at most once in a right-hand-side expression. It has been shown that streaming string transducers operating on strings over infinite data domains are of interest in algorithmic verification of list-processing programs, as they lead to PSPACE decision procedures for checking pre/post conditions and for checking semantic equivalence, for a well-defined class of heap-manipulating programs. In order to understand the theoretical expressiveness of streaming transducers, we focus on streaming transducers processing strings over finite alphabets, given the existence of a robust and well-studied class of "regular" transductions for this case. Such regular transductions can be defined either by two-way deterministic finite-state transducers, or using a logical MSO-based characterization. Our main result is that the expressiveness of streaming string transducers coincides exactly with this class of regular transductions.

1 Introduction

Deterministic finite-state automata are a canonical model for finite-state *acceptors* of strings, since many variations turn out to be equally expressive and the resulting class of regular languages enjoys a number of desirable theoretical properties. In this paper, we focus on *transducer* models that define (partial) functions from input strings to output strings. The most natural model for a finite-state transducer is a finite-state machine that, at each step, reads an input symbol and produces zero or more output symbols. If we restrict such a machine to read the input string only once from left to right, then the model is too restrictive: while "delete all a symbols" can be implemented, "delete all a symbols, if the input string contains a b symbol" cannot be implemented. However, the *two-way deterministic finite-state transducers* have appealing theoretical properties: the equivalence problem is decidable, they are expressively equivalent to MSO (monadic second-order logic) definable transductions, and this class of "regular" transductions is closed under operations such as sequential composition [2, 4, 3].

Recently, we proposed the model of *streaming string transducers* for algorithmic verification of single-pass list processing programs [1]. A streaming string transducer makes a single pass through the input string to produce an output string. It uses a finite set of variables that range over strings from the output alphabet. At every step, the transducer processes an input symbol, and updates all the variables in parallel using assignments whose right-hand-sides are concatenations of output symbols and variables with the restriction that a variable can be used at most once in a right-hand-side expression. For example, with two variables x and y, the update $(x, y) = (x.y, a)$ sets x to the concatenation of x and y, and sets y to the constant a. While such an update is permitted, the update $(x, y) = (x.y, y)$ is not, since y appears twice in the right-hand-sides, and would amount to "copying". The output in a given state is specified as a concatenation of output symbols and variables with a similar

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

no-copy restriction. Unlike classical tape-based models, the streaming string transducer is not constrained to add output symbols only at the end, and can compute the output in multiple chunks that can be extended and concatenated as needed.

Streaming string transducers have been shown to be useful for algorithmic verification of list processing programs [1]. The problems of checking functional equivalence of two streaming transducers, and of checking whether a streaming transducer satisfies pre/post verification conditions specified by streaming acceptors over input/output strings, are decidable with PSPACE complexity. There is an expressively equivalent class of imperative programs that manipulate heap-allocated single-linked list data structure, as well as a corresponding class of list-processing functional programs with syntactic restrictions on recursive calls. These results lead to algorithms for checking functional equivalence of two programs, written possibly in different programming styles, for commonly used routines such as insert, delete, and reverse.

A goal of this paper is to study the expressiveness of streaming string transducers in order to gain better theoretical insights into their computing power. The transducers in [1] process strings over a potentially infinite data domain that supports the operations of equality and ordering. Given that the notion of regular transductions is well-understood in the context of strings over finite alphabets, we restrict our attention to streaming transducers that process strings over finite alphabets. The main result of the paper is that streaming string transducers exactly capture regular transductions, and thus, are equivalent to two-way transducers as well as MSO-definable string transductions.

In order to develop our results, we consider another single-pass transducer model called *heap-based string transducer* that reads the input string from left to right in a single pass, and computes the output using a heap of cells each of which can store an output symbol and has a next pointer. The next-pointers induce an (unranked) forest structure over cells. The transducer accesses the heap using a finite number of pointer variables, and can change next-pointers of cells referenced by these variables. It can also add new cells to the heap. The sequence of symbols labeling the cells accessible from a state-dependent output-pointer is the output of the transducer. For example, to output the *reverse* of the input string, the heap-based transducer, at each step, reads the next input symbol, and adds a cell containing this input symbol to the front of the output list being computed, exactly the same way as a C program would reverse a linked-list in a single pass. While proving assertions of programs manipulating heaps is typically undecidable [6], the key restriction for our model of a heap-based transducer is that it can update, but not traverse, the next-pointers of the cells referenced by its pointer variables (that is, an assignment of the form $next(x) = y$ is allowed, but $x = next(y)$ is not). Heap-based transducers can be viewed as syntactically restricted and abstract version of imperative single-pass programs studied in [1].

We first show that given a two-way transducer, one can construct an equivalent one-way heap-based transducer. The proof builds on the classical simulation of a two-way acceptor by a one-way acceptor [7], but needs new insights in order to maintain the potentially needed output segments in a shared heap that can be modified using a bounded number of updates at each step. The fact that a streaming string transducer can simulate a heap-based transducer is a corollary of a result of [1] that shows that single-pass list processing programs can be simulated by streaming transducers. Finally, we establish that a streaming string transducer can be captured by an MSO-definable string transformation, thereby establishing equivalence of all the transducer models.

We also show that streaming string transducers are closed under sequential composition. This result follows from the MSO characterization, but we give a direct proof using summaries of a computation of a streaming string transducer. Finally, we show that extending heap-based

transducers by allowing the traversal instruction that updates a pointer to the next-pointer of a referenced cell, leads to a strictly more expressive model than the class of regular transductions.

2 Regular String Transductions

A *deterministic transduction* from an input alphabet Σ to an output alphabet Γ is a partial function from Σ^* to Γ^*. We briefly review two equivalent ways of defining deterministic transductions using two-way finite-state transducers and using monadic second-order logic (MSO), and some known results (the details can be found in [3]). We will use the following three transductions throughout the paper. Let $\Sigma = \Gamma = \{a, b\}$. The transduction f_1 rewrites an input string to the string followed by its reverse: $f_1(w) = w.rev(w)$. For the transduction f_2, if the input string ends with the letter b, then f_2 deletes all occurrences of a, otherwise it leaves the input unchanged: if $w \in \Sigma^* b$ then $f_2(w) = b^k$ where k is the number of b's in w, else $f_2(w) = w$. The transduction f_3 replaces each symbol b by as many b's as there are a's between this occurrence of b and the previous occurrence of b: $f_3(a^{i_1} b a^{i_2} b \cdots a^{i_k} b a^{i_{k+1}}) = a^{i_1} b^{i_1} a^{i_2} b^{i_2} \cdots a^{i_k} b^{i_k} a^{i_{k+1}}$.

A *two-way deterministic (finite-state) transducer* M from input alphabet Σ to output alphabet Γ consists of a finite set of states Q, an initial state $q_0 \in Q$, a final state $q_f \in Q$, and a transition function δ from $Q \times (\Sigma \cup \{\vdash, \dashv\})$ to $Q \times \{-1, 0, +1\} \times \Gamma^*$. The symbols $\vdash$ and $\dashv$ are used to mark the two ends of the input. Given an input string $w \in \Sigma^*$, the transducer M starts in state q_0 with the input tape containing the string $\vdash w \dashv$, scanning the left-most symbol. At every step, based on the current state q and the current input symbol a, the machine updates the state (as specified by the first component of $\delta(q, a)$), moves the read-head (as specified by the second component of $\delta(q, a)$, where -1 means move left, 0 means stay put, and $+1$ means move right), and outputs a sequence of symbols in Γ (as specified by the third component of $\delta(q, a)$). If the current state is the final state q_f, the machine stops, and in this case, the output $[\![M]\!](w)$ corresponding to the input string w is the concatenation of outputs emitted along the run. If the machine never enters the final state, or tries to move left while reading $\vdash$, or tries to move right while reading $\dashv$, then this is an error, and $[\![M]\!](w)$ is undefined. This partial function $[\![M]\!]$ defines the semantics of the machine M, and is a deterministic transduction from Σ to Γ.

To illustrate the definition of a two-way transducer, let us consider how to implement the example transductions by two-way transducers. To implement the transduction f_1, the two-way transducer reads the string in one left-to-right pass followed by one right-to-left pass, emitting the symbol read from the input at every step. The two-way transducer for f_2, first moves all the way to the right. If the last symbol is b, then while moving right to left it outputs b for every b symbol it reads, while ignoring a symbols. If the last symbol is not b, it moves all the way to the left, and using a final left-to-right pass, it outputs every symbol it reads. The transducer for f_3 starts moving to the right emitting an a symbol for each a symbol read. If it encounters the right end-marker, it halts, If it encounters a b symbol, it emits the empty string and starts moving left. For every a symbol it reads, it emits b, and keeps moving left. When it encounters a b symbol or the left end-marker, it starts moving right again skipping over the a's until it encounters a b, emitting the empty string at each step. Then the whole cycle repeats.

For defining transductions using monadic second-order logic, a string $w = w_1 w_2 \ldots w_k$ is viewed as a (string) graph G_w with $k+1$ vertices $v_0 v_1 \ldots v_k$, with an edge from each v_i to v_{i+1} labeled with the symbol w_i. Then, an MSO formula over an alphabet Σ, to be

interpreted over such a graph G_w, consists of Boolean connectives, quantifiers, first-order variables that range over vertices of G_w, monadic second-order variables that range over sets of vertices of G_w, and atomic formulas of the form $a(x,y)$, for $a \in \Sigma$, meaning that the vertex x has an a-labeled edge to the vertex y. A *deterministic MSO transducer* T from input alphabet Σ to output alphabet Γ consists of a finite copy set C, vertex formulas $\varphi^c(x)$, for each $c \in C$, each of which is an MSO formula over Σ with one free first-order variable x, and edge formulas $\varphi_a^{c,d}(x,y)$, for each $a \in \Gamma$ and $c,d \in C$, each of which is an MSO formula over Σ with two free first-order variables x and y. Given an input string w, consider the following output graph: for each vertex x in G_w and $c \in C$, there is a vertex x^c in the output if the formula $\varphi^c(x)$ holds over G_w, and for all such vertices x^c and y^d, there is an a-labeled edge from x^c to y^d if the formula $\varphi_a^{c,d}(x,y)$ holds over G_w. If this graph is the string graph corresponding to the string u over Γ then $[\![T]\!](w) = u$, and if this graph is not a string graph, then $[\![T]\!](w)$ is undefined.

Let us revisit our example transductions. To define the transduction f_1, we choose the copy set $C = \{1,2\}$. The output graph retains both copies of each vertex in input graph, except the last one. When $a(x,y)$ holds in the input graph $a(x^1,y^1)$ and $a(y^1,x^1)$ holds in the output graph (the last vertex needs to be handled specially to connect the two copies). For the transduction f_2, we need only one copy of each vertex. The formula φ^1 is defined so that $\varphi^1(x)$ holds precisely when either the position x corresponds to a b-symbol in the input graph, or if the last symbol in the input is b (such "regular" look-ahead is easily definable using MSO formulas). The output graph has an edge from x to y if $\varphi^1(x)$ holds, y is the least position following x for which $\varphi^1(y)$ holds, and the label is the same as the input symbol corresponding to position x. The transduction f_3 can be defined by a deterministic MSO transducer with the copy set $C = \{1,2\}$, where both output copies of a vertex are retained, or omitted, depending on whether the corresponding symbol in the input string is a, or b, respectively. The edge formulas are slightly complicated, but can be defined in MSO [3].

The two frameworks are equally expressive [3], and this class of transductions is called *regular string transductions*. The class of regular transductions is closed under sequential composition [2], and it is decidable to check whether two such transductions, presented by, say, two two-way deterministic transducers, are equivalent [4]. Observe that for any regular transduction, the ratio of the length of the output string to the length of the input string is bounded by a constant, namely, the size of the copy set C of the corresponding deterministic MSO transducer.

3 Streaming Transducer Model

A *(deterministic) streaming string transducer* W from input alphabet Σ to output alphabet Γ consists of a finite set of states Q, an initial state $q_0 \in Q$, a finite set of variables X, a partial output function F from Q to $(\Gamma \cup X)^*$ such that for each $q \in Q$ and $x \in X$, there is at most one occurrence of x in $F(q)$, a state-transition function δ_1 from $Q \times \Sigma$ to Q, and a variable-update function δ_2 from $Q \times \Sigma \times X$ to $(\Gamma \cup X)^*$ such that for each $q \in Q$ and $a \in \Sigma$ and $x \in X$, there is at most one occurrence of x in the set of strings $\{\delta_2(q,a,y) \mid y \in X\}$.

To define the semantics of such a transducer, consider configurations of the form (q,s), where s is a valuation from X to Γ^*. A valuation from X to Γ^* is extended to a valuation from $(X \cup \Gamma)^*$ to Γ^* in the natural way. The initial configuration is (q_0, s_0) where s_0 maps each variable to the empty string. The transition function is defined by $\delta((q,s),a) = (\delta_1(q,a), s')$ where for each variable x, $s'(x) = s(\delta_2(q,a,x))$. For an input string $w \in \Sigma^*$, if $\delta^*((q_0,s_0),w) = (q,s)$, then if $F(q)$ is undefined then so is $[\![W]\!](w)$, otherwise $[\![W]\!](w) = s(F(q))$.

The transduction f_1 can be implemented by a streaming string transducer W_{rev} with a single state and two variables x and y. Each symbol a is processed by the update $(x, y) = (xa, ay)$, and the output function is xy.

The transduction f_2 can be implemented by a streaming string transducer with two states q_0 and q_1, and two variables x and y. Initially the state is q_0, and the transducer is in state q_1 precisely when the most recent input symbol is b (it goes to state q_1 on reading b and to state q_0 on reading a). At every step, x contains the input string read so far, and y contains only b's encountered so far (the variable-update function on reading a is $(x, y) = (xa, y)$, and on reading b is $(x, y) = (xb, yb)$). In state q_1 the output function returns y, in state q_0 the output function returns x.

The transduction f_3 can be implemented by a streaming string transducer W_{cp} with a single state and two variables x and y. The symbol a is processed by the update $(x, y) = (xa, yb)$, the symbol b is processed by $(x, y) = (xy, \varepsilon)$, and the output function is x.

The following proposition states that streaming string transducers are closed under sequential composition. Note that streaming data string transducers [1] are not closed under sequential composition.

▶ Proposition 1. Given a streaming string transducer W_1 from input alphabet Σ_1 to output alphabet Σ_2 and a streaming string transducer W_2 from input alphabet Σ_2 to output alphabet Σ_3, one can effectively construct a streaming string transducer W from input alphabet Σ_1 to output alphabet Σ_3, such that for all strings w in Σ_1^*, we have that $[\![W]\!](w) = [\![W_2]\!]([\![W_1]\!](w))$ if $[\![W_1]\!](w)$ and $[\![W_2]\!]([\![W_1]\!](w))$ are both defined, and $[\![W]\!](w)$ is undefined otherwise.

Proof. We first define a notion of a summary of a computation for a streaming string transducer on an input string w. We then show how the transducer W can compute the sequential composition by simulating W_1 and keeping track of summaries of W_2 corresponding to string variables of W_1.

Given a string w, a summary of a finite state automaton is just a pair of states (q, q'), indicating that if the automaton starts reading w in a state q, it finishes in state q'. For a deterministic streaming string transducer U from Σ to Γ, a summary for a given start state q and a string w includes not only an end state q', but also a *string variable summary*, that is, a description of how the contents of the string variables get updated while processing w. Let X be the set of string variables of U. A string variable summary is a function κ from X to strings in $(\Gamma \cup X)^*$, with the same restrictions of copyless assignments as in the definition of streaming string transducers. For every valuation s from X to Γ^*, if the configuration of U is (q, s), then after processing the input string w, the configuration of U is (q', s') where $s'(x) = s(\kappa(x))$. Note that the computation of U, and hence the summary, is not influenced by the valuation s of string variables at the start of the computation on w.

A key observation is that a string variable summary can be represented by a set X' of $2 \times k$ string variables, where k is the number of string variables in X, in addition to finite bookkeeping information which is also independent in size from w. For instance, if X consists of two variables, the function κ can be $\kappa(x) = \alpha x \beta y \gamma, \kappa(y) = \iota$, or $\kappa(x) = \alpha y \beta, \kappa(y) = \delta x \iota$, or a similar combination, where $\alpha, \beta, \gamma, \iota$ are strings over Γ. The summary can be represented by X' containing in this case four string variables (that will store $\alpha, \beta, \gamma, \iota$), and a bounded amount of bookkeeping information to store how κ is constructed from X and X'.

We now describe the construction of the streaming string transducer W that computes the sequential composition of W_1 and W_2. The transducer W simulates W_1 processing the input string, and for every string variable x of W_1, it maintains a summary of W_2. It is easy to see how summaries for string variables of W_1 are maintained: for example, consider the case when W_1 executes an assignment $z = az_1z_2$. First, we construct a string variable summary

for computation of W_2 on processing the letter a. Then we compose this summary with summaries for strings stored in string variables z_1 and z_2. We explain how string variable summaries are composed on the following example. Let κ_1 be a string variable summary of a computation of W_2 on z_1 defined by $\kappa_1(x) = x\alpha y, \kappa_1(y) = \beta$. Let κ_2 be a string variable summary of a computation of W_2 on z_2 defined by $\kappa_2(x) = \epsilon, \kappa_2(y) = x\gamma y\iota$. Note that each of the strings $\alpha, \beta, \gamma, \iota$ can be stored in a string variable. Then the string variable summary κ after processing $z_1 z_2$ is defined by $\kappa_2(x) = \epsilon, \kappa_2(y) = x\alpha y\gamma\beta\iota$ (where the string $\gamma\beta\iota$ can be stored in one string variable). This finishes the construction. The number of string variables of W is $m_1 \times n_2 \times (2 \times m_2)$, where m_1 is the number of string variables of W_1, n_2 is the number of states of W_2 and m_2 is the number of string variables of n_2. ◀

As the following proposition shows, streaming string transducers are closed under conditional composition, where the condition is given as a regular language over the input alphabet. The proof is based on product construction and is similar to the proof of closure under conditional composition for streaming data string transducers [1].

▶ Proposition 2. Given two streaming string transducers W_1 and W_2 from input alphabet Σ to output alphabet Γ and a regular language L over Σ, there exists a streaming stream transducer W such that for all strings $w \in \Sigma^*$, (i) if $w \in L$, then $[\![W]\!](w) = [\![W_1]\!](w)$ if $[\![W_1]\!](w)$ is defined and is undefined otherwise, and (ii) if $w \notin L$, then $[\![W]\!](w) = [\![W_2]\!](w)$ if $[\![W_2]\!](w)$ is defined, and is undefined otherwise.

4 Heap-based Transducer Model

A *heap-based (finite-state) deterministic string transducer* H from input alphabet Σ to output alphabet Γ consists of a finite set of states Q, an initial state $q_0 \in Q$, a finite set of pointers X, an output function F from states Q to variables X, and a transition function δ from $Q \times \Sigma$ to $Q \times A_X^*$, where the set A_X of atomic actions consists of $x := \nu(a)$ and $\eta(x) := y$, for $x, y \in X$ and $a \in \Gamma$.

Given an input string over Σ, the transducer computes the output by maintaining a heap. A *heap* h consists of a finite set C of cells, a mapping ℓ from C to Γ ($\ell(c)$ denotes the output symbol stored in the cell c), a mapping η from C to $C_\perp$, where $C_\perp$ denotes the set $C \cup \{\perp\}$ ($\eta(c)$ denotes the cell that the next-pointer of the cell c points to, with $\perp$ denoting the null value), and a mapping μ from X to $C_\perp$ ($\mu(x)$ is the cell that the pointer x points to). A configuration of the transducer H consists of a state $q \in Q$ and a heap h. Initially, the state is q_0 and in the initial heap h_0, C is empty and $\mu(x) = \perp$ for each x. The action $x := \nu(a)$ creates a new cell labeled by a symbol $a \in \Gamma$ and makes x point to the new cell. The action $\eta(x) := y$ modifies the next pointer of the cell pointed to by x to make it point to the cell pointed to by y, if x is not nil; if x is nil, the action has no effect. More formally, each action in A_X updates the heap as follows: $(C, \ell, \eta, \mu) \xrightarrow{x:=\nu(a)} (C', \ell', \eta', \mu)$ holds where $C' = C \cup \{c\}$, with $c \notin C$ [1], $\ell'(c) = a$, $\eta'(c) = \perp$ and ℓ' and μ' agree with ℓ and μ, respectively, on cells in C; $(C, \ell, \eta, \mu) \xrightarrow{\eta(x):=y} (C, \ell, \eta', \mu)$ holds where if $\mu(x) = c$ then $\eta'(c) = \mu(y)$ and η' agrees with η for cells other than c (if $\mu(x) = \perp$, the action has no effect). This transition relation can be lifted to configurations: for an input symbol $a \in \Sigma$, $(q, h) \xrightarrow{a} (q', h')$ if $\delta(q, a) = (q', \alpha)$ and $h \xrightarrow{\alpha} h'$. For an input string w, if $(q_0, h_0) \xrightarrow{w} (q, h)$ then h is the output heap corresponding to w. In the output heap $h = (C, \ell, \eta, \mu)$, if $\mu(F(q)) = \perp$ then $[\![H]\!](w)$ is the empty string. If

[1] The behavior of the transducer is deterministic as long as the choice of this new cell c is according to some deterministic naming policy.

the sequence of next-pointers starting from the cell $\mu(F(q))$ contains a cycle, then $[\![H]\!](w)$ is undefined. Otherwise, let $c_0c_1\ldots c_n$ be the unique sequence of cells such that $c_0 = \mu(F(q))$, $\eta(c_i) = c_{i+1}$ for $i < n$, and $\eta(c_n) = \bot$. Then $[\![H]\!](w)$ is defined to be $\ell(c_0)\ell(c_1)\cdots\ell(c_n)$.

Such a transducer can implement the transduction f_1 as follows. After processing an input string w, suppose the current heap is a linear sequence of cells storing $w.\mathit{rev}(w)$ such that the pointer x_0 points to the first cell, the pointer x_1 points to the last cell corresponding to w and x_2 points to the first cell holding $\mathit{rev}(w)$, with an additional auxiliary pointer x_3. When the transducer reads the next input symbol, say a, it adds two new a-labeled cells in the middle of the current output, by executing the sequence $x_3 := \nu(a); \eta(x_1) := x_3; x_1 := \nu(a); \eta(x_3) := x_1; \eta(x_1) := x_2$. In the updated heap, x_3 and x_1 point to the two middle cells, and x_2 can be used as an auxiliary pointer. The result is given by the output pointer x_0.

To implement f_2, the heap-based transducer maintains two lists with pointers to the two ends of both the lists. An input symbol a is added to the end of only the first list by creating one new a-labeled cell, while an input symbol b is added to the end of both the lists by creating two new b-labeled cells. The transducer needs two states, and the state remembers whether the last symbol is b or not, and the state is used to return the pointer that points to the head of the appropriately chosen list.

To implement f_3, the heap-based transducer again maintains two lists with pointers to the two ends of both the lists. The output pointer points to the head of the first list. To process the input symbol a, it adds a a-labeled cell at the end of the first list, and a b-labeled cell at the end of the second list. To process the input symbol b, the first list is updated to the concatenation of the two (which can be implemented by changing the next-pointer of the last cell of the first list point to the first cell of the second list) and the second list is set to empty.

The two types of pointer manipulating instructions (node creation and next-pointer modification) are expressively adequate for our purpose. The class of programs in [1] allows a more general set of instructions for manipulating pointer variables (such as testing whether two pointers point to the same cell, testing whether a pointer is null, checking and updating the symbol stored at a cell pointed to by a pointer, assigning one pointer to another). From the results of this paper and the compilation of such programs into streaming transducers described in [1], it follows that such extensions do not add to expressiveness. The key missing instruction is the *traversal assignment* $x := \nu(y)$. Consider the transduction *merge*: given an input $u_1u_2\ldots u_m\#v_1v_2\ldots v_m$, output $u_1v_1u_2v_2\ldots u_mv_m$. Adding traversal assignments would allow the heap-based transducer to define the *merge* transduction by using two traversal pointers, one traversing the u part of the input and the other traversing the v part of the input. The next proposition establishes that this transduction cannot be captured by a streaming string transducer. As we show later in this paper that heap-based string transducers and streaming string transducers exactly define regular transductions, the proposition implies that adding the traversal assignment strictly increases the expressiveness of heap-based string transducers.

▶ Proposition 3. The transduction *merge* is not definable by a streaming stream transducers.

Proof. Let us consider a streaming string transducer W with n states and k string variables and let us assume that W defines *merge*. We derive a contradiction as follows. Consider the set of inputs I_m where m is the length of the sequence $u = u_1u_2\ldots u_m$ (resp. $v = v_1v_2\ldots v_m$), and where u_i is in $\{a, b\}$ for all i such that $1 \leq i \leq m$, and v_j is in $\{c, d\}$, for all j such that $1 \leq j \leq m$.

Intuitively, the proof is simple: if x_1,..., x_k are values of string variables after reading the first part of the input, $u\#$, then these will appear (each at most once) as substrings

in the final output. Assuming that $x_1, \ldots, x_k$ contain only a's and b's after processing $u\#$, and that while processing the second part of the input, v, W adds only c's and d's to the string variables, it is clear that the final output can contain at most a bounded number of alternations of letters in $\{a, b\}$ with letters in $\{c, d\}$. The transducer W therefore does not implement *merge*. We now formalize this argument.

Let us consider the set of inputs I_m as above, with m such that $2^m > n * k * 2^r$, where r is such that $2^r > k$. A string w is called *short* if $|w| \leq r$. Let us call a *short-configuration* of W the pair (q, ρ) where q is a state of W and ρ is a valuation of string variables, where each variable is assigned either a short string or a $*$. A short-configuration is an abstraction of a configuration of W, where a variable x keeps its original value if its value is a short string, otherwise it is abstracted by $*$. If we prove the following claim, we have that W does not define *merge*, and we obtain the desired contradiction.

Claim: There exist two different strings w_1 or w_2 from I_m such that (i) either the output on both w_1 and w_2 is the same, (ii) or the output on one of the strings w_1 and w_2 is incorrect.

To prove the claim, consider two different strings u_1 and u_2 in $\{a, b\}^m$ such that that W is in the same short configuration after processing u_1 and u_2. Such strings exist, as 2^m is more than the number of short configurations $(n \times k \times 2^r)$. We now construct a string v in $\{c, d\}^m$ such that the claim holds for $u_1\#v$ and $u_2\#v$. Let us look at non-short strings that W stores after processing u_1 or u_2. If these are to be used in the output, they have to have c or d on every other position. These strings thus contain guesses for the v part of the whole input string. As the length of these strings is greater than r, W cannot have stored all the possible guesses for v (as it has k string variables, and $2^r > k$). We consider v that does not contain any sequence contained in a non-short string stored by W. Thus for $u_1\#v$ and $u_2\#v$ we have that if W uses in the output a string variable with a non-short string stored after processing u_1 (or u_2), then the output is incorrect. If it does not use any such string variable, the output will be the same in both cases (as W is deterministic). ◀

5 Expressiveness

5.1 From Two-way Transducers to Heap-based Transducers

▶ **Theorem 1.** *For every two-way deterministic transducer M with n states, there exists a deterministic heap-based transducer H with $O(n^n)$ states and $O(n)$ pointer variables such that $[\![M]\!] = [\![H]\!]$.*

Proof: The proof is an extension of Shepherdson's proof [7] for standard two-way finite automata to the case of transducers. We start by describing the main ideas of the proof. The heap-based transducer has only a single (left-to-right) pass on the input string. It thus needs to precompute information on possible back-and-forth traversals. More precisely, at each position p of w and for each state u of M, the information needed by H is captured by a pair (q_u, w_u) such that

if M reads the symbol at position p in state u, the first time M reaches the position $p+1$, it will be at state q_u, having produced output w_u along this stretch. We show how such pair can be updated, even in the presence of left moves. Let us suppose that the symbol at position p is the symbol a and that in state u, M moves left to state v and outputs string α. Furthermore, let us assume that for state v, a pair (q_v, w_v) was stored in the previous step; and that in state q_v the machine M moves right on a to q_u, outputting β. Then for state u, we need to store the pair $(q_u, \alpha w_v \beta)$. The situation is depicted in Figure 1.

We need to add a few important details to this intuitive explanation. First, H can store strings w_u (possible partial outputs of M) only on the heap. In order to store a string w_u, it

will store two pointers, x_u^b and x_u^e. The two pointers will point to the first and last position of the string. Second, H cannot copy arbitrarily long strings, but it can instead use the fact that suffixes of strings represented on the heap can be shared. Third, in order to model the first forward traversal of M, we add a state m to the set of states of M. For each position p the pair (q_m, w_m) represents the situation that the first time M reaches $p+1$, it will be in state q_m and the output will be w_m. Fourth, it is possible that M never reaches the position $p+1$, either because it reached a final state and stopped, or because an error occurred, for example if M tried to move left on $\vdash$. If, for example, after a left move from a state u at position p, M reaches the final state q_f (and outputs w) before arriving at $p+1$, H represents this by the pair (q_f, w). On the other hand, if an error occurred, then H represents this by a pair (q_{err}, ϵ), where q_{err} is a special state. Fifth, note that H (as opposed to M) does not see the symbols $\vdash, \dashv$. This can be remedied by H having two copies of possible outputs of M as described above, with the additional copy assuming that the next character is $\dashv$ and simulating M on this symbol. The final output is the string starting at x, with x being the pointer variable representing the first forward traversal of M in the second copy of the possible outputs of M.

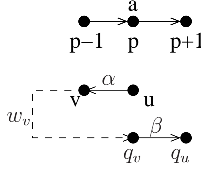

Figure 1 Representing the computation of M

We now present the construction in more details. The heap-based transducer H that we construct uses pointer-assignment instructions of the form $x = y$, where x and y are pointer variables. Such a heap-based transducer can be easily converted to a heap-based transducer H' which does not use such instructions. The state of H' records, in addition to the state of H, for every pair of pointers whether they are equal, and for every class such equal pointers, which pointer accesses the heap content. Then, to simulate the pointer assignment instruction $x = y$ in H, H' does not update x, but simply records that x and y are equal, and the pointer y points to the desired content. The construction does not change the number of pointers, but blows up the number of states by a factor of the number of possible partitions of the pointers. Since H has pointer assignment instructions, it suffices for H to have a single pointer variable x_o to be the output pointer in every state. The output function of H' maps a state to the pointer that is equal to x_o and accesses the actual content.

Let M be defined by the tuple $(\Sigma, \Gamma, Q_M, q_0^M, \delta_M)$. The heap-based transducer H is defined by the tuple $(\Sigma, \Gamma, Q_H, q_0^H, X, x_m^{b2}, \delta_H)$. Let Q_H be the set of functions from $Q_M \cup \{m\}$ to $Q_M \cup \{m, q_{err}\}$. Initial state is one which represents the identity function. The set of pointers contains four pointers x_u^b, x_u^e, x_u^{b2}, x_u^{e2} for each state u in $Q_M \cup \{m\}$. (The pointers x_u^{b2}, x_u^{e2} point to the beginning and end of the second copy of w_u.)

We explain the definition of the transition function δ_H using an example. Let us consider a state g of H such that $g(v) = q_v$ and a transition $\delta_M(u, a) = (v, -1, \alpha)$ in δ_M. Let us suppose that $\delta_M(q_v, a) = (q_u, +1, \beta)$. Then $\delta_H(g, a) = (f, A^*)$, for some state f such that $f(u) = q_u$. Recall that H stores the string w_v that M produced while moving from v to q_v on the heap, between pointers x_v^b and x_v^e. We have that while moving from u to q_u, M produces $\alpha w_v \beta$. The sequence of actions A^* puts α on the heap, make x_u^b point to the first node of α, and connects the last node of α to the node pointed to by x_v^b. Similarly, β is appended at x_v^e. So far, we have assumed that M moves right from q_v on the symbol a. If it moves left, we can again use the the information that H stores in the finite-state control and on the heap to find the state in which it will return to the current position and the string it will output in the process. Note that the process may repeat several times, and potentially even cycle. However, H has all the necessary information stored locally. If it discovers a

cycle (this happens for example if $q_v = u$), then the value of $f(u)$ will be q_{err}.

We describe how copying arbitrarily long strings on the heap is avoided by exploiting the fact that M is deterministic and the fact that suffixes of strings on the heap can be shared. Let us suppose that there are two states u and t of M such that M moves left on a symbol a to v from both u and t. Let us also suppose that at the previous step, H stored the information that from v, the first time H will move to the right it will be in state q_v and produce w_v in between. Using the idea described above, we obtain that we need to store the pair $(q_u, \alpha_u w_v \beta_u)$ for the state u, and the pair $(q_t, \alpha_t w_v \beta_t)$ for the state t. As the string w_v cannot be copied, it must be shared. As the heap is singly-linked, this would not be possible if β_u was different from β_t. However, since M is deterministic, we have that the execution from v will be identical in both cases — we have that $\beta_u = \beta_t$ (and $q_u = q_t$).

5.2 From Heap-based Transducers to Streaming Transducers

▶ **Theorem 2.** *For every deterministic heap-based string transducer H with n states and m pointer variables, there exists a deterministic streaming transducer W with $O(n2^m)$ states and $O(n)$ string variables such that $[\![H]\!] = [\![W]\!]$.*

Proof: The heap-based transducer and the streaming transducer both traverse the string only once. The difference between the two models is that one uses the heap, which allows sharing of suffixes of represented strings, whereas the other uses string variables, and thus sharing is not possible. The proof is the same as the compilation from imperative single-pass list-processing programs into streaming transducers described in [1], we describe it here for the sake of completeness.

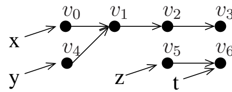

Figure 2 Singly-linked heap

The heart of the proof is therefore to show that the streaming transducer can represent the heap of the heap-based transducer using a bounded number of string variables. In order to achieve this, we adapt the approach (and terminology) of [5] for representing a singly-linked heap. A node v is called an *interruption* if it is either pointed to by a program variable or there are at least two distinct nodes with edges to v. An *uninterrupted list segment* is a finite sequence of nodes where: (i) the first node is an interruption, (ii) the next pointer of each node (except the last) points to the next node in the sequence, (iii) the last node is either an interruption or the value of its next pointer is nil, and (iv) no other node is an interruption. Consider the heap in Figure 2. The nodes v_0, v_1, v_4, v_5, v_6 are interruptions. The sequence of nodes $v_1 v_2 v_3$ is an uninterrupted list segment. It can be easily seen that for a heap-based transducer with k pointer variables, there can be at most $2k - 1$ interruptions and $2k - 1$ uninterrupted list segments. The heap can be compressed by replacing uninterrupted list segments by strings (to be stored in string variables of W).

The number of compressed heaps is exponential in the number of interruptions, and thus exponential in the number of pointer variables of H. The heap can therefore be stored by the streaming automaton W as follows: each uninterrupted list segment will be represented by a string variable. The shape of the heap (a forest) and the information on where the pointer variables of H point will be represented in the finite-state control of W. The number of states of W will be therefore linear in the number of states of H and exponential in the number of pointer variables of H.

It remains to show that the semantics of each instruction of the heap-based transducer can be modeled effectively on the representation described above. Let us consider the heap

in Figure 2 and the action $\eta(y) := x$. The shape of the heap changes in two ways: first, the node pointed to by y points to the node pointed to by x (this information is kept in the finite-state control of W); second, there will be an uninterrupted list segment starting at the node pointed to by x containing the list segment $v_0 v_1 v_2 v_3$. The (labels of nodes in) this list segment will be stored in a string variable. The string variable that in the previous step represented the list segment $v_1 v_2 v_3$ can be freed (and reused).

5.3 From Streaming Transducers to MSO

▶ **Theorem 3.** *For every deterministic streaming string transducer W there exists a deterministic MSO transducer T such that $[\![W]\!] = [\![T]\!]$.*

Proof. The proof is based on the idea that the computation of the streaming string transducer W can be represented in the copy set (from the definition of the MSO transducer). The unique sequence of states of W over a given input string w can be captured in MSO using second order existential quantification. Given a state of W and a letter in Σ, the function δ_2 is a function from the string variables to sequences in $(\Gamma \cup X)^*$. This function is what is represented using the copy set. The last step then consists of "reading out" the final output string from this representation.

We explain the encoding on an example. Let us consider the transduction f_1 and the streaming transducer W_{rev} defined in Section 3. The top of Figure 3 contains the input string abb. Recall that W_{rev} has only one state. The columns below each letter of the input string represent the assignments to the string variables: the top three nodes in a column represent the right-hand side (RHS) of the assignment $x := xa$ (resp. $x := xb$), the bottom three nodes represent the RHS of the assignment $y := ay$ (resp. $y := by$). The representation of an RHS with two symbols has three nodes. A symbol in Γ is represented by a marked edge, a variable z is represented by two nodes. The first of these nodes links to the previous column to where the representation of z begins. The second of these two nodes will be linked to from the last node representing z from the previous column. Such "linking" edges are represented by dashed (ε) lines in Figure 3.

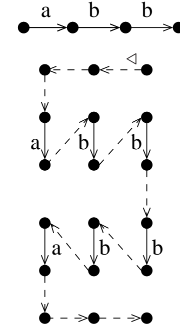

Figure 3 Representing the computation of W

In the first column, we connect the two nodes representing a variable in a right-hand side by a dashed (ε) line, as the string variables are initially empty. The last column is not used (and not pictured in Figure 3). In the column before last (the column that correspond to the last character of the input string), we also represent the effect of the output function $F(q_0) = xy$. We mark the first edge of x by a special symbol $\triangleleft$ to indicate where the output starts, and we connect the last symbol of x to the first symbol of y.

Note that the graph representing the computation of W_{rev} is not a string graph (there are disconnected nodes in the last column, not pictured), and it contains ε edges. We thus need to use an MSO transducer that deletes unused nodes (nodes unreachable from the marked node) and removes ε edges. Such a transducer is easily defined. We can then use the result that MSO transducers are closed under sequential composition [2] to conclude.

Details of the concrete MSO formulas are straightforward. We only note that the copy set of T needs to represent the right-hand side of any possible assignment to a string variable of W, therefore (the upper bound on) the size of the copy set is $n * d$, where n is the number of pointer variables of W and d is the maximal length of the string $\delta(q, a, x)$, over all q (states

of W), $a \in \Sigma$ and x (string variables of W).

6 Conclusions

The model of streaming string transducers is of potential interest for algorithmic verification of list-processing programs due to decidability of equivalence problem and correspondence to restricted classes of imperative heap-manipulating programs [1]. In this paper, we have established that its expressiveness coincides with the classical model of two-way transducers. This suggests robustness of their computing power. It also justifies the choice of primitive instructions in the corresponding class of heap-based transducers, which model single-pass programs that transform the input list using updates to a "traversal-free" heap consisting of singly-linked cells. A number of theoretical directions are worth pursuing. These include minimization of streaming string transducers, learning such transducers from input-output examples, synthesis of streaming transducers, extension to nondeterministic transducers, and extension to streaming transducers for tree-structured data.

References

1. R. Alur and P. Černý. Streaming transducers for algorithmic verification of single-pass list processing programs. In *Proceedings of the 38th Annual ACM Symposium on Principles of Programming Languages*, 2011.
2. M. Chytil and V. Jákl. Serial composition of 2-way finite-state transducers and simple programs on strings. In *Automata, Languages and Programming: Proceedings of Fourth International Colloquium, ICALP'77*, LNCS 52, pages 135–147. Springer, 1977.
3. J. Engelfriet and H. Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Trans. Comput. Log.*, 2(2):216–254, 2001.
4. E. Gurari. The equivalence problem for deterministic two-way sequential transducers is decidable. *SIAM J. Comput.*, 11(3):448–452, 1982.
5. R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference*, LNCS 3385, pages 181–198. Springer, 2005.
6. G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.
7. J. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal of Research and Development*, 3:198–200, 1959.