Transformers as Transducers

Lena Strobl
Umeå University, Sweden
lena.strobl@umu.se

Dana Angluin
Yale University, USA
dana.angluin@yale.edu

David Chiang
University of Notre Dame, USA
dchiang@nd.edu

Jonathan Rawski
San José State University, USA
jon.rawski@sjsu.edu

Ashish Sabharwal
Allen Institute for AI, USA
ashishs@allenai.org

Abstract

We study the sequence-to-sequence mapping capacity of transformers by relating them to finite transducers, and find that they can express surprisingly large classes of (total functional) transductions. We do so using variants of RASP, a programming language designed to help people ''think like transformers,'' as an intermediate representation. We extend the existing Boolean variant B-RASP to sequence-to-sequence transductions and show that it computes exactly the first-order rational transductions (such as string rotation). Then, we introduce two new extensions. B-RASP[pos] enables calculations on positions (such as copying the first half of a string) and contains all first-order regular transductions. S-RASP adds prefix sum, which enables additional arithmetic operations (such as squaring a string) and contains all first-order polyregular transductions. Finally, we show that masked average-hard attention transformers can simulate S-RASP.

1 Introduction

Transformers (Vaswani et al., 2017) have become a standard tool in natural language processing and vision tasks. They are primarily studied in terms of their expressivity (which functions they can or cannot compute) or learnability (which functions they can or cannot learn from examples). Much recent expressivity work views transformers as recognizers of formal languages, by comparing them to automata, circuits, or logic (Strobl et al., 2024). Here we take the more general view that they compute (total functional) transductions, or functions from strings to strings.

Transductions are a fundamental object in computer science, with a long history in linguistics and natural language processing (Mohri, 1997; Roark and Sproat, 2007). Many empirical tests of transformer reasoning ability use transductions to define algorithmic sequence generation tasks (e.g., Suzgun et al., 2023; Delétang et al., 2023) such as tracking shuffled objects, sorting strings, concatenating all k-th letters, or removing duplicates from a list.

This paper is the first theoretical analysis, to our knowledge, of transformers as transducers of formal languages (Figure 1). Previous work on transformers as recognizers showed that unique-hard attention transformers correspond to star-free regular languages (Yang et al., 2024); here, we prove the analogous result for transformers as transducers, that unique-hard attention transformers correspond to *aperiodic rational transductions*. We then study two superclasses of aperiodic rational transductions that are (also) analogous to star-free regular languages: *aperiodic regular transductions* (e.g., $w \mapsto w^{\mathrm{R}}$ or $w \mapsto ww$) and *aperiodic polyregular transductions* (e.g., $w \mapsto w^{|w|}$). We prove unique-hard attention transformers cannot compute all of these, but average-hard attention transformers can.

To do this, we introduce two new variants of RASP (Weiss et al., 2021), a programming language designed to make it easier to write down the kinds of computations that transformers can perform. This makes our analysis more simple, concise, and interpretable compared to describing transformers directly using linear algebra. These variants, called B-RASP[pos] and S-RASP, compute more than just the aperiodic regular and aperiodic polyregular transductions, and are interesting in their own right.

2 Preliminaries

We write $[n]$ for the set $\{0, \ldots, n-1\}$. Fix finite input and output alphabets Σ and Γ. We sometimes use special symbols # and $\dashv$, which we assume do

Transactions of the Association for Computational Linguistics, vol. 13, pp. 200–219, 2025. https://doi.org/10.1162/tacl_a_00736
Action Editor: Alexander Clark. Submission batch: 8/2024; Revision batch: 10/2024; Published 2/2025.

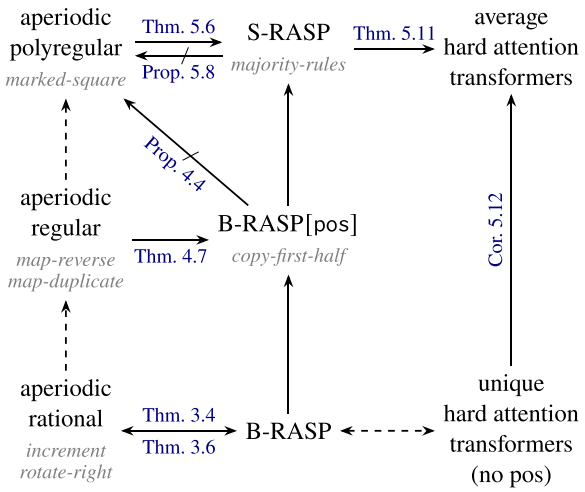


Figure 1: Overview of results of this paper. Arrows denote inclusion; dashed arrows denote inclusions that are known from previous work. Slashed arrows denote non-inclusions. The columns, from left to right, are: (1) the hierarchy of aperiodic transductions; (2) RASP variants; (3) variants of transformer encoders.

not belong to Σ or Γ. Let Σ^* and Γ^* be the sets of strings over Σ and Γ, respectively. The empty string is denoted ε. For any string w, we number its positions starting from 0, so $w = a_0 \cdots a_{n-1}$. We write uv or $u \cdot v$ for the concatenation of strings u and v, and w^{R} for the reverse of string w.

2.1 Transductions and Transducers

A transduction is a binary relation between strings in Σ^* and strings in Γ^*. Here we consider only total functional transductions, that is, functions $\Sigma^* \to \Gamma^*$, and all of our transducers define total functional transductions.

Definition 2.1 (string homomorphism)**.** A *string homomorphism* is a function $f\colon \Sigma^* \to \Gamma^*$ such that, for any strings $u, v \in \Sigma^*$, we have $f(uv) = f(u)\, f(v)$.

Definition 2.2 (deterministic finite transducer)**.** A *deterministic finite transducer* (DFT) is a tuple $T = (\Sigma, \Gamma, Q, q_0, \delta)$ where

- Σ and Γ are the input and output alphabets,
- Q is the finite set of states,
- $q_0 \in Q$ is the initial state,
- $\delta\colon Q \times (\Sigma \cup \dashv) \to \Gamma^* \times Q$ is the transition function.

The transition function δ extends to strings as follows: $\delta(q, \varepsilon) = (\varepsilon, q)$ and for $u \in \Sigma^*$ and $a \in \Sigma$, $\delta(q, ua) = (u'v', s)$ where $\delta(q, u) = (u', r)$ and $\delta(r, a) = (v', s)$ for some $r \in Q$. Then for any $w \in \Sigma^*$, we say that T transduces w to w' iff $\delta(q_0, w\dashv) = (w', r)$ for some $r \in Q$. We call a transduction *sequential* if it is definable by a DFT.

Next, we introduce several nested classes of transductions: rational, regular, and polyregular. We first give examples of transductions in these classes and informal descriptions of the classes in terms of various transducers.

In brief, a transduction is *rational* if it is definable by a *nondeterministic* transducer (the kind probably most familiar to NLP researchers, except we are assuming it is total functional). A transduction is *regular* if it is definable by a *two-way* transducer, which can be thought of as a rational transducer that can go back and forth on the input string, or a Turing machine with a read-only input tape and a write-only, one-way output tape.

Example 2.3. The following transductions are regular but not rational:

- *map-reverse*: Reverse each substring between markers.

$$| \text{ ab } | \text{ cde } | \text{ fg } | \mapsto | \text{ ba } | \text{ edc } | \text{ gf } |$$

- *map-duplicate*: Duplicate each substring between markers.

$$| \text{ ab } | \text{ cde } | \mapsto | \text{ abab } | \text{ cdecde } |$$

A transduction is *polyregular* if it is definable by a *pebble* transducer, which is a two-way transducer augmented with a stack of up to k *pebbles* (Bojańczyk, 2022). It can push the current head position onto the stack and jump to the beginning of the string, and it can pop the top pebble from the stack and jump to that pebble's position. It can read the symbol at every pebble, and it can compare the positions of the pebbles.

Example 2.4. The transduction *marked-square* is polyregular but not regular. It makes $|w|$ many copies of w separated by bars with successively longer prefixes marked, here by uppercasing:

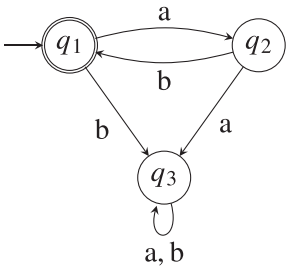
$$\text{abaa} \mapsto | \text{ Abaa } | \text{ ABaa } | \text{ ABAa } | \text{ ABAA } |$$

Next we restrict to the *aperiodic* subsets of these classes, and give formal definitions of these subclasses as composition closures of sets of *prime transductions*. We will use these definitions for the rest of the paper.

Definition 2.5 (aperiodicity)**.** Let T be a deterministic finite automaton or transducer. For any input string w, there is a binary relation on states, $p \xrightarrow{w}_T q$, which holds iff $\delta(p, w)$ arrives at state q; if T is a DFT, this means that $\delta(p, w) = (w', q)$ for some w'. Then T is *aperiodic* (or *counter-free*) if there is an $N \geq 0$ (depending on T) such that for all strings $w \in \Sigma^*$ and all $n \geq N$, the relations $\xrightarrow{w^n}_T$ and $\xrightarrow{w^{n+1}}_T$ are the same.

Aperiodic deterministic finite automata (DFAs) are equivalent to star-free regular expressions and first-order logic with order (Schützenberger, 1965; McNaughton and Papert, 1971). They are also equivalent to masked hard-attention transformers (Yang et al., 2024). We take this equivalence as our starting point.

Example 2.6. The regular language $(\mathrm{ab})^*$ is definable by an aperiodic DFA (with $N = 2$):



But $(\mathrm{aa})^*$ is not defined by any aperiodic DFA: The relations $\xrightarrow{\mathrm{a}^n}_T$ and $\xrightarrow{\mathrm{a}^{n+1}}_T$ are always different.

Each of the classes of transductions described above has an aperiodic subclass.

Definition 2.7. *Aperiodic sequential transductions* (which include string homomorphisms) are those defined by aperiodic DFTs.

Aperiodic rational transductions are the composition closure of aperiodic sequential transductions and *right-to-left aperiodic sequential transductions*, that is, transductions that can be expressed as $w \mapsto f(w^{\mathrm{R}})^{\mathrm{R}}$, where f is aperiodic sequential.[1]

Aperiodic regular transductions are the composition closure of aperiodic sequential transductions and the transductions *map-reverse* and *map-duplicate* (Ex. 2.3).[2]

Aperiodic polyregular transductions (Bojańczyk, 2018, Def. 1.3) are the composition closure of aperiodic regular transductions and the transduction *marked-square* (Ex. 2.4).

2.2 Transformers

We assume familiarity with transformers (Vaswani et al., 2017) and describe a few concepts briefly. For more detailed definitions, please see the survey by Strobl et al. (2024).

In standard attention, attention weights are computed from attention scores using the softmax function. In *average-hard attention* (Pérez et al., 2021; Merrill et al., 2022), each position i attends to those positions j that maximize the score $s_{i,j}$. If there is more than one such position, attention is divided equally among them. In *unique-hard attention* (Hahn, 2020), exactly one maximal element receives attention. In leftmost hard attention, the leftmost maximum element is chosen, while in rightmost hard attention, the rightmost maximum element is chosen.

In this work, we use RASP (Weiss et al., 2021) as a proxy for transformers. Specifically, we use extensions of B-RASP, a version of RASP restricted to Boolean values (Yang et al., 2024). B-RASP is equivalent to masked hard-attention transformer encoders with leftmost and rightmost hard attention, and with strict future and past masking.

3 B-RASP and Unique Hard Attention Transformers

In this section, in order to facilitate our study of transformers and how they relate to classes of transductions, we modify the definition of B-RASP to compute transductions and use it to show that unique-hard attention transformers are equivalent to aperiodic rational transductions.

In Sections 4 and 5, we consider two extensions: B-RASP[pos] adds position information,

[1] Nguyễn et al. (2023, fn. xii) characterize aperiodic rational transductions using just one aperiodic sequential transduction and one right-to-left aperiodic sequential transduction, and Filiot et al. (2016, Prop. 3) use a closely related characterization in terms of *bimachines*. Here we use a composition of any number of transductions, which is equivalent because aperiodic rational transductions are closed under composition (Carton and Dartois, 2015, Thm. 10).

[2] This characterization is given by Nguyễn (2021, p. 15). It is also given by Bojańczyk and Stefański (2020, Thm. 18) for the more general setting of infinite alphabets constructed from *atoms*; our definition here corresponds to the special case of finite alphabets (that is, where the set of atoms is empty).

`in`	0	1	0	1	1
`not`	1	0	1	0	0
`carry`	0	0	1	1	1
`out`	0	1	1	0	0

Table 1: B-RASP computation for *increment*.

and S-RASP also includes an operator for prefix sum. These have various correspondences both with more realistic transformers and with larger classes of transductions.

3.1 Definition and Examples

We give an example first, followed by a more systematic definition.

Example 3.1. The following B-RASP program computes the transduction *increment*, which takes as input a binary number (with its high-order bit on the left) and increments it, ignoring overflow.

$$\begin{aligned}\texttt{not}(i) &= \text{`1' if } \texttt{in}(i) = \text{`0'else `0'}\\ \texttt{carry}(i) &= \blacktriangleright_j\,[j > i, \texttt{in}(j) = \text{`0'}]\ \bot : \top\\ \texttt{out}(i) &= \texttt{not}(i) \text{ if } \texttt{carry}(i) \text{ else } \texttt{in}(i)\end{aligned}$$

Table 1 shows a sample run. The input string is stored in $\texttt{in}(0), \ldots, \texttt{in}(n-1)$. The vector `not` is the bitwise negation of `in`. The if expression is in Python-style syntax: if $\texttt{in}(i) =$ '0', then $\texttt{not}(i) =$ '1'; otherwise, $\texttt{not}(i) =$ '0'. The vector `carry` tests at each position i whether there is a carry at that position, that is, whether at every position $j > i$ the symbol is a '1'. It can be read as: ''Find the rightmost ($\blacktriangleright$) position j such that $j > i$ and $\texttt{in}(j) =$ '0'. If there is such a position, return false ($\bot$); if there is no such position, return true ($\top$).'' Finally, the vector `out` is the output of the program.

We give a definition of B-RASP that is equivalent to that of Yang et al. (2024), extended to transductions. For now, we consider B-RASP programs for length-preserving transductions, and will later consider two schemes for defining non-length-preserving transductions as needed.

There are two types of values: Booleans from $\{\top, \bot\}$, and symbols from a finite alphabet Δ. These are stored in *vectors*, which all share the same length n, mirroring the transformer encoders they are intended to model.

A B-RASP program receives an input string $w = a_0 \cdots a_{n-1}$ represented as a symbol vector `in`, where $\texttt{in}(i) = a_i$ for $i \in [n]$.

A B-RASP program is a sequence of definitions of the form $P(i) = \rho$, where P is a vector name, i is a variable name, and ρ is a right-hand side, to be defined below. The type of P is the type of ρ. No two definitions can have the same left-hand side.

The syntax of B-RASP expressions, with Boolean (bool) and symbolic (char) type, is:

$$\begin{aligned}e^{\mathsf{bool}} ::= {} & \top \mid \bot \mid P^{\mathsf{bool}}(i) \mid e^{\mathsf{char}} = e^{\mathsf{char}}\\ & \mid e^{\mathsf{bool}} \wedge e^{\mathsf{bool}} \mid e^{\mathsf{bool}} \vee e^{\mathsf{bool}} \mid \neg e^{\mathsf{bool}}\\ e^{\mathsf{char}} ::= {} & \text{`a'} \mid \text{`b'} \mid \cdots \mid P^{\mathsf{char}}(i)\\ & e^{\mathsf{char}} \text{ if } e^{\mathsf{bool}} \text{ else } e^{\mathsf{char}}\end{aligned}$$

where P is a vector name and i is a variable name. We write $\text{FV}(e)$ for the variables occurring in e. As mentioned above, conditional expressions use Python syntax: e_1 if e_2 else e_3 means ''if e_2 evaluates to $\top$, then return e_1; otherwise, return e_3.'' The syntax of expressions could be extended to include arbitrary operations on Booleans or symbols.

Each definition has one of the following forms:

1. *Position-wise operations* $P(i) = e$, where e is an expression such that $\text{FV}(e) \subseteq \{i\}$.
2. *Attention operations,* which have one of the two forms

$$\begin{aligned}P(i) &= \blacktriangleleft_j\,[M(i,j), S(i,j)]\ \ V(j) : D(i)\\ P(i) &= \blacktriangleright_j\,[M(i,j), S(i,j)]\ \ V(j) : D(i)\end{aligned}$$

   where:

   - The *choice function* is either leftmost ($\blacktriangleleft$) or rightmost ($\blacktriangleright$).
   - $M(i,j)$ is a *mask predicate*, one of
     1. *no masking*: $M(i,j) = \top$
     2. *future masking*: $M(i,j) = (j < i)$ or $M(i,j) = (j \leq i)$
     3. *past masking*: $M(i,j) = (j > i)$ or $M(i,j) = (j \geq i)$.
   - $S(i,j)$ is an *attention predicate*, given by a Boolean expression with $\text{FV}(S(i,j)) \subseteq \{i,j\}$.
   - $V(j)$ is a *value function*, given by a Boolean or symbol expression with $\text{FV}(V(j)) \subseteq \{j\}$.
   - $D(i)$ is a *default function*, given by a Boolean or symbol expression with $\text{FV}(D(i)) \subseteq \{i\}$.

The attention operation defines a new vector P, as follows. For $i \in [n]$ and choice function $\blacktriangleleft$, j_i is the minimum $j \in [n]$ such that $M(i,j) = \top$ and $S(i,j) = \top$, if any, and $P(i)$ is set to the value $V(j_i)$. If there is no such j, then $P(i)$ is set to the value $D(i)$. If the choice function is $\blacktriangleright$ then j_i is the maximum $j \in [n]$ such that $M(i,j) = \top$ and $S(i,j) = \top$, if any, and $P(i)$ is set to the value $V(j_i)$. If there is no such j, then $P(i)$ is set to the value $D(i)$.

The output of a B-RASP program is given in a designated symbol vector `out`, which has the same form as the input vector `in`.

Example 3.2. The rational transduction *rotate-right* rotates the input string to the right by one symbol, moving the last symbol to the first position. For example,

$$\text{abc} \mapsto \text{cab}$$

The following B-RASP program computes *rotate-right*:

$$\begin{aligned}
\texttt{right}(i) &= \blacktriangleleft_j\,[j > i, \top]\ \texttt{in}(j) : \text{`\#'} \\
\texttt{last}(i) &= \blacktriangleleft_j\,[\top, \texttt{right}(j) = \text{`\#'}]\ \texttt{in}(j) : \text{`\#'} \\
\texttt{left}(i) &= \blacktriangleright_j\,[j < i, \top]\ \texttt{in}(j) : \text{`\#'} \\
\texttt{out}(i) &= \texttt{left}(i) \text{ if } \texttt{left}(i) \neq \text{`\#'} \text{ else } \texttt{last}(i)
\end{aligned}$$

An example run is in Table 2. The vector `right`, at each position i, records the symbol immediately to the right of i (or '#' if there is no symbol to the right). We distinguish the position j of the rightmost symbol in the input string by testing whether $\texttt{right}(j) = \text{`\#'}$, and propagate its input symbol to all positions in the vector `last`. The vector `left` records the symbol immediately to the left of each position (or '#' if there is no symbol to the left). To compute the output vector `out`, the first position takes on the value of the rightmost symbol of the input string and each other position takes on the value of its left neighbor, via a position-wise operation.

3.2 Packed Outputs

So far, we have defined B-RASP to encompass only length-preserving transductions. But even some simple classes of transductions, like string homomorphisms, are not length-preserving.

To address this, we allow the program to output a vector containing strings up to some length k instead of a vector of symbols. For any finite alphabet A, let $A^{\leq k}$ denote the set of all strings over A of length at most k (including the empty string ε).

`in`	a	b	c	b	b	a	c
`right`	b	c	b	b	a	c	#
`last`	c	c	c	c	c	c	c
`left`	#	a	b	c	b	b	a
`out`	c	a	b	c	b	b	a

Table 2: B-RASP computation for *rotate-right*.

The input vector is still a vector of input symbols: $a_0 a_1 \cdots a_{n-1}$, where $a_i \in \Sigma$ for $i \in [n]$. However, the output vector is a vector of symbols over the alphabet $\Gamma^{\leq k}$ for some k. The output vector is a *k-packed representation* of a string u if the concatenation of the strings at positions $0, \ldots, n-1$ is u. There may be many different k-packed representations of the same string. For an input string of length n, the output string has length at most kn. Packed outputs make it possible to compute any string homomorphism, as in the following example.

Example 3.3. Apply the homomorphism a $\mapsto$ aa, b $\mapsto$ ccb to an input string over the alphabet $\{\text{a}, \text{b}\}$.

$$\texttt{out}(i) = \text{`aa'} \text{ if } \texttt{in}(i) = \text{`a'} \text{ else `ccb'}$$

3.3 B-RASP Defines Exactly the Aperiodic Rational Transductions

Examples 3.1 and 3.2 show that B-RASP can compute some aperiodic rational transductions that are not sequential. The following theorem shows that B-RASP can compute only aperiodic rational transductions.

Theorem 3.4. *Any B-RASP program with packed outputs defines an aperiodic rational transduction.*

Proof. Let $\mathcal{P}$ be a B-RASP program. By Lemma 12 of Yang et al. (2024), $\mathcal{P}$ can be rewritten so that every score predicate $S(i,j)$ depends only on j. Denote the sequence of vectors of $\mathcal{P}$ as $P_1, \ldots, P_m$, and treat the input vector `in` as P_0. We prove by induction that the first k operations of $\mathcal{P}$ can be converted to a

composition of left-to-right and right-to-left aperiodic sequential transductions. The output of the composition is the sequence of $(k+1)$-tuples $(x_{0,0},\ldots,x_{0,k}),\ldots,(x_{n-1,0},\ldots,x_{n-1,k})$, where for $i \in [n]$ and $j \in [k+1]$, we have $x_{i,j} = P_j(i)$.

If $k = 1$, we just construct the identity transducer. If $k > 1$, assume that the first $k-1$ operations have been converted to a composition of transductions. If P_k is a position-wise operation, it can be computed by a one-state DFT that appends the value of $x_{i,k} = P_k(i)$ onto the end of the input k-tuple. The interesting cases are the attention operations.

Case $P_k(i) = \blacktriangleright_j\,[j < i, P_s(i)]\ P_v(j) : P_d(i)$, where $s, v, d < k$: Let T be the set of values in the type of P_k. Then we construct the following (left-to-right) DFT. Starting from the first position, it appends $P_d(i)$ onto the end of the input k-tuple. Every time it reaches a position j where $P_s(j)$ is true, it switches, starting from position $j+1$, to appending $P_v(j)$. In the following, $\vec{x}$ is the input k-tuple, x_r is the component of $\vec{x}$ with index r, and $(\vec{x}, x)$ is the $(k+1)$-tuple obtained by appending the element x to the end of $\vec{x}$.

$$Q = \{q_{\text{def}}\} \cup \{q_x | x \in T\}$$
$$\delta(q_{\text{def}}, \vec{x}) = \begin{cases} ((\vec{x}, x_d), q_{x_v}) & x_s = \top \\ ((\vec{x}, x_d), q_{\text{def}}) & x_s = \bot \end{cases}$$
$$\delta(q_x, \vec{x}) = \begin{cases} ((\vec{x}, x), q_{x_v}) & x \in T, x_s = \top \\ ((\vec{x}, x), q_x) & x \in T, x_s = \bot. \end{cases}$$

To see that this is counter-free: Let u be any string. If u contains a tuple $\vec{x}$ such that $x_s = \top$, let $\vec{x}$ be the rightmost such tuple. Then $q \xrightarrow{u} q_{x_v}$ for all q, so $(\xrightarrow{u^i}) = (\xrightarrow{u^{i+1}})$ for all $i \geq 1$. If u does not contain such a tuple, then $q \xrightarrow{u} q$ for all q, so $(\xrightarrow{u^i}) = (\xrightarrow{u^{i+1}})$ for all $i \geq 0$.

Case $P_k(i) = \blacktriangleleft_j\,[j < i, P_s(j)]\ P_v(j) : P_d(i)$, where $s, v, d < k$: Let Q and T be as above. Then we construct the following DFT. Starting from the first position, it appends $P_d(i)$ onto the end of the input k-tuple. The first time it reaches a position j where $P_s(j)$ is true, it switches to appending $P_v(j)$, from position $j+1$ to the end.

$$\delta(q_{\text{def}}, \vec{x}) = \begin{cases} ((\vec{x}, x_d), q_{x_v}) & x_s = \top \\ ((\vec{x}, x_d), q_{\text{def}}) & x_s = \bot \end{cases}$$
$$\delta(q_x, \vec{x}) = ((\vec{x}, x), q_x) \qquad x \in T.$$

To see that this is counter-free: Same as the previous case, except $\vec{x}$ is the *leftmost* tuple in u such that $v_s = \top$.

The cases

$$P_k(i) = \blacktriangleleft_j\,[j > i, p_s(j)]\ P_v(j) : P_d(i)$$
$$P_k(i) = \blacktriangleright_j\,[j > i, p_s(j)]\ P_v(j) : P_d(i)$$

are the same, but using right-to-left transducers.

Case $P_k(i) = \blacktriangleleft_j\,[\top, P_s(j)]\ P_v(j) : P_d(i)$, where $s, v, d < k$: This operation could be replaced by the following sequence of three operations, which are covered in the preceding cases.

$$R_k(i) = \blacktriangleleft_j\,[j > i, p_s(j)]\ P_v(j) : P_d(i)$$
$$C_k(i) = P_v(i) \text{ if } P_s(i) \text{ else } R_k(i)$$
$$P_k(i) = \blacktriangleleft_j\,[j > i, P_s(j)]\ P_v(j) : C_k(i).$$

Here, $R_k(i)$ is the value from the leftmost $j > i$ with $P_s(j) = \top$ (if any), else $P_d(i)$; then $C_k(i)$ is the value from the leftmost $j \geq i$ with $P_s(j) = \top$ (if any), else $P_d(i)$; finally, $P_k(i)$ is the value from the leftmost j overall with $P_s(j) = \top$ (if any), else $P_d(i)$.

Case $P_k(i) = \blacktriangleright_j\,[\top, P_s(j)]\ P_v(j) : P_d(i)$ is the mirror image of the previous case. □

For the converse, we need the following lemma.

Lemma 3.5. *If $\mathcal{P}$ is a B-RASP program with packed outputs and f is an aperiodic sequential transduction, there is a B-RASP program with packed outputs that computes $f \circ \mathcal{P}$.*

Proof. We can adapt the proof of Lemma 19 of Yang et al. (2024). By the Krohn-Rhodes decomposition theorem for aperiodic sequential transductions (Pradic and Nguyễn, 2020, Thm. 4.8), f is equivalent to the sequential composition of finitely many two-state aperiodic DFTs. Hence, without loss of generality, we can assume that f is defined by a two-state aperiodic DFT T. This machine T is an *identity–reset* transducer, which means that for any symbol $\sigma \in \Sigma$, the state transformation $\xrightarrow{\sigma}_T$ either is the identity (maps both states to themselves) or *resets* to one of the states q (maps both states to q). For each state q of T, let R_q be the set of symbols that reset to q. Let $R = \bigcup_q R_q$ and $I = \Sigma \setminus R$. Let q_1 be the start state

and q_2 the other state. We write $T(q, w) = w'$ if $\delta(q, w) = (w', q')$ for some q'.

Modify $\mathcal{P}$ so that its output vector is a fresh vector $\mathtt{z}$ instead of $\mathtt{out}$. Then $f \circ \mathcal{P}$ is defined by appending the following operations to $\mathcal{P}$:

$$\mathtt{state}_q(i) = \blacktriangleright_j \left[j < i, \bigvee_{\substack{uav \in \Gamma^{\leq k} \\ a \in R, v \in I^*}} \mathtt{z}(j) = uav \right] \left(\bigvee_{\substack{uav \in \Gamma^{\leq k} \\ a \in R_q, v \in I^*}} \mathtt{z}(j) = uav \right) : q = q_1$$

$$\mathtt{sym}(i) = \blacktriangleleft_j [j > i, \top] \ \mathtt{z}(i) : \mathtt{z}(i) \dashv$$

$$\mathtt{out}(i) = T(q_1, \mathtt{sym}(i)) \text{ if } \mathtt{state}_{q_1}(i) \text{ else } T(q_2, \mathtt{sym}(i))$$

Vector $\mathtt{state}_q(i)$ tests whether T is in state q just before reading (packed) symbol w_i. It does so by searching for the rightmost symbol a that resets to any state. If a exists and resets in particular to q, then T must still be in state q; otherwise, it is not. But if a does not exist, then T must still be in the start state q_1. Vector $\mathtt{sym}(i)$ simply appends $\dashv$ to the last position. Finally, $\mathtt{out}$ maps $\mathtt{sym}(i)$ to $T(q, \mathtt{sym}(i))$ (where q is the state just before reading w_i). □

Theorem 3.6. *For any aperiodic rational transduction $f \colon \Sigma^* \to \Gamma^*$, there is a B-RASP program $\mathcal{P}$ with packed outputs that computes f.*

Proof. The transduction f can be written as $f_R \circ f_L$, where f_L is an aperiodic sequential transduction and f_R is a right-to-left aperiodic sequential transduction (Def. 2.7). The identity transduction can clearly be computed by a B-RASP program, and by Lem. 3.5 there is a B-RASP program computing f_L. Finally, Lem. 3.5 can be easily modified to apply also to f_R, using the mirror images of $\mathtt{state}_q$ and $\mathtt{sym}$ above. □

3.4 Unique-hard Attention Transformers Compute Exactly the Aperiodic Rational Transductions

Yang et al. (2024) show that a B-RASP program can be simulated by a unique-hard attention transformer with no position information besides masking, and vice versa. With Thms. 3.4 and 3.6, this implies that masked unique-hard attention transformers with packed outputs can compute exactly the aperiodic rational transductions.

4 B-RASP with Positions

4.1 Definition

We extend B-RASP to B-RASP[$\mathtt{pos}$], which adds a type nat for integers in $[n]$, and vectors containing integers.

We extend the syntax of expressions as follows:

$$\begin{aligned} e^{\mathtt{nat}} &::= 0 \mid 1 \mid e^{\mathtt{nat}} + e^{\mathtt{nat}} \mid e^{\mathtt{nat}} - e^{\mathtt{nat}} \\ e^{\mathtt{bool}} &::= \cdots \\ &\quad \mid c^{\mathtt{bool}} \qquad \text{where } \mid \mathrm{FV}(c^{\mathtt{bool}}) \mid \leq 1 \\ c^{\mathtt{bool}} &::= e^{\mathtt{nat}} = e^{\mathtt{nat}} \mid e^{\mathtt{nat}} < e^{\mathtt{nat}} \mid e^{\mathtt{nat}} \leq e^{\mathtt{nat}} \\ &\quad \mid e^{\mathtt{nat}} > e^{\mathtt{nat}} \mid e^{\mathtt{nat}} \geq e^{\mathtt{nat}} \end{aligned}$$

where $\cdots$ means all of the productions from the syntax of B-RASP. Then vector definitions are extended as follows.

1. There is a pre-defined integer vector $\mathtt{pos}(i)$, whose value is simply i at every position $i \in [n]$.
2. There are position-wise operations $P(i) = e^{\mathsf{nat}}$, where $\mathrm{FV}(e^{\mathsf{nat}}) \subseteq \{i\}$. Addition and subtraction have their usual meaning, but values less than 0 are replaced by 0 and values greater than $n - 1$ are replaced by $n - 1$. (Since this is not associative, we fully parenthesize arithmetic expressions.)
3. There are position-wise operations $P(i) = c^{\mathsf{bool}}$, where $\mathrm{FV}(c^{\mathsf{bool}}) \subseteq \{i\}$. The operators $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$ have their usual meaning.
4. In B-RASP, $S(i, j)$ was a Boolean expression (e^{bool}); in B-RASP[$\mathtt{pos}$], it can be either a Boolean expression (e^{bool}) or, as a special case, $V_1(i) = V_2(j)$, where V_1 and V_2 are previously defined integer vectors. We emphasize that only tests for equality are allowed (not, for example, $V_1(i) < V_2(j)$). This restriction is used in the transformer simulation in Section 5.5.

4.2 Examples

Informally, we omit a default value from a leftmost or rightmost operation if the operation is such that the default value will never be taken.

in	input	|	a	b	|	c	d	e	|
pos	position	0	1	2	3	4	5	6	7
prev	previous '|'	0	0	0	0	3	3	3	3
next	next '|'	3	3	3	7	7	7	7	0
src	source	3	2	1	4	6	5	4	0
y1	in(src(i))	|	b	a	c	e	d	c	|
out	output	|	b	a	|	e	d	c	|

Table 3: Example B-RASP[pos] computation for *map-reverse*. Details in Ex. 4.1.

Example 4.1 (*map-reverse*)**.** Reverse each substring between markers.

$$\text{| ab | cde | fg } \mapsto \text{| edc | gf |}$$

$$\begin{aligned}
\texttt{prev}(i) &= \blacktriangleright_j\,[j < i, \texttt{in}(j) = \text{'|'}]\ \texttt{pos}(j) : 0\\
\texttt{next}(i) &= \blacktriangleleft_j\,[j > i, \texttt{in}(j) = \text{'|'}]\ \texttt{pos}(j) : 0\\
\texttt{src}(i) &= \texttt{prev}(i) + \texttt{next}(i) - \texttt{pos}(i)\\
\texttt{y1}(i) &= \blacktriangleleft_j\,[\top, \texttt{src}(i) = \texttt{pos}(j)]\ \texttt{in}(j)\\
\texttt{out}(i) &= \text{'|' if } \texttt{in}(i) = \text{'|' else } \texttt{y1}(i)
\end{aligned}$$

An example run is in Table 3.

Above, the vector y1 just retrieves, for each i, the input symbol at position $\texttt{src}(i)$. This idiom is so common that we will write it using the syntactic sugar:

$$\texttt{y1}(i) = \texttt{in}(\texttt{src}(i)).$$

Example 4.2 (*map-duplicate*)**.** Duplicate each substring between markers.

$$\text{|ab|cde|} \mapsto \text{|abab|cdecde}$$

$$\begin{aligned}
\texttt{prev}(i) &= \blacktriangleright_j\,[j < i, \texttt{in}(j) = \text{'|'}]\ \texttt{pos}(j) : 0\\
\texttt{next}(i) &= \blacktriangleleft_j\,[j > i, \texttt{in}(j) = \text{'|'}]\ \texttt{pos}(j) : 0\\
\texttt{nowrap}(i) &= \texttt{pos}(i) + (\texttt{pos}(i) - \texttt{prev}(i) - 1)\\
\texttt{wrap}(i) &= \texttt{pos}(i) - (\texttt{next}(i) - \texttt{pos}(i))\\
\texttt{src1}(i) &= \texttt{nowrap}(i) \text{ if } \texttt{nowrap}(i) < \texttt{next}(i)\\
&\qquad \text{else } \texttt{wrap}(i)\\
\texttt{src2}(i) &= \texttt{nowrap}(i) + 1 \text{ if } \texttt{nowrap}(i) + 1 < \texttt{next}(i)\\
&\qquad \text{else } \texttt{wrap}(i) + 1\\
\texttt{out}(i) &= \text{'|' if } \texttt{in}(i) = \text{'|'}\\
&\qquad \text{else } \texttt{in}(\texttt{src1}(i)) \cdot \texttt{in}(\texttt{src2}(i))
\end{aligned}$$

Here $\cdot$ denotes string concatenation over $\Gamma^{\leq k}$. An example run is in Table 4. Note that $\texttt{nowrap}(6) = 7$, not 8, because addition and subtraction are clipped to lie in $[0, n-1]$.

in	input	|	a	b	|	c	d	e	|
pos	position	0	1	2	3	4	5	6	7
prev	previous '|'	0	0	0	0	3	3	3	3
next	next '|'	3	3	3	7	7	7	7	0
nowrap		0	1	3	5	4	6	7	7
wrap		0	0	1	0	1	3	5	7
src1	left symbol	0	1	1	5	4	6	5	7
src2	right symbol	1	2	2	6	5	4	6	7
out	output	|	ab	ab	|	cd	ec	de	|

Table 4: Example B-RASP[pos] computation for *map-duplicate*. Details in Ex. 4.2.

in	input	a	b	c	a	a	b	c	b	b
po	i	0	1	2	3	4	5	6	7	8
last	$n-1$	8	8	8	8	8	8	8	8	8
sum	$\min(2i, n-1)$	0	2	4	6	8	8	8	8	8
out	output	a	b	c	a	ε	ε	ε	ε	ε

Table 5: Example B-RASP[pos] computation for *copy-first-half*. Details in Ex. 4.3.

Example 4.3 (*copy-first-half*)**.** Copy just the first half of the input string, rounding down.

$$\text{abcaabcbb} \mapsto \text{abca}$$

$$\begin{aligned}
\texttt{last}(i) &= \blacktriangleright_j\,[\top, \top]\ \texttt{pos}(j)\\
\texttt{sum}(i) &= \texttt{pos}(i) + \texttt{pos}(i)\\
\texttt{out}(i) &= \texttt{in}(i) \text{ if } \texttt{sum}(i) < \texttt{last}(i) \text{ else '}\varepsilon\text{'}
\end{aligned}$$

An example run is in Table 5.

Proposition 4.4. *The transduction copy-first-half is neither regular nor polyregular.*

Proof. Both regular and polyregular transductions preserve regular languages under inverse (Bojańczyk, 2018, Thm 1.7). The inverse of the regular language a^* under *copy-first-half* is the set of words of the form $a^n w$ where $|w| \leq n$, which is not regular if $|\Sigma| > 1$, so the transduction *copy-first-half* is neither regular nor polyregular.[3] □

Example 4.5 (*residues-mod-m*)**.** Let m be a positive integer and define the transduction *residues-mod-m* to map any input $a_0 a_1 \cdots a_{n-1}$ to the sequence $b_0 b_1 \cdots b_{n-1}$ where $b_i = i \bmod m$. This transduction is rational but not aperiodic.

Proposition 4.6. *For any m, B-RASP*[pos] *can compute the transduction residues-mod-m.*

[3]Thanks to an anonymous reviewer for suggesting this argument to us.

Proof. For concreteness, we give a program for the case $m = 3$, which is easily generalized. The second line deals with clipping to $n - 1$.

$$\begin{aligned}
\texttt{sum3}(i) &= \texttt{pos}(i) + \texttt{pos}(i) + \texttt{pos}(i) \\
\texttt{sum3c}(i) &= \texttt{sum3}(i) \text{ if } \texttt{sum3}(i) = \texttt{sum3}(i-1) + 3 \\
&\quad \text{else } 0 \\
\texttt{mult3}(i) &= \blacktriangleleft_j \left[\top, \texttt{pos}(i) = \texttt{sum3c}(j)\right] \ \top : \bot \\
\texttt{out}(i) &= 0 \text{ if } \texttt{mult3}(i) \text{ else } 1 \text{ if } \texttt{mult3}(i-1) \\
&\quad \text{else } 2
\end{aligned}$$

□

4.3 Expressivity

Theorem 4.7. *B-RASP*[pos] *programs with packed outputs can compute all aperiodic regular transductions.*

Proof. If f is an aperiodic regular transduction, then by Def. 2.7, it can be decomposed into a composition of transductions, each of which is (a) aperiodic sequential, (b) *map-reverse*, or (c) *map-duplicate*. We convert f to a B-RASP[pos] program by induction on the number of functions in the composition. Case (a) is the same as the proof of Lem. 3.5, *mutatis mutandis*. The following two lemmas handle the other two cases. □

Lemma 4.8. *If $\mathcal{P}$ is a B-RASP*[pos] *program with packed outputs, then there is a B-RASP*[pos] *program with packed outputs that computes map-reverse $\circ\, \mathcal{P}$.*

Proof. We'd like to compose $\mathcal{P}$ with the program of Ex. 4.1, but since $\mathcal{P}$ uses packed outputs, we must adapt Ex. 4.1 to use packed inputs. Define the functions *head*, *body*, and *tail* as follows. If w does not contain the separator (|), then $head(w) = tail(w) = w$ and $body(w) = \varepsilon$. Otherwise, factor w as xyz, where x is the prefix of w before the first separator and z is the suffix of w after the last separator, and $head(w) = x$, $body(w) = y$, and $tail(w) = z$. Position-wise operations allow the application of these functions, as well as *map-reverse* itself and the test of whether a string contains a symbol ($w \mapsto (a \in w)$), to bounded-length strings. Modify $\mathcal{P}$ so that its output vector is a fresh vector z instead of out. Then append the following operations to $\mathcal{P}$:

$$\begin{aligned}
\texttt{prev}(i) &= \blacktriangleright_j \left[j < i, \text{`l'} \in z(j)\right] \ \texttt{pos}(j) : n-1 \\
\texttt{next}(i) &= \blacktriangleleft_j \left[j > i, \text{`l'} \in z(j)\right] \ \texttt{pos}(j) : 0 \\
\texttt{head}(i) &= head(\texttt{z}(i)) \\
\texttt{body}(i) &= body(\texttt{z}(i)) \\
\texttt{tail}(i) &= tail(\texttt{z}(i)) \\
\texttt{nosep}(i) &= \texttt{z}(\texttt{next}(i) - (\texttt{pos}(i) - \texttt{prev}(i)))^{\mathrm{R}} \\
\texttt{ptail}(i) &= \texttt{tail}(\texttt{prev}(i))^{\mathrm{R}} \\
\texttt{rbody}(i) &= map\text{-}reverse(\texttt{body}(i)) \\
\texttt{nhead}(i) &= \texttt{head}(\texttt{next}(i))^{\mathrm{R}} \\
\texttt{sep}(i) &= \texttt{ptail}(i) \cdot \texttt{rbody}(i) \cdot \texttt{nhead}(i) \\
\texttt{out}(i) &= \texttt{sep}(i) \text{ if `l'} \in \texttt{z}(i) \text{ else } \texttt{nosep}(i)
\end{aligned}$$

□

To see how this works, consider a packed symbol $\texttt{z}(i)$. If it contains at least one separator, it is parsed into $head$, $body$, and $tail$ as xyz. The correct output for position i is computed in $\texttt{sep}(i)$, and consists of replacing x by the reverse of the tail of the closest left neighbor that has a separator, replacing y with *map-reverse*(y), and replacing z by the reverse of the head of the closest right neighbor that has a separator. If $\texttt{z}(i)$ contains no separator, then it appears in a maximal subsequence $w_0, w_1, \ldots, w_{k-1}$ with no separator, say as w_ℓ, and the correct output for position i is computed in nosep, and consists of the reverse of $w_{k-1-\ell}$.

Lemma 4.9. *If $\mathcal{P}$ is a B-RASP*[pos] *program with packed outputs, then there is a B-RASP*[pos] *program with packed outputs that computes $map\text{-}duplicate \circ \mathcal{P}$.*

Proof. As in the proof of Lem. 4.8, we want to compose $\mathcal{P}$ with the program in Ex. 4.2, so we adapt Ex. 4.2 to use packed inputs. Modify $\mathcal{P}$ so that its output vector is a fresh vector z instead of out. First append the following operations to $\mathcal{P}$ (where prev, next, head, body, and tail are as in the proof of Lem. 4.8):

$$\begin{aligned}
\texttt{ptail}(i) &= (\texttt{tail}(i-1) \text{ if } i > 0 \text{ else `}\varepsilon\text{'}) \cdot \texttt{head}(i) \\
\texttt{nhead}(i) &= \texttt{tail}(i) \cdot (\texttt{head}(i+1) \text{ if } i < n-1 \text{ else `}\varepsilon\text{'}) \\
\texttt{dbody}(i) &= map\text{-}duplicate(\texttt{body}(i)) \\
\texttt{sep}(i) &= \texttt{ptail}(i) \cdot \texttt{dbody}(i) \cdot \texttt{nhead}(i)
\end{aligned}$$

This computes in the vector sep the correct outputs for those positions i that have a separator in the input symbol $\mathtt{z}(i)$. The symbol is parsed into $head$, $body$, and $tail$ as xyz, and the correct output is the concatenation of the $tail$ of the preceding symbol, the strings x, $map\text{-}duplicate(y)$, z, and the $head$ of the the following symbol. Note that *map-duplicate* is applied only to strings of bounded length.

The outputs for positions i where $\mathtt{z}(i)$ does not contain a separator are computed in the vector nosep and combined with the values in sep to produce the final output by the following operations.

$$
\begin{aligned}
\mathtt{nowrap}(i) &= \mathtt{pos}(i) + (\mathtt{pos}(i) - \mathtt{prev}(i)) \\
\mathtt{wrap}(i) &= \mathtt{pos}(i) - (\mathtt{next}(i) - \mathtt{pos}(i)) + 1) \\
\mathtt{half}(i) &= (\mathtt{pos}(i) - \mathtt{prev}(i)) \leq (\mathtt{next}(i) - \mathtt{pos}(i)) \\
\mathtt{src1}(i) &= \mathtt{nowrap}(i) \text{ if } \mathtt{half}(i) \text{ else } \mathtt{wrap}(i) \\
\mathtt{src2}(i) &= \mathtt{src1}(i) + 1 \text{ if } \mathtt{src1}(i) < \mathtt{next}(i) \\
&\quad \text{else } \mathtt{prev}(i) \\
\mathtt{sym1}(i) &= \mathtt{tail}(\mathtt{src1}(i)) \text{ if } \mathtt{src1}(i) < \mathtt{next}(i) \\
&\quad \text{else } \mathtt{head}(\mathtt{src1}(i)) \\
\mathtt{sym2}(i) &= \mathtt{tail}(\mathtt{src2}(i)) \text{ if } \mathtt{src2}(i) < \mathtt{next}(i) \\
&\quad \text{else } \mathtt{head}(\mathtt{src2}(i)) \\
\mathtt{nosep}(i) &= \mathtt{sym1}(i) \cdot \mathtt{sym2}(i) \\
\mathtt{out}(i) &= \mathtt{sep}(i) \text{ if `|'} \in \mathtt{z}(i) \text{ else } \mathtt{nosep}(i)
\end{aligned}
$$

If the input symbol does not contain a separator, it is the concatenation of the symbols from sym1 and sym2, whose positions are calculated using the vectors nowrap and wrap in a manner similar to Ex. 4.2, but also including the $tail$ of the closest symbol on the left with a separator, and the $head$ of the closest symbol on the right with a separator. □

On the other hand, every operation in B-RASP[pos] is computable by a family of $\mathtt{AC}^0$ circuits, that is, a family of Boolean circuits of constant depth and polynomial size (Hao et al., 2022), which implies that any transduction computable in B-RASP[pos] is computable in $\mathtt{AC}^0$.

5 S-RASP and Average Hard Attention Transformers

5.1 Definition

We further extend B-RASP[pos] to *RASP with prefix sum* (or S-RASP) by adding a prefix sum operation.

Definition 5.1 (Prefix sum)**.** A prefix sum operation has the form

$$P(i) = \mathbf{psum}_j\, [j \leq i]\ V(j)$$

where $V(j)$ is an integer expression with $\mathrm{FV}(V(j)) \subseteq \{j\}$. It defines an integer vector $P(i)$ containing the sum of the values $V(j)$ for those positions j such that $j \leq i$. As with arithmetic operations, if the value of the prefix sum at a position is greater than $n-1$, it is replaced with $n-1$.

5.2 Padded Inputs

We defined non-length-preserving transductions for B-RASP and B-RASP[pos] by employing the convention of packed outputs. However, for S-RASP, we introduce a simpler scheme: using only symbol, not string, vectors, while assuming that the input string is followed by padding symbols #, enough to accommodate the output string.

The input vector is $a_0 a_1 \cdots a_{\ell-1} \#^{n-\ell}$, where $\ell < n$ and $a_i \in \Sigma$ for $i \in [\ell]$. The output vector, similarly, is $b_0 b_1 \cdots b_{k-1} \#^{n-k}$, where $k < n$ and $b_i \in \Gamma$ for $i \in [k]$.

With this input/output convention, padding symbols may be necessary to create enough positions to hold the output string. But in order to be able to prove closure under composition for transductions computable in B-RASP and its extensions, we allow additional padding symbols to be required. In particular, the program $\mathcal{P}$ *computes the transduction* f iff there exists a nondecreasing function q, called the *minimum vector length*, such that for every input string $w \in \Sigma^\ell$, we have $q(\ell) \geq k = |f(w)|$, and if $\mathcal{P}$ is run on $w \cdot \#^{n-\ell}$, where $n > q(\ell)$, then the output is $f(w) \cdot \#^{n-k}$. In all of the examples in this section, except *marked-square*, q is linear.

We could have used padded inputs with B-RASP programs, but it can be shown that programs would only be able to map input strings of length n to output strings of length at most $n+k$, for some constant k. Packed outputs give B-RASP the ability to define transductions with longer outputs, like string homomorphisms. However, the situation is exactly opposite with S-RASP. Packed outputs do not add any power to S-RASP, because ''unpacking'' a packed output into a vector of output symbols can be computed within S-RASP (Lem. 5.3). Moreover, packed outputs only allow transductions with linear growth, and, as we

will see, S-RASP can define transductions with superlinear growth (Ex. 2.4).

5.3 Properties

Lemma 5.2. *If $f_1 \colon \Sigma_1^* \to \Sigma_2^*$ and $f_2 \colon \Sigma_2^* \to \Sigma_3^*$ are computable in S-RASP, then their composition $f_2 \circ f_1 \colon \Sigma_1^* \to \Sigma_3^*$ is computable in S-RASP.*

Proof. Let the S-RASP program $\mathcal{P}_i$ compute the transduction f_i with minimum vector length q_i for $i = 1, 2$. Let $\mathcal{P}_3$ be the S-RASP program that consists of the operations of $\mathcal{P}_1$ followed by the operations of $\mathcal{P}_2$, where $\mathcal{P}_1$ is modified to output a fresh vector `z` (instead of `out`) and $\mathcal{P}_2$ is modified to input vector `z` (instead of `in`). We can choose a nondecreasing function q_3 such that $q_3(\ell) \geq \max(q_1(\ell), q_2(q_1(\ell)))$, so that q_3 as a minimum vector length ensures that $\mathcal{P}_3$ correctly computes $f_2 \circ f_1$. □

Lemma 5.3. *For any string homomorphism $h : \Sigma^* \to \Gamma^*$ there exists an S-RASP program to compute h, with minimum vector length $q(\ell) = K\ell$, where K is the maximum of $|h(\sigma)|$ over $\sigma \in \Sigma$.*

Proof. Number the symbols of Σ as $\sigma_0, \ldots, \sigma_{m-1}$. We use a position-wise operation to record in position i the length of $h(\texttt{in}(i))$.

$$\texttt{lens}(i) = |h(\texttt{in}(i))|$$

Then we determine the starting position of each $h(\texttt{in}(i))$ in the output.

$$\texttt{ends}(i) = \textbf{psum}_j\,[j \leq i]\ \texttt{lens}(j)$$
$$\texttt{starts}(i) = \texttt{ends}(i) - \texttt{lens}(i)$$

For $k \in [K]$, define $\texttt{sym}_k(i)$ such that if output position i is to be the k-th symbol generated from input position j, then $\texttt{sym}_k(i) = \texttt{in}(j)$:

$$\texttt{sym}_0(i) = \blacktriangleright_j\,[\top, pos(i) = \texttt{starts}(j)]\ \texttt{in}(j) : \text{`\#'}$$
$$\texttt{sym}_1(i) = \blacktriangleright_j\,[j < i, \top]\ \texttt{sym}_0(j) : \text{`\#'}$$
$$\vdots$$
$$\texttt{sym}_{K-1}(i) = \blacktriangleright_j\,[j < i, \top]\ \texttt{sym}_{K-2}(j) : \text{`\#'}$$

`in`	input	A	B	B	C	#	#
`pos`	i	0	1	2	3	4	5
`lens`	length of $h(\texttt{in}(i))$	2	0	0	3	0	0
`ends`	end of $h(\texttt{in}(i))$	2	2	2	5	5	5
`starts`	start of $h(\texttt{in}(i))$	0	2	2	2	5	5
`sym0`	mark start	A	#	C	#	#	#
`sym1`	mark start+1	#	A	#	C	#	#
`sym2`	mark start+2	#	#	A	#	C	#
`out`	output	a	a	c	c	d	#

Table 6: Example S-RASP computation for a string homomorphism. Details in Ex. 5.4.

Finally, we can define the output vector:

$$\texttt{out}(i) = \sigma_0 \text{ if } \bigvee_{\substack{a \in \Sigma, k \in [K] \\ h(a)_k = \sigma_0}} \texttt{sym}_k(i) = a$$
$$\vdots$$
$$\text{else } \sigma_{m-2} \text{ if } \bigvee_{\substack{a \in \Sigma, k \in [K] \\ h(a)_k = \sigma_{m-2}}} \texttt{sym}_k(i) = a$$
$$\text{else } \sigma_{m-1}$$

An example is in Ex. 5.4. □

5.4 Examples and Expressivity

Example 5.4 (string homomorphisms)**.** Consider the homomorphism A ↦ aa, B ↦ ε, C ↦ ccd.

$$\text{ABBC\#\#} \mapsto \text{aaccd\#}$$

$$\texttt{lens}(i) = 2 \text{ if } \texttt{in}(i) = \text{`A'}$$
$$\text{else 3 if } \texttt{in}(i) = \text{`C' else 0}$$
$$\texttt{ends}(i) = \textbf{psum}_j\,[j \leq i]\ \texttt{lens}(j)$$
$$\texttt{starts}(i) = \texttt{ends}(i) - \texttt{lens}(i)$$
$$\texttt{sym0}(i) = \blacktriangleright_j\,[\top, \texttt{pos}(i) = \texttt{starts}(j)]\ \texttt{in}(j) : \text{`\#'}$$
$$\texttt{sym1}(i) = \blacktriangleright_j\,[j < i, \top]\ \texttt{sym0}(j) : \text{`\#'}$$
$$\texttt{sym2}(i) = \blacktriangleright_j\,[j < i, \top]\ \texttt{sym1}(j) : \text{`\#'}$$
$$\texttt{out}(i) = \text{`a' if } \texttt{sym0}(i) = \text{`A'} \vee \texttt{sym1}(i) = \text{`A'}$$
$$\text{else `c' if } \texttt{sym0}(i) = \text{`C'} \vee \texttt{sym1}(i) = \text{`C'}$$
$$\text{else `d' if } \texttt{sym2}(i) = \text{`C' else `\#'}$$

An example run is in Table 6.

Example 5.5 (*marked-square*)**.** Make $|w|$ many copies of w separated by bars, with successively longer prefixes marked (here by uppercasing).

$$\text{abaa} \mapsto \text{| Abaa | ABaa | ABAa | ABAA |}$$

This transduction is aperiodic polyregular but not regular. It has greater than linear growth, and is therefore not computable in B-RASP[pos] with

in	input	a	a	b	#	#	#	#	#	#	#	#	#	#	#
pos	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
len	input length	3	3	3	3	3	3	3	3	3	3	3	3	3	3
inpos	i in input?	$\top$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
glen	ℓ_g = group length ($i > 0$)	0	4	4	4	4	4	4	4	4	4	4	4	4	4
mglen	$\min(n-1, i\ell_g)$	0	4	8	12	13	13	13	13	13	13	13	13	13	13
starts	starts of groups	0	4	8	0	0	0	0	0	0	0	0	0	0	0
isstart	is i in starts?	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
isstartnum	isstart numeric	1	0	0	0	1	0	0	0	1	0	0	0	0	0
gnumber	group number	1	1	1	1	2	2	2	2	3	3	3	3	3	3
gstart	start of i's group	0	0	0	0	4	4	4	4	8	8	8	8	8	8
src	$i - \texttt{gstart}(i) - 1$	0	0	1	2	0	0	1	2	0	0	1	2	3	4
ismarked	is i marked?	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$
y1	letters moved	a	a	a	b	a	a	a	b	a	a	a	b	#	#
y2	mark and add initial '|''s	|	A	a	b	|	A	A	b	|	A	A	B	#	#
lastbar	i for last '|'	12	12	12	12	12	12	12	12	12	12	12	12	12	12
out	output	|	A	a	b	|	A	A	b	|	A	A	B	|	#

Table 7: Example S-RASP computation for *marked-square*. Details in Ex. 5.5.

packed outputs. But it can be computed by the following S-RASP program.

$$
\begin{aligned}
\texttt{len}(i) &= \blacktriangleleft_j\,[\top, \texttt{in}(j) = \text{`\#'}]\ \texttt{pos}(j)\\
\texttt{inpos}(i) &= \texttt{pos}(i) < \texttt{len}(i)\\
\texttt{glen}(i) &= \texttt{len}(i) + 1 \text{ if } \texttt{pos}(i) > 0 \text{ else } 0\\
\texttt{mglen}(i) &= \textbf{psum}_j\,[j \le i]\ \texttt{glen}(j)\\
\texttt{starts}(i) &= \texttt{mglen}(i) \text{ if } \texttt{inpos}(i) \text{ else } 0\\
\texttt{isstart}(i) &= \blacktriangleleft_j\,[\top, \texttt{pos}(i) = \texttt{starts}(j)]\ \top : \bot\\
\texttt{isstartnum}(i) &= 1 \text{ if } \texttt{isstart}(i) \text{ else } 0\\
\texttt{gnumber}(i) &= \textbf{psum}_j\,[j \le i]\ \texttt{isstartnum}(j)\\
\texttt{gstart}(i) &= \blacktriangleright_j\,[j \le i, \texttt{isstart}(j)]\ \texttt{pos}(j)\\
\texttt{src}(i) &= \texttt{pos}(i) - \texttt{gstart}(i) - 1\\
\texttt{ismarked}(i) &= \texttt{src}(i) < \texttt{gnumber}(i)\\
\texttt{y1}(i) &= \texttt{in}(\texttt{src}(i))\\
\texttt{y2}(i) &= \text{`|'} \text{ if } \texttt{isstart}(i)\\
&\quad\ \text{else } \mathit{mark}(\texttt{y1}(i)) \text{ if } \texttt{ismarked}(i)\\
&\quad\ \text{else } \texttt{y1}(i)\\
\texttt{lastbar}(i) &= \blacktriangleleft_j\,[\top, \texttt{y2}(j) = \text{`\#'}]\ \texttt{pos}(j)\\
\texttt{out}(i) &= \text{`|'} \text{ if } \texttt{pos}(i) = \texttt{lastbar}(i) \text{ else } \texttt{y2}(i)
\end{aligned}
$$

The finite function mark changes the input symbol to uppercase. An example run is in Table 7.

Theorem 5.6. *Every aperiodic polyregular transduction is computable in S-RASP.*

Proof. By Def. 2.7, any aperiodic polyregular transduction can be decomposed into a composition of aperiodic regular transductions and *marked-square*. All aperiodic regular transductions are computable in B-RASP[pos] (Thm. 4.7), and their packed outputs can be unpacked in S-RASP (Lem. 5.3), so all aperiodic regular transductions are computable in S-RASP. Further, *marked-square* is computable in S-RASP (Ex. 5.5) and S-RASP is closed under composition (Lem. 5.2). Thus, S-RASP can compute all aperiodic polyregular transductions. □

in	input	b	b	a	b	b	a	b	a	#	#
pos	i	0	1	2	3	4	5	6	7	8	9
pa	count-left(a)	0	0	1	1	1	2	2	3	3	3
na	count(a)	3	3	3	3	3	3	3	3	3	3
pb	count-left(b)	1	2	2	3	4	4	5	5	5	5
nb	count(b)	5	5	5	5	5	5	5	5	5	5
out	output	b	b	b	b	b	b	b	b	#	#

Table 8: Example S-RASP computation for *majority-rules*. Details in Ex. 5.7.

Example 5.7 (*majority-rules*)**.** If there are at least as many a's as b's in the input, change all inputs to a; otherwise change inputs to b (Bakovic, 2000).

$$\text{abbabbba\#\#} \mapsto \text{bbbbbbbb\#\#}$$

The number of a's and the number of b's are computed and broadcast to every position. Each position determines whether its output is a, b or #.

$$
\begin{aligned}
\texttt{pa}(i) &= \textbf{psum}_j\,[j \le i]\ (1 \text{ if } \texttt{in}(j) = \text{`a'} \text{ else } 0)\\
\texttt{na}(i) &= \blacktriangleright_j\,[\top, \top]\ \texttt{pa}(j)\\
\texttt{pb}(i) &= \textbf{psum}_j\,[j \le i]\ (1 \text{ if } \texttt{in}(j) = \text{`b'} \text{ else } 0)\\
\texttt{nb}(i) &= \blacktriangleright_j\,[\top, \top]\ \texttt{pb}(j)\\
\texttt{out}(i) &= \text{\#} \text{ if } \texttt{in}(i) = \text{\#}\\
&\quad\ \text{else `a' if } \texttt{na}(i) \ge \texttt{nb}(i) \text{ else `b'}
\end{aligned}
$$

An example run is in Table 8.

Proposition 5.8. *The transduction majority-rules is neither polyregular nor computable in B-RASP*[pos].

Proof. Polyregular transductions preserve regular languages under inverse (Bojańczyk, 2018, Thm. 1.7). The preimage of the regular language a^* under *majority-rules* is $M = \{w \mid w$ contains more a's than b's$\}$, which is not regular, so *majority-rules* is not polyregular.

A circuit family computing *majority-rules* can be modified to decide M, which is not in $\mathtt{AC}^0$ (Furst et al., 1984). Thus the *majority-rules* transduction is not computable in B-RASP[pos]. □

Example 5.9 (*count-mod-m*)**.** Let m be a positive integer and define the transduction *count-mod-m* to map any input sequence $a_0 a_1 \cdots a_{n-1}$ to the sequence $b_0 b_1 \cdots b_{n-1}$ where $b_i = (\sum_0^i a_j) \bmod m$. This transduction is rational but not aperiodic; it is a generalization of the parity problem, which has been discussed at length elsewhere (Hahn, 2020; Chiang and Cholak, 2022).

Proposition 5.10. *For any m, S-RASP can compute the transduction count-mod-m.*

Proof. We just give the case of $m = 3$, which is easily generalized. The vector `residues` contains the residues of positions modulo 3 computed by the program in Prop. 4.6. Define the finite function $\mathit{fmod3}(x, y) = (x + 2y) \bmod 3$ for $x, y \in [3]$.

$$
\begin{aligned}
\mathtt{ones}(i) &= 1 \text{ if } \mathtt{in}(i) = 1 \text{ else } 0 \\
\mathtt{ps1}(i) &= \mathbf{psum}_j\, [j \le i]\ \mathtt{ones}(j) \\
\mathtt{ps1m3}(i) &= \mathtt{residues}(\mathtt{ps1}(i)) \\
\mathtt{twos}(i) &= 1 \text{ if } \mathtt{in}(i) = 2 \text{ else } 0 \\
\mathtt{ps2}(i) &= \mathbf{psum}_j\, [j \le i]\ \mathtt{twos}(j) \\
\mathtt{ps2m3}(i) &= \mathtt{residues}(\mathtt{ps2}(\mathtt{i})) \\
\mathtt{out}(i) &= \mathit{fmod3}(\mathtt{ps1m3}(i), \mathtt{ps2m3}(i))
\end{aligned}
$$

□

On the other hand, because prefix sum can be simulated by a family of $\mathtt{TC}^0$ circuits (threshold circuits of constant depth and polynomial size), any transduction computable in S-RASP is in $\mathtt{TC}^0$.

5.5 Average-hard Attention Transformers

We prove the following connection between S-RASP programs and average hard attention transformers in Appendix B.

Theorem 5.11. *Any transduction computable by an S-RASP program is computable by a masked average-hard attention transformer encoder with a position encoding of i/n, $(i/n)^2$, and $1/(i+2)$.*

One consequence is the following result relating unique-hard and average-hard attention:

Corollary 5.12. *Any transduction computable by a masked unique-hard attention transformer encoder can be computed by a masked average-hard attention transformer encoder with a position encoding of i/n, $(i/n)^2$, and $1/(i+2)$.*

6 Conclusions

This is, to our knowledge, the first formal study of transformers for sequence-to-sequence transductions, using variants of RASP to connect classes of transformers to classes of transductions. We showed that unique-hard attention transformers and B-RASP compute precisely the class of aperiodic rational transductions; B-RASP[pos] strictly contains all aperiodic regular transductions; and average-hard attention transformers and S-RASP strictly contain all aperiodic polyregular transductions. Our finding that B-RASP[pos] and S-RASP can compute transductions outside the corresponding aperiodic class in the transduction hierarchy raises the question of fully characterizing their expressivity, a promising future research direction.

Acknowledgments

We thank Mikołaj Bojańczyk, Michaël Cadilhac, Lê Thành Dũng (Tito) Nguyễn, and the anonymous reviewers for their very helpful advice.

References

Eric Bakovic. 2000. *Harmony, Dominance, and Control*. Ph.D. thesis, Rutgers, The State University of New Jersey. https://doi.org/10.7282/T3TQ60BJ

Pablo Barceló, Alexander Kozachinskiy, Anthony Widjaja Lin, and Vladimir Podolskii. 2024. Logical languages accepted by transformer encoders with hard attention. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*.

Mikołaj Bojańczyk. 2018. Polyregular functions. arXiv:1810.08760.

Mikołaj Bojańczyk. 2022. Transducers of polynomial growth. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–27. https://doi.org/10.1145/3531130.3533326

Mikołaj Bojańczyk and Rafal Stefański. 2020. Single-use automata and transducers for infinite alphabets. In *Proceedings of the 47th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 168 of *LIPIcs*, pages 113:1–113:14. https://doi.org/10.4230/LIPIcs.ICALP.2020.113

Olivier Carton and Luc Dartois. 2015. Aperiodic two-way transducers and FO-transductions. In *24th EACSL Annual Conference on Computer Science Logic (CSL)*, volume 41 of *LIPIcs*, pages 160–174. https://doi.org/10.4230/LIPIcs.CSL.2015.160

David Chiang and Peter Cholak. 2022. Overcoming a theoretical limitation of self-attention. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 7654–7664. https://doi.org/10.18653/v1/2022.acl-long.527

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023. Neural networks and the Chomsky hierarchy. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR)*.

Emmanuel Filiot, Olivier Gauwin, and Nathan Lhote. 2016. First-order definability of rational transductions: An algebraic approach. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 387–396. https://doi.org/10.1145/2933575.2934520

Merrick Furst, James B. Saxe, and Michael Sipser. 1984. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17:13–27. https://doi.org/10.1007/BF01744431

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171. https://doi.org/10.1162/tacl_a_00306

Yiding Hao, Dana Angluin, and Robert Frank. 2022. Formal language recognition by hard attention Transformers: Perspectives from circuit complexity. *Transactions of the Association for Computational Linguistics*, 10:800–810. https://doi.org/10.1162/tacl_a_00490

R. McNaughton and S. Papert. 1971. *Counter-free Automata*. M.I.T. Press Research Monographs. M.I.T. Press.

William Merrill and Ashish Sabharwal. 2024. The expressive power of transformers with chain of thought. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*.

William Merrill, Ashish Sabharwal, and Noah A. Smith. 2022. Saturated transformers are constant-depth threshold circuits. *Transactions of the Association for Computational Linguistics*, 10:843–856. https://doi.org/10.1162/tacl_a_00493

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Lê Thành Dũng Nguyễn. 2021. Two-way transducers with planar behaviours are aperiodic. Presentation slides.

Lê Thành Dũng Nguyễn, Camille Noûs, and Cécilia Pradic. 2023. Two-way automata and transducers with planar behaviours are aperiodic. arXiv:2307.11057.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. 2021. Attention is Turing-complete. *Journal of Machine Learning Research*, 22:75:1–75:35.

Cécilia Pradic and Lê Thành Dũng Nguyễn. 2020. Implicit automata in typed λ-calculi I: aperiodicity in a non-commutative logic. In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*. Full version. https://doi.org/10.4230/LIPIcs.ICALP.2020.135

Brian Roark and Richard Sproat. 2007. *Computational Approaches to Morphology and Syntax*. Oxford University Press.

Marcel Paul Schützenberger. 1965. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194.

https://doi.org/10.1016/S0019-9958(65)90108-7

Lena Strobl, William Merrill, Gail Weiss, David Chiang, and Dana Angluin. 2024. What formal languages can transformers express? A survey. *Transactions of the Association for Computational Linguistics*, 12:543–561. https://doi.org/10.1162/tacl_a_00663

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, and Jason Wei. 2023. Challenging BIG-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051. https://doi.org/10.18653/v1/2023.findings-acl.824

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11080–11090.

Andy Yang, David Chiang, and Dana Angluin. 2024. Masked hard-attention transformers recognize exactly the star-free languages. In *Advances in Neural Information Processing 37 (NeurIPS)*.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. 2021. Self-attention networks can process bounded hierarchical languages. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, pages 3770–3785. https://doi.org/10.18653/v1/2021.acl-long.292

Appendices

In the following appendices, we prove Thm. 5.11. Appendix A reviews the definition of average-hard attention transformers. Appendix B contains our main proof, while Appendix C contains another construction using a different position embedding. Appendix D compares some features of our simulation with other simulations.

A Average Hard Attention Transformers

We recall the definition of a transformer encoder with average-hard attention, also known as saturated attention (Yao et al., 2021; Hao et al., 2022; Barceló et al., 2024). Let $d > 0$ and $n \geq 0$. An *activation sequence* is a sequence of n vectors in $\mathbb{R}^d$, one for each string position. The positions are numbered $-1, 0, 1, \ldots, n-1$. Position -1, called the *default position*, does not hold an input symbol and will be explained below. A transformer encoder is the composition of a constant number (independent of n) of layers, which of which maps an activation sequence $u_{-1}, \ldots, u_{n-1}$ to an activation sequence $u'_{-1}, \ldots, u'_{n-1}$.

There are two types of layers: (1) position-wise and (2) average hard attention. A *position-wise layer* computes a function $u'_i = u_i + f(u_i)$ for all positions i, where f is a position-wise two-layer feed-forward network (FFN) with ReLU activations. An *average hard attention layer* is specified by three linear transformations $Q, K, V \colon \mathbb{R}^d \to \mathbb{R}^d$. The dot product $S(i,j) = \langle Qu_i, Ku_j \rangle$ is the *attention score* from position i to position j. For each position i, let M_i be the set of positions j that maximize $S(i,j)$. Then $u'_i = u_i + (\sum_{j \in M_i} Vu_j)/|M_i|$. An average hard attention layer may be *masked* using *strict* or *non-strict future masking*, in which for each position i, only positions $j < i$ or $j \leq i$ (respectively) are considered in the attention calculation. With strict future masking, the default position has nowhere to attend to, so the result is $u'_{-1} = u_{-1}$.

B Simulating S-RASP

B.1 Overview of the Simulation

To define the computation of a transduction by a transformer, we need to specify how the input and output strings are represented in the initial and final activation sequences. If the input string is $w = a_0 a_1 \cdots a_{\ell-1}$, we let $a_i = \#$ for $i = \ell, \ldots, n-1$.

Let $\Sigma \cup \{\#\} = \{\sigma_0, \ldots, \sigma_{k-1}\}$ be totally ordered. The first k coordinates of each activation vector hold the one-hot encoding of a_i (or the

zero vector at the default position). The representation of the output string is analogous, using the alphabet $\Gamma \cup \{\#\}$.

Five more coordinates are designated to hold the *position encoding* (PE) and quantities computed from it. Descriptive names for these coordinates of the activation vector at position i are as follows.

$$\begin{array}{rc} 0 & \left[\begin{array}{c} \mathbb{I}[a_i = \sigma_0] \\ \vdots \\ \mathbb{I}[a_i = \sigma_{k-1}] \\ i/n \\ (i/n)^2 \\ 1/(i+2) \\ \mathbb{I}[i = -1] \\ \mathbb{I}[i = 0] \end{array}\right] \\ \vdots & \\ k-1 & \\ pos & \\ posq & \\ posi & \\ default & \\ zero & \end{array}$$

where $\mathbb{I}[\cdot]$ is 1 if the argument is true, 0 otherwise. In the simulation, i/n is used for sum and difference, i/n and $(i/n)^2$ are used for equality comparison, and i/n, $(i/n)^2$ and $1/(i+2)$ are used for the prefix sum operation. We note that the last two coordinates above can be computed from i/n.

We turn to how S-RASP programs may be simulated. Vectors of Boolean, symbol, and integer values in an S-RASP program are represented in one or more coordinates of an activation sequence in the transformer. Each operation of an S-RASP program computes a new vector of values, and is simulated by one or more transformer encoder layers which compute new values in one or more coordinates of the activation sequence. Assume that $\mathcal{P}_f$ is an S-RASP program computing a transduction $f : \Sigma^* \to \Gamma^*$ with minimum vector length $q(\ell)$, and that $n > q(\ell)$.

B.2 Representing S-RASP Vectors

Vectors of Booleans, symbols, and integers in the program $\mathcal{P}_f$ are represented in the activation sequence of the transformer as follows.

Each Boolean vector $v_0, v_1, \ldots, v_{n-1}$ in $\mathcal{P}_f$ is represented by one coordinate r of the transformer activation sequence $u_{-1}, u_0, \ldots, u_{n-1}$, where for each $i \in [n]$, $u_i[r] = 0$ if $v_i = \bot$ and $u_i[r] = 1$ if $v_i = \top$. For the default position, $u_{-1}[r] = 0$.

Let $\Delta = \{\delta_0, \delta_1, \ldots, \delta_k\}$ denote the finite set of all symbols that appear in any symbol vector in $\mathcal{P}_f$. Each symbol vector $v_0, v_1, \ldots, v_{n-1}$ in $\mathcal{P}_f$ is represented by $|\Delta|$ coordinates $r_0, r_1, \ldots, r_{k-1}$, which hold a one-hot representation of v_i (or the zero vector at the default position).

Each integer vector $v_0, v_1, \ldots, v_{n-1}$ in the program is represented by a specified coordinate r in the transformer activation sequence, where for each $i \in [n]$, $u_i[r] = v_i/n$. In the PE, the value of $u_{-1}[pos]$ is $-1/n$, but for other integer vectors we have $u_{-1}[r] = 0$. We note that all of the representing values are less than or equal to 1.

B.3 Table Lookup

A key property of S-RASP is that every integer value computed in the program must be equal to some position index $i \in [n]$. We use this property to implement a table lookup operation.

Lemma B.1. *For any integers x, q, let*

$$f_q(x) = 2qx - x^2.$$

Then:

1. *$f_q(x)$ is uniquely maximized at $x = q$;*
2. *if $x \neq q$, then $f_q(q) - f_q(x) \geq 1$.*

Proof. This is a generalized version of a technique by Barceló et al. (2024). It can easily be shown by looking at the first and second derivatives of f, and by comparing $f_q(q)$ with $f_q(q-1)$ and $f_q(q+1)$. □

Lemma B.2. *Fix an activation sequence $u_{-1}, \ldots, u_{n-1}$ and coordinates r, s, t such that $u_i[r] = k_i/n$, where each $k_i \in [n]$. Then there is an average-hard attention layer that computes $u'_{-1}, \ldots, u'_{n-1}$, where $u'_i[t] = u_{k_i}[s]$ and the other coordinates stay the same.*

Proof. Consider an attention layer with no mask and the following attention score:

$$\begin{aligned} S(i,j) &= 2u_i[r]u_j[pos] - u_j[posq] \\ &= \frac{2k_i j - j^2}{n^2} \end{aligned}$$

which is a bilinear form in u_i and u_j, and (by Lem. B.1) is uniquely maximized when $j = k_i$. The value is $u_j[s]$, which is stored in coordinate t of the output activation sequence. □

We remark that if $k_i \geq n$, the unique maximizing value of $S(i,j)$ for $j \in [-1, n-1]$ is $j = n-1$, so the attention layer in the proof above returns the value v_{n-1} for such positions i.

B.4 Simulating S-RASP Operations

For each operation below, let $u_{-1}, \ldots, u_{n-1}$ be the input activation sequence, and let $u'_{-1}, \ldots, u'_{n-1}$ be the output activation sequence. If k, v_1, v_2, b, and t are S-RASP vectors, we also write k, v_1, v_2, b, and t, respectively, for the coordinates representing them in the transformer.

B.4.1 Position-wise Operations

Position-wise Boolean operations on Boolean vectors can be simulated exactly by position-wise FFNs, as shown by Yang et al. (2024). Position-wise operations on symbol values reduce to Boolean operations on Boolean values.

To simulate *addition of two integer vectors*, $t(i) = v_1(i) + v_2(i)$, we first use a FFN to compute $k/n = \max(0, u_i[v_1] + u_i[v_2])$. The result may exceed $(n-1)/n$, so we use table lookup (Lem. B.2) to map k/n to $u_k[pos]$; this sets values larger than $(n-1)/n$ to $(n-1)/n$. The result is stored in $u'_i[t]$. Subtraction is similar, with ReLU ensuring the result is non-negative.

For *position-wise comparison of integer vectors* $t(i) = v_1(i) \leq v_2(i)$, we use a FFN to compute $k/n = \max(0, u_i[v_1] - u_i[v_2])$. We use table lookup to map k/n to $u_k[zero]$, which is 1 if $u_i[v_1] - u_i[v_2] \leq 0$, and 0 otherwise. The other comparison operators are similar.

For the *position-wise operation* $t(i) = v_1(i)$ if $b(i)$ else $v_2(i)$: If v_1 and v_2 are both Boolean vectors or both symbol vectors, this can be reduced to position-wise Boolean operations. If v_1 and v_2 are integer vectors, we use a FFN to compute

$$\begin{aligned} u'_i[t] = &\max(0, u_i[v_1] + u_i[b] - 1) \\ &+ \max(0, u_i[v_2] - u_i[b]). \end{aligned}$$

Thus if $u_i[b] = 1$ then $u'_i[t] = u_i[v_1]$ and $u'_i[t] = 0$, and if $u_i[b] = 0$ then $u'_i[t] = 0$ and $u'_i[t] = u_i[v_2]$.

B.4.2 Prefix Sum

Next, we turn to the *prefix sum* operation, $t(i) = \textbf{psum}_j\, [j \leq i]\ k(j)$. Assume that $u_i[k] = k(i)/n$, where each $k(i)$ is an integer in $[n]$ and $k(-1) = 0$. Let $p_i \geq 0$ be the sum of $k(-1), k(0), \ldots, k(i)$ and let $p'_i = \min(n-1, p_i)$, which is the sequence of values to be computed and stored in coordinate t.

The first attention layer uses non-strict future masked average hard attention with $S(i,j) = 0$, and the value is $u_j[k]$. The resulting activation sequence has the following values in coordinate s:

$$\frac{0}{n}, \frac{p_0}{2n}, \frac{p_1}{3n}, \ldots, \frac{p_{n-1}}{(n+1)n}. \tag{1}$$

Each value is smaller than the desired value by a factor of $(i+2)$; to remove this factor, we use a second attention layer. Let $v_{-1}, v_0, \ldots, v_{n-1}$ denote the activation sequence after the first layer. We use an average hard attention layer with no mask and the following attention score:

$$\begin{aligned} S(i,j) &= 2v_i[s]v_j[pos] - v_i[posi]v_j[posq] \\ &= \frac{2p_i j - j^2}{(i+2)n^2} \end{aligned}$$

which is a bilinear form in v_i and v_j, and (by Lem. B.1) is uniquely maximized when $j = p_i$. As in the remark after Lemma B.2, if $p_i \geq n$, the maximizing $j \in [n]$ is $j = n-1$. The value is $v_j[pos] = j/n = p'_{i-1}/n$ for $i > 0$ (and 0 if $i = 0$), which is assigned to coordinate t, and the other coordinates are unchanged.

B.4.3 Leftmost and Rightmost Attention

The operations

$$\begin{aligned} t(i) &= \blacktriangleleft_j\, [M(i,j), S(i,j)]\ \ V(j) : D(i) \\ t(i) &= \blacktriangleright_j\, [M(i,j), S(i,j)]\ \ V(j) : D(i) \end{aligned}$$

require that if there is any position $j \in [n]$ that makes the attention predicate $S(i,j)$ true, then the unique minimum or maximum such j is selected, but if there is no satisfying position $j \in [n]$, then the default value is used. Attention may be past or future masked, either strictly or non-strictly. We assume that transformers have only (strict or non-strict) future masking; to simulate past masking, we can calculate the index $(n-1)/n - i/n$, use Lem. B.2 to reverse the relevant vectors, and then use future masking.

The attention score $S(i,j)$ is either a Boolean combination of Boolean vectors, or an equality comparison between two integer vectors. In either case, we compute an attention score

$$S'(i,j) = S_{\text{base}}(i,j) \pm S_{\text{tie}}(i,j) \ +\ S_{\text{def}}(i)\ default(j)$$

where $S_{\text{base}}(i,j)$ is maximized for positions where $S(i,j)$ is true, $+S_{\text{tie}}$ breaks ties to right, $-S_{\text{tie}}$ to the left, and S_{def} handles the default case.

Maximization. If $S(i,j)$ is a Boolean combination of Boolean vectors, to ensure that attention from any position to the default position is 0, we let $S_{\text{base}}(i,j) = \neg default(j) \wedge S(i,j)$. This may be computed by dot product attention, as described by Yang et al. (2024).

For the special case where $S(i,j)$ is an equality comparison of integer vectors, say $v_1(i) = v_2(j)$: We first use a lookup operation (Lem. B.2) with the $posq$ entry of the PE to get the squares of the values in v_2 in coordinate t. Let $u_{-1}, u_0, \ldots, u_{n-1}$ be the resulting activation sequence. We then use an average hard attention operation with the attention score function

$$\begin{aligned} S_{\text{base}}(i,j) &= 2u_i[v_1]u_j[v_2] - u_j[t] \\ &= \frac{2v_1(i)v_2(j) - v_2(j)^2}{n^2} \end{aligned}$$

which is a bilinear form in u_i and u_j, and is maximized (by Lem. B.1) when $v_2(j) = v_1(i)$.

Breaking Ties. If $S_{\text{base}}(i,j)$ were used with average hard attention, then the activation values would be averaged for all the satisfying j. To ensure that the maximum satisfying position j has a unique maximum score, we break ties by adding or subtracting $S_{\text{tie}}(i,j)$. We must ensure that the values added or subtracted are smaller than the minimum difference between the values for satisfying and non-satisfying positions.

For a Boolean combination of Boolean vectors, let $S_{\text{tie}}(i,j) = \max(0, j/(2n))$. Then under rightmost attention, the rightmost satisfying j has the highest attention score, which is at least 1, while every non-satisfying j has an attention score less than $1/2$. Similarly for leftmost attention.

For an equality comparison $v_1(i) = v_2(j)$, the difference between the maximum score attained and any other score is at least $(1/n)^2$ by Lem. B.1. So if we add or subtract values less than $(1/n)^2$, no non-equality score can exceed an equality score. This can be achieved by letting $S_{\text{tie}}(i,j) = j/(2n^3)$. This is computable using dot product attention because j/n is in the PE for j and $(1/n)^2$ is in the PE for 1 and can be initially broadcast to all positions.

Default Values. The term S_{def} needs to give the default position an attention score strictly between the possible scores for satisfying and non-satisfying j.

For a Boolean combination of Boolean vectors, the maximum non-satisfying score is less than $1/2$ and the minimum satisfying score is at least 1, so if we let $S_{\text{def}}(i) = 3/4$, then the default position has an attention score of $3/4$, so it will be the unique maximum in case there are no satisfying positions.

For an equality comparison of integer vectors, the maximum non-satisfying score is less than $(v_1(i)/n)^2 - (1/2)(1/n^2)$, and the minimum satisfying score is at least $(v_1(i)/n)^2$, so $S_{\text{def}}(i) = (v_1(i)/n)^2 - (1/4)(1/n^2)$ is strictly between these values. The value of $(v_1(i)/n)^2$ may be obtained at position i using Lem. B.2 with index $v_1(i)/n$ and the *posq* coordinate of the PE.

Thus, the default position is selected when there is no $j \in [n]$ satisfying the attention predicate; it remains to supply the default value. We use an attention layer with the attention score S' given above and value $V(j) = \begin{bmatrix} default(j) \\ V(j) \end{bmatrix}$. Let j_i be the position that i attends to. Then we use a position-wise if/else operation that returns (the simulation of) $D(i)$ if $default(j_i) = 1$ and $V(j_i)$ otherwise. This concludes the proof of Theorem 5.11.

C An Alternate Position Encoding

The simulation of S-RASP via average hard attention transformers in Thm. 5.11 relies on three kinds of position encoding: i/n, $(i/n)^2$, and $1/(i+2)$. In this section, we present evidence for the following.

Conjecture C.1. *Any transduction computable by an S-RASP program is computable by a masked average-hard attention transformer encoder with a position encoding of i/n.*

First, $1/(i+2)$ can be computed from i/n.

Proposition C.2. *A transformer with positions $i \in \{-1, 0, \ldots, n\}$ and position encoding i/n can compute $1/(i+2)$ at all positions i.*

Proof. As observed by Merrill and Sabharwal (2024), a transformer can use the i/n encoding to uniquely identify the first position (-1) and compute $1/(i+2)$ by using non-strict future masked attention with value 1 at that position and 0 elsewhere $(0, \ldots, n-1)$. □

In Thm. C.4 we show that the position encoding i/n and $1/(i+2)^2$ suffices for the simulation of S-RASP by a masked average hard attention transformer. Though it's unclear whether a transformer with position encoding i/n can compute $1/(i+2)^2$, we note the following.

Proposition C.3. *A transformer with positions $i \in \{-1, 0, \ldots, n\}$ and position encoding i/n can compute $1/((i+2)^2 - 1)$ at positions $i < n$.*

Proof. By Proposition C.2, the transformer can compute $1/(i+2)$ at position i. It can then compute $1/((i+2)^2 - 1)$ simply as the difference between the $1/(i+2)$ values at the two neighbors of position i:

$$\frac{1}{(i+2)^2 - 1} = \frac{1}{2}\left(\frac{1}{i+1} - \frac{1}{i+3}\right).$$

□

Theorem C.4. *Any transduction computable by an S-RASP program is computable by a masked average-hard attention transformer encoder with a position encoding of i/n and $1/(i+2)^2$.*

Proof Sketch. The proof of this theorem closely follows the argument presented earlier for Thm. 5.11, except for the position encoding used. We will show how each use of $(i/n)^2$ in that original argument can be replaced with an equivalent use of $1/(i+2)^2$, which we assume to be stored in a coordinate called $posiq$ (for ''inverse quadratic''). We also assume that $1/(i+2)$ is available by Prop. C.2.

The original proof uses the quadratic maximization in Lem. B.1, which we replace with:

Lemma C.5. *For any integers x, q, let*

$$f_q(x) = \frac{2}{n(x+2)} - \frac{q+2}{n(x+2)^2}. \qquad (2)$$

Then $f_q(x)$ is uniquely maximized over values of $x \geq -1$ when $x = q$.

Proof. Consider the derivative, $-2/n(j+2)^2 + 2(q+2)/n(j+2)^3$, whose only real-valued root is $j = q$. Furthermore, the derivative is positive for $j < q$ and negative for $j > q$. □

This score is a bilinear form that can be computed via average hard attention using query $\langle 2/n, -q/n - 2/n\rangle$ at position i and key $\langle 1/(j+2), 1/(j+2)^2\rangle$ at position j. In all our applications of this new score, we will ensure that q/n is available at position i. The $2/n$ term can also be computed at position i by attending uniformly (without masking) with value 2 at the first position and 0 elsewhere. There are three uses of $posq$ in the original argument that we have to modify.

The first use is in the proof of Lem. B.2, for the basic lookup operation. Instead of using an attention score of $2u_i[r]u_j[pos] - u_j[posq]$, we use Eq. (2) with $q = k_i$ (recall that $u_i[r] = k_i/n$):

$$S(i,j) = \frac{2}{n}\, u_j[posi] - \left(u_i[r] + \frac{2}{n}\right) u_j[posiq].$$

By Lem. C.5, $S(i,j)$ is maximized over j uniquely when $j = k_i$, as needed in the proof of Lem. B.2.

The next use of $posq$ in the original argument is for the prefix sum (Appendix B.4.2). As before, we compute $p_i/((i+2)n)$ and store it as $v_i[s]$, with $v_i[0]$ being $0/n$. Instead of the original attention score of $2v_i[s]v_j[pos] - v_j[posq]v_i[posi]$, we use:

$$S(i,j) = \frac{2}{n}\, v_j[posi]v_i[posi] - \left(v_i[s] + \frac{2}{n(i+2)}\right) v_j[posiq]$$

where the $2/(n(i+2))$ term is computed at position i by using future masked attention with a score of $2/n$ (computed earlier) at the first position and 0 elsewhere. This gives an attention score of:

$$S(i,j) = \frac{2}{n(j+2)(i+2)} - \frac{p_i + 2}{n(j+2)^2(i+2)} \qquad (3)$$

which, by Lem. C.5, is uniquely maximized when $j = p_i$. This allows us to retrieve value $j/n = p_i/n$ from position j, as needed in the proof in Appendix B.4.2.

The third and final use of $posq$ is in the simulation of *leftmost and rightmost attention* and its *default values* (Appendix B.4.3). Specifically, suppose the attention predicate in S-RASP is an equality comparison of two integer vectors, say $v_1(i)$ and $v_2(j)$, represented as $u_i[r] = k_i/n$ and $u_j[s] = k_j/n$, respectively. In this case, we first use two lookup operations (Lem. B.2, updated for the inverse square position embedding) with the $posi$ and $posiq$ entries of the position embedding to copy inverses and inverse squares of the values in v_2 to coordinates t and z of the activation. As in the original proof, let $u_{-1}, u_0, \ldots, u_{n-1}$ denote the resulting activation sequence. We thus have $u_i[t] = 1/(k_i+2)$ and $u_i[z] = 1/(k_i+2)^2$. We then use the attention score function

$$S(i,j) = \frac{2}{n}\, u_j[t] - \left(u_i[r] + \frac{2}{n}\right) \cdot u_j[z] \qquad (4)$$

a bilinear combination of u_i and u_j equivalent to:

$$S(i,j) = \frac{2}{n(k_j+2)} - \frac{k_i+2}{n(k_j+2)^2}. \quad (5)$$

By Lem. C.5, $S(i,j)$ is uniquely maximized over values of $j \geq -1$ when $k_j = k_i$.

As in the original argument, there may be multiple matches and we thus need to break ties in favor of the leftmost or rightmost match. To this end, we observe that $S(i,j) = 1/(n(k_i+2))$ when $k_j = k_i$, and compare this to the maximum value of $S(i,j)$ for $k_j \neq k_i$, which is $(k_i+4)/(n(k_i+3)^2)$, attained at $k_j = k_i+1$. Thus, the gap between the attention score when $k_j = k_i$ versus the maximum possible when $k_j \neq k_i$ is $1/(n(k_i+2)(k_i+3)^2)$. Since $k_i < n$, this is lower bounded by $1/(n(n+1)(n+2)^2) > g(n)$ where $g(n) = 1/(20n^4)$. As in the original argument, if we add or subtract from $S(i,j)$ values less than $g(n)$, no non-equality score can exceed the corresponding equality score. We achieve this by adding or subtracting the tie-breaking term $g(n)j/(2n) = j/(40n^5)$; the reason for using this specific tie-breaking term will become apparent when we discuss default values below. This term is computable by first computing $1/n^4$ at position i and then using dot product attention with j/n in the position encoding of j. In order to compute $1/n^4$, we can attend uniformly with only the first position having value $1/n$ (the rest having value 0) to obtain $1/n^2$, and repeat this process twice more to obtain $1/n^4$. This finishes the updates needed for the simulation of leftmost and rightmost attention.

We address *default values* in a similar way as in the original proof. When it involves an equality comparison of integer vectors and rightmost attention, we observe that with the tie-breaking term $g(n)j/(2n)$ discussed above, the gap between the matching attention score $1/(n(k_i+2))$ and the maximum non-matching attention score for rightmost attention is at least $g(n)/2$. Hence, a default position value of $1/(n(k_i+2)) - g(n)/4$ is strictly between these two values. Further, this default position value is computable at position i by the same arguments as above. We treat default values with leftmost attention analogously. □

D Comparison with Other Simulations

In the prefix sum operation (1), the result at position i is $s(i)/(i+1)$, where $s(i)$ is the prefix sum of $v(i)$. The fact that the denominator of this expression varies with position is an obstacle to comparing or adding the values $s(i)$ and $s(j)$ at two different positions i and j. This problem is addressed by Yao et al. (2021) and Merrill and Sabharwal (2024) using a non-standard layer normalization operation to produce a vector representation of the quantities, which allows them to be compared for equality using dot product attention. Pérez et al. (2021) include $1/(i+1)$ in their position embedding to enable the comparison; however, they compute attention scores as $-|\langle Qu_i, Ku_j\rangle|$ in place of the standard dot-product. The approach of the current paper is based on that of Barceló et al. (2024), who show how average hard attention can be used to compute the prefix sum of a $0/1$ vector.