THINKING LIKE TRANSFORMERS

Anonymous authors
Paper under double-blind review

ABSTRACT

What is the computational model behind a transformer? Where recurrent neural networks have direct parallels in finite state machines, allowing clear discussion and thought around architecture variants or trained models, transformers have no such familiar parallel. In this paper we aim to change that, proposing a *computational model for the transformer-encoder* in the form of a programming language. We map the basic components of a transformer-encoder – attention and feed-forward computation – into the simple primitives of `select`, `aggregate` and `zipmap`, around which we form a programming language: the *Restricted Access Sequence Processing Language (*RASP*)*. We show how RASP can be used to program solutions to tasks that could conceivably be learned by a transformer, augmenting it with tools we discover in our work. In particular, we provide RASP programs for histograms, sorting, and even logical inference similar to that of Clark et al. (2020). We further use our model to relate their difficulty in terms of the number of required layers and attention heads. Finally, we see how insights gained from our abstraction might be used to explain phenomena seen in recent works.

1 INTRODUCTION

We present a *computational model for the transformer-encoder* in the form of a simple sequence-processing language. Our *Restricted Access Sequence Processing Language (*RASP*)* language captures the information flow and processing constraints posed by a Transformer (Vaswani et al., 2017).

Unlike convolution networks (which can be thought of as sequence of filters) and RNNs (state machines), the computation model of a transformer is not well understood. While some works prove that sufficiently large transformers are universal sequence processors (Yun et al., 2019), they do not show how the computation is structured.

Considering computation problems and their implementation in the RASP language allows us to "think like a transformer" while abstracting away the technical details of a neural network in favor of symbolic programs. A RASP program operates on *sequences of values* from uniform atomic types, and transforms them by composing a restricted set of sequence processors. Some processors can be used to perform *local computation* on a limited-size window over the sequence. Other processors can be used to *select* inputs for aggregation, and *aggregate* the selected items. Local processors can perform arbitrary computation over their (limited size) input. However, access to the complete sequence is available *only* through aggregate operations that reduce a stream of numbers to a scalar. The key to performing complex global computations under such a model is to arrange the local computations such that they produce the correct sequences that can then be streamed for aggregation.

Given a RASP program, we can analyze it to infer minimal number of layers and upper-bound number of heads that is requires to implement it. We show several examples of expressive programs written in the RASP language, showing how complex operations can be implemented by a transformer. Thinking in terms of the RASP model also allows us to shed light on empirical observation of transformer variants Press et al. (2020) and on concrete limitations of "efficient transformers" with restricted-attention Tay et al. (2020).

```
1 def sort(vals):
2   num_prevs = count_conditioned(
3                 (vals,indices),(vals,indices),
4                 lambda val,ind,val',ind':
5                     (val' < val) or
6                     (val' == val and ind' < ind))
7   select_sorted_val = select(indices,num_prevs,lambda i,np:i == np)
8   return aggregate(select_sorted_val,seq)
```

Figure 1: RASP program taking a sequence `vals` and returning a sequence `y` sorting the elements of `v`, e.g.: if `v`(x)=`[0,4,2]`, then `y`(x)=`[0,2,4]`. It can be expanded to accept an additional sequence `key`, such that it sorts the values of `vals` according to order of those in `key`. Note that `indices` is not provided as a parameter to `sort` - it is a special sequence in the language, available to any program.

2 THE RESTRICTED ACCESS SEQUENCE PROCESSING LANGUAGE

In this section, we present the *the Restricted Access Sequence Processing Language* (RASP). RASP assumes a machine composed of several Turing-complete processors, each of which can only run functions taking and returning a fixed number of primitive arguments, and a simple memory accessor that is controlled by these processors. The `select`, `aggregate`, and `zipmap` operations which we present will define and constrain how the processors work together to process an input sequence.

We will focus here only on the language itself, leaving the discussion of its exact relation to transformers to Section 3. We will begin with a very brief description of RASP behaviour, and dive right into a motivating example (fig. 1) before presenting a fuller explanation of RASP.

Overview A RASP program works by manipulating *sequences*, occasionally with the help of *selectors*. Sequences contain values of uniform atomic type, such as booleans, integers, floats, or strings. They are functions used for selecting elements from sequences, and are used (together with the appropriate operations) only in the process of creating new sequences. All sequences in RASP are lazily evaluated, meaning that they are not available ahead of time, but are a result of a computation.

The Base Sequences Every program in RASP begins from the same set of base sequences and manipulates them using a small number of core operations to create new ones, , up to its output sequence. These base sequences are **`indices`**, **`length`**, and **`tokens`**), evaluated on input $x_1, x_2, ..., x_n$ as their names suggest: $(0, 1, ..., n-1)$, $(n, n, ..., n)$ (of length n), and $(x_1, x_2, ..., x_n)$, respectively.

Combining Sequences Sequences can be combined in an 'elementwise' manner, such that the value of the resulting sequence at each position i is a function of the values in the combined sequences at position i (similar to a `map` operation), or have positions 'mixed' in more complicated ways using *selectors*, which are functions $f : \mathbb{N} \times \mathbb{N} \to \{\text{True}, \text{False}\}$ whose sole purpose is to guide the combination of existing sequences into new ones.

We present the basic ingredients of RASP using an example. Figure 1 shows a simple RASP function for sorting a sequence of values. It accepts an input sequence `vals` and uses the *base* sequence `indices`, that is available to any RASP program, to compute its output in three operations as follows:

1. `count_conditioned` of Line 2 creates a new sequence that counts for each element of `vals` the number of "previous items" it has in `vals`, where the "previous items" are defined to be all items that have a lesser value, or equal value and lower index. Thus, `num_prevs` creates a sequence of counts, representing the target sorted position of each item.
2. `select` of line 7 creates a new *selector* which will select for each position i the corresponding position j for which `indices[`i`]` is equal to `num_prevs[`j`]`. Effectively, it will direct the elements in each position j towards their target location i.
3. Finally, `aggregate` of line 8 applies the `select_sorted_val` to the input sequence `vals`, moving each i-th element of `vals` to its calculated sorted position `num_prevs[`i`]`.

We now describe the base operations of RASP in-depth, occasionally presenting an example on the hypothetical input sequence x of length n.

- **zipmap** The zipmap operation takes a tuple of sequences and a processing function f, and applies f per-index to the values in those sequences to create a new sequence. For a simple example, y1=zipmap((indices,indices), lambda i,j:i+j) creates a sequence that always evaluates to $(0, 2, ..., 2n-2)$.
- **aggregate** The aggregate operation takes a selector s and a sequence x, and averages subsets of the values of x into a new sequence y as follows: for every index i, y[i] is assigned the average of x[j_0], x[j_1], ...x[j_k], where $j_0, j_1, ..., j_k$ are the indices j for which f(i,j) is True. If there are no such j's, y[i] is assigned the value d, an optional parameter to aggregate which has default value 0.0. Whenever $k = 1$, aggregate will not explicitly compute the division, in this case x may have uniform type other than number. For example: if f(i, j) returns True iff i is odd and j=0, y will evaluate to (d,x[0],d,...,y[$n-1$]) where y[$n-1$] is d or x[0] depending on the parity of n.
- **select** The select operation takes two tuples of sequences, me=(m1,m2,...,mk) and other=(ot1,ot2,...,otl), and a selection function f expecting k+l atomic values, and composes them to create a selector s as follows: for every two indices i, j, s(i, j) is the output of f on the i-th and j-th slice of me and other respectively, i.e., s(i, j)=f(m1[i],...,mk[i],ot1[j]...otl[j]) [1]. For a simple example, s=select((indices,),(indices,),lambda mi,oti:mi%2==1 and oti==0) will evalute to the same select we used for our example in aggregate above.
- **count_conditioned** This operation takes the same parameters me,other and f as select, but this time returns a sequence y describing the *number* of selected influencing positions j for each output position i that s=select(me,other,f) would have created. In other words, for each i, y[i]$= k$ where $j_1, ..., j_k$ is the set of positions j for which s(i, j)=True. For example, h=count_conditioned((tokens,),(tokens,),lambda a,b:a==b) returns an in-place histogram for the tokens in the input sequence: h("abaa")=$(3, 1, 3, 3)$.

This concludes the base operations of RASP – all other operations are shortcuts for combinations of the above 4, occasionally with the base sequences.

2.1 EXAMPLES

We now present some more example RASP programs, by increasing order of complexity.

Simple Examples The first and simplest example is to compute an in-place histogram for some sequence vals. This is achieved with a single application of count_conditioned: histogram=count_conditioned(vals,vals,lambda a,b:a==b).

From Length to Parity While length is provided as a primitive in the language, it can actually be achieved as a composition of the other base operations and sequences. This is done by computing full_s=select((),(),lambda :True) followed by 1/aggregate(full_s,indices,lambda i:int(i==0)) (the fraction of elements equal to 0 in indices, inverted). From length and that same full_s we can then implement count(vals,v) that takes any sequence vals and value v, using length*aggregate(full_s,vals,lambda e:e==v): the number of values v in vals. This in turn enables us to write programs like parity simply as count(tokens,1)%2==0. (This does not contradict with the findings of Hahn (2019), who found that parity is not computable in transformers *when each selector is restricted to width 1* ("hard attention").)

Reverse Using length, we can reverse a sequence seq. This is achieved by computing an up-to-one selector mapping each position to its opposite in the sequence: flip_s = select(indices,length-1-indices,lambda m,oth:m==oth) [2], and then using it to re-order the tokens of seq: reversed=aggregate(flip_s,seq).

[1] (We will never evaluate s(i,j) for i or j greater than n, though for completeness we say that in this case the value is False.)

[2] When f_i is not given, as is the case here, select1 assumes it to be the identity. Note that length-1-indices is a sequence mapping each input x of length n to $(n-1, n-2, ..., 0)$.

```
def by_frequency(vals,da):
  hist = count_conditioned(vals,vals,lambda a,b:a==b)
  num_earlier = count_conditioned(
                          (indices,vals),(indices,vals),
                          lambda iq,vq,ik,vk:(vq==vk) and (ik<iq))
  has_earlier = num_earlier > 0
  masked_hist = hist - (length * has_earlier)
  unique_vals = zipmap((vals,has_earlier),
                       lambda t,he:t if not he else da)
  return sort(unique_vals,key=-masked_hist)
```

Figure 2: RASP program sorting the unique elements of a sequence `vals` by decreasing frequency. It requires a default atom `da` of the same type as `vals`, to put in place of all otherwise-duplicated values in its result. For example, `by_frequency(tokens_str,"-")("abacca")` will return "`acb--`".

Balanced Parentheses For balanced parentheses we use `count_conditioned` twice, storing in `prev_opens` and `prev_closes` the number of previous "`(`" or "`)`" (respectively) tokens each position has, including itself. The sequence is balanced if `prev_opens-prev_closes` has no negative values, and is 0 at position `length-1`. These two qualities can be easily computed using two final `select-aggregate` pairs, and then combined with a `zipmap`.

Most Frequent Tokens In fig. 2 we show how RASP can be used to arrange for any input sequence `s` the most frequent tokens in `s`, without repetition. The solution has 3 parts: computing the histogram for all the tokens, masking it such that all but the first of each token is given a negative value, and then sorting the tokens according to the masked histogram. The presented code uses the variant of `sort` that also accepts a `key` sequence, (as described in the caption of fig. 1), and the library operation `conditioned_contains(me,other,f)` which is equal to `conditioned_count(me,other,f)>0`.

```
  def count_conditioned(me,other,f):
    other = other+(indices,)
    s_with_0 = select(me,other,lambda *a:f(*(a[:-1])) or (a[-1]==0))
    s_just_0 = select(me,other,lambda *a:f(*(a[:-1])) and (a[-1]==0))
    frac = aggregate(s_with_0,indices,lambda i:int(i==0))
    count_outside_0 = (1/frac)-1
    count_in_0 = aggregate(s_with_0,(),lambda :1,default=0)
    return count_outside_0 + count_in_0
```

Figure 3: Implementation of the operation `count_conditioned` in terms of `indices` and the other base operations, assuming that `other` is indeed a tuple of sequences (and not a single one).

Count Conditioned The operation `count_conditioned` is a powerful part of RASP, appearing in many other programs. Surprisingly, it is realisable as a composition of the other operations (and base sequence `indices`)! Understanding how this is achieved is interesting for learning how to "truly" think like a transformer, and we present the code in fig. 3. The intuition is as follows: we compute the select whose width we want to calculate twice, once such that it *also* selects the element in position 0, and once such that it *only* selects this element. We then aggregate both these values, broadcasting 1 from position 0 and 0 from everywhere else, and using default value 0. The first `aggregate` computes for each position the *inverse* of the number of selected positions (excluding 0) plus one, and the second computes whether that position would also focus on 0. A straightforward `zipmaps` then gives us the result. To further help the intuition, we present in fig. 4 the computation flow for a `histogram` calculation, when `count_conditioned` is implemented as described here.

Note Although many functions provided with RASP can be expressed in terms of `count_conditioned` – such as `count` (which counts how many elements in a sequence are equal to a given value) and `contains` (which checks if `count` is greater than 0) – they actually have more efficient implementations when analysed with respect to the number of transformer layers and heads that they translate to. These are of course the implementations provided in the RASP library.

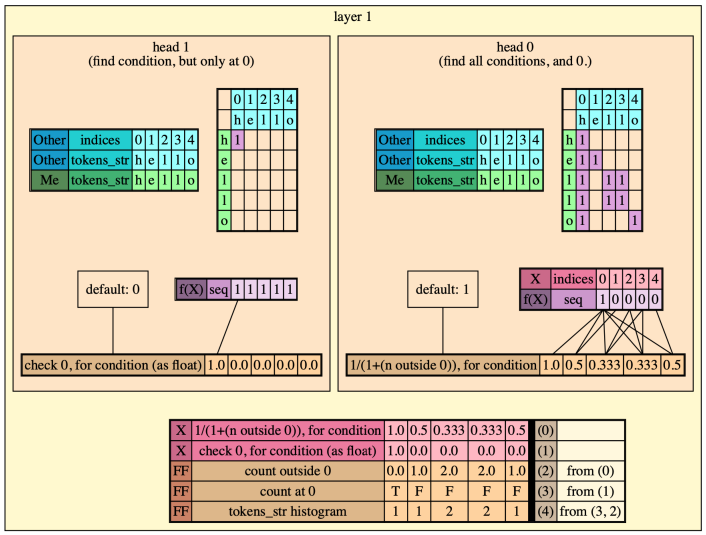

Figure 4: The computation flow for the sequence `histogram(tokens_str)` applied to input sequence "`hello`", when implemented in terms of `select`, `aggregate`, and `zipmap`.

2.2 Implementing Symbolic Reasoning a la Rover

How might a transformer implement *reasoning*, as in Clark et al. (2020)? The RASP empowers us to clearly think about such problems, and we sketch a solution direction here. We begin by significantly simplifying and reformulating the task of Clark et al. (2020) to a form that focuses on the core problem, moving away from natural language into more 'concrete' domain, and limiting the type of relations we will consider.

We simplify the reasoning task of Clark et al. 2020, by which a transformer is presented a series of relations and implications between them, ending with one final relation Q, and must recognise whether Q is implied by the previous statements or not. For example: `a1∈A1, b∈A1` $\implies$ `b∈A2`, `a1∈A2?` evaluates to `True`, whereas `a1∈A1, a1∈A2?` evaluates to `False`.[3] As in (Clark et al., 2020), we note that different inputs to this task can have different "depth": the number of 'logical hops' needed to correctly identify whether the query statement is true.

We suggest approaching this task in RASP as follows: first, mark each relation or inference 'block' in the sequence by its index (as a block), by counting for each token the number of separators before it in the index. Set aside the final block (the query), which is the set of tokens with no separators *after* it, so it is not involved in the following computations.

Then, by passing these block-indices to selectors, mark each non-query block as being either a relation or inference statement. Create four 'processing' sequences, `contents`, `not_contents`, `into`, and `outof`, and for each relation block load the element `a` into the `content` or `not_content` sequence (according to the contains-relation token) at the position of the set `A`, and similarly load `A` into the `into` or `outof` sequence at the position of `a`. This is the information known using no logical hops.

Now repeat the following for as many hops as you require your transformer to capture: first, globally, combine the contents of the four processing sequences across all sets and elements: for example, if element `a1` appears at two positions i, j and `into[`i`]` contains `A1` whereas `into[`j`]` does not, augment `into[`j`]` with `A1` as well. Then, for each inference statement block, transfer information in the processing sequences according to the contents of the block. For example, for `a∈A1` $\implies$ `a∈A2`, augment the `contents` sequence in all `A2` positions with its values in the first `A1` position, and the `not_contents` sequence in all `A1` positions with its values in the first `A2` position. Finally, share information between the `contents`/`not_contents` sequences and the `into`/`outof` sequences: for

[3] Our simplification is in moving to a predefined set of element and set tokens, and only considering "contains" relations.

every element token a, augment into and outof with all set tokens A for which it appears in their contents or not_contents. Similarly, update contents and not_contents for the set tokens.

In this way the contents of the processing sequences are augmented, such that after every new iteration d they are capable of answering all queries requiring at most d logical hops. The final output is calculated by seeking the query's element token among the provided statements, and seeing whether the query's set token is in the into or the outof sequence at that position.

Generalisation on Inference Depth A similar solution to the one we have proposed would be to make all of these logical inferences backwards from the query Q, i.e. by maintaining sequences sufficient_if_contains, sufficient_if_doesnt_contain, sufficient_if_in, sufficient_if_out that are updated by the inference statements in the opposite direction, and share values in the same way as above. If a trained transformer implements both of these solutions in parallel (for instance, to increase its robustness), this may explain the generalisation to greater query depth observed by Clark et al.

3 Relation to Transformers, and Analysis

We discuss how the RASP relates to the real transformer architecture, and how it may also be used to compare transformer variants, or analyse the 'difficulty' of a task for transformers.

Relation to Transformers While we give a detailed explanation in the appendix, intuitively, the select and aggregate operations correspond to attention heads, zipmap operations correspond to the feed forward layers, and computed sequences correspond to head or feedforward outputs. Length and conditioned contains are in fact combinations of the other primitives.

The elementwise zipmap operation is an abstraction of the feed-forward component of a transformer. The aggregate operation is a direct reflection of the end of an attention operation, where the distribution is used to gather and average temporarily computed values into a new output.

The base selector length and base operation count_conditioned are in fact combinations of the other operations and selectors, as described in section 2.1. Finally, the persistence of sequences – such that they may be accessed multiple times over the course of a RASP program – is encouraged by the existence of skip connections in the transformer architecture.

3.1 Structure Analysis (Number of Heads and Layers)

For any given RASP program, we can compute the minimal number of layers required to implement it in a transformer, and upper bound the number of heads this implementation requires.[4] This is provided its internal dimensions are wide enough to replicate the given processing functions. This analysis can give us intuition regarding the relative difficulty of different tasks for the transformer architecture, where each algorithm we find for a task gives us an upper bound on the number of layers and heads a transformer needs to solve it.

We implement such an analysis and provide a draw_comp_flow function, which automatically positions each attention head using a greedy scheduler, and displays the computation flow accordingly (see fig. 4, and others in the supplementary material).

A similar analysis can be done for different "primitive" computations in isolation, giving us intuition on the 'cost' of various common computations: how many additional heads and layers each computation adds when applied to a previously computed sequence. For example, our implementation of sort takes 2 layers, and so whenever we apply it to a computed sequence we know it will increase our program's depth by 2 from that sequence.

Algorithm for computing program depth and width As noted, the first 3 base operations of RASP – select, aggregate, and zipmap – have direct parallels in the transformer architecture, and

[4]The reason the number of heads can only be upper bounded is because it is possible that some selects in the program are equivalent, or can be combined into a single select without interfering with each other, but it is impossible to identify this statically.

so we may easily analyse the result of any RASP program to see how many layers and heads it would take to realise in an actual transformer. (For `length` and `count_conditioned`, we analyse them in terms of their deconstruction to the other operations and sequences.)

The first part of the analysis is simple: every `sequence` and `selector` is initiated with a "minimum depth" d, reflecting the earliest layer at which it can be computed, and so the minimum number of layers needed to create any given `sequence` is d. d is computed for each new `sequence` as follows:

1. The base sequences `indices` and `tokens` have $d = 0$, as they are input to the transformer rather than part of its computation.
2. Any sequence created from a `zipmap` is given d equal to the maximum d of the inputs to that `zipmap`, as it can be created immediately after the last of them in the same feed forward computation that concludes it.[5]
3. Every selector gets d equal to the maximum d of its creating sequences `X` plus 1, to reflect the fact that all of them must be calculated before it can even begin (as multi-headed attention happens only once, at the beginning of each layer).
4. Any selector created from an `aggregate` has d equal to at least that of its creating selector, and at least one plus those of the input sequences to the `aggregate` operation (as they must be passed through the attention to create the new sequences).

RASP makes it easy to access *all* sequence and selectors on which an unfinished value is dependent, and allowing us to analyse not only the depth but also the width of a given program. The *width* of the computation is a reflection of how many (unique) selectors are being used at every layer of the transformer: the number of attention heads needed to mimick that layer in a transformer. We say that a selector s is being used at some layer l if a sequence that is aggregated from s is calculated at l. This is because it is possible a selector s may have minimum depth d, but is only or also used at later layers: for instance if the sequences needed for the aggregation operation using s are not yet ready.

Similarly, a sequence does not *need* to be computed at its minimum possible depth, as it is possible it will only be needed much later. Hence there is no one analysis for a given program, and there is room for creating a scheduling algorithm that minimises the maximum width of the transformer, i.e., the maximum of the widths of all layers (useful, as transformers tend to be created with uniform width).

4 Implications for Transformer Variants

4.1 Restricted-Attention Transformers

Multiple works propose restricting the attention mechanism of transformers in order to create more efficient transformers, reducing the time complexity of each layer from $\mathrm{O}(n^2)$ to $\mathrm{O}(nlog(n))$ or even $\mathrm{O}(n)$ with respect to the input sequence length n (see Tay et al. (2020) for a survey of such approaches and their complexity). Several of these do so using *sparse attention*, in which the attention is masked using different patterns to reduce the number of locations that can interact (see for instance (Child et al., 2019; Beltagy et al., 2020; Ainslie et al., 2020; Zaheer et al., 2020; Roy et al., 2020)).

Considering these variants of transformers in terms of the RASP language, allows us to reason about the computations they can and cannot perform. In terms of RASP, these variants of transformers all impose restrictions on the `selectors`, forcing some of the n^2 index pairs (i, j) to `False`.

Figure 1 showed how to implement sorting of an input sequence with arbitrary alphabet size, comparison function, and length[6]. We now prove that RASP variants where the `selector` is restricted to $\mathrm{O}(n)$ pairs (i.e., transformer variants with sufficiently restricted attention), cannot sort.

The computation model of RASP and indeed of all transformer variants allows comparison of values between more than one sequence position only during the `select` operation, i.e., only while

[5]Except in the special case where this maximum d is 0, and the `zipmap` is not being called from within an aggregate, in which case the sequence is assigned $d = 1$. This reflects that at least one layer must be used to reach the first feed-forward computation, but also that attention may do a little processing itself before it "aggregates", using the linear translation $\mathbf{V}$.

[6]Providing sufficiently stable word and positional embeddings, a practical limitation that applies to all transformer variants.

computing the attention distribution Hence, all comparisons necessary for sorting must be applied in `select`. It follows that whenever `select` is restricted such that it compares at most $O(n)$ index pairs per head, no constant number of heads and layers will be sufficient for the model to perform sorting on arbitrary length – as sorting is known to require at least $O(nlog(n))$ comparisons.

Thus, variants of transformers in which the attention is masked to impose $O(n)$ complexity require at least $O(log(n))$ layers to sort. It also follows that they require $O(log(n))$ layers to implement `count_conditioned`, as we see in fig. 1 that `count_conditioned` can be applied to create a sequence (`num_prevs`) which is sufficient to complete a sorting operation with only $O(n)$ further operations.

4.2 SANDWICH TRANSFORMERS

Recently, Press et al. (2020) showed that reordering the attention and feed-forward sublayers of a transformer affects its ability to train on language modeling tasks. In particular, they showed that pushing feed-forward sublayers towards the bottom of a transformer weakened it, while pushing attention sublayers to the bottom and feed-forward sublayers to the top strengthened it, provided there was still some interleaving in the middle (making a *sandwich transformer*).

Thinking about such variations in terms of the base RASP operations helps us understand the observations of Press et al.. In RASP, the feed-forward and attention sublayers are of course the `zipmap` and `select-aggregate` (or **`gather`** for short) operations. Any arrangement of the sublayers into a set architecture, from the 'standard' transformer to the variations considered in (Press et al., 2020), can be seen as imposing a restriction on the number and order of RASP operations that can be chained in a RASP program. For example, a standard 8-layer transformer architecture imposes that at most 8 `gather` operations may be chained one after the other, and the width of the architecture determines the maximum number of `gather` operations that may be scheduled at each layer. Similarly, an architecture in which all feed-forward sublayers appear before the attention sublayers imposes that no `zipmap` operation may be applied to the results of any `gather` operation.

Similarly, it is clear that there is no value to repeated applications of `zipmap` before the first `gather`, explaining the first observation of Press et al. (2020). This is because `zipmap` can never increase at any position the information already available there by simply observing the current sequences (as it only locally processes information), and before the first `gather` operations there is simply not enough interesting information in each individual cell to process.

Conversely, an architecture beginning with several attention sublayers – i.e., multiple `gather` operations – will be able to gather a large amount of information into each position early in the computation, if only by simple rules. More complicated gathering rules can be realised by applying `zipmaps` to the gathered information before generating new `selectors`[7], explaining the interleaved attention/feed-forward middle section present in the discovered architecture.

5 CONCLUSIONS

We abstract the computation model of the Transformer-encoder into a simple sequence processing language, that captures the information flow and processing constraints posed by a Transformer. Considering computation problems and their implementation in the RASP language allows us to "think like a transformer" while abstracting away the technical details of a neural network in favor of symbolic programs. Moreover, we can analyze the RASP programs to infer minimal number of layers and upper-bound number of heads that is requires to implement them. We show several examples of expressive programs written in the RASP language, showing how complex operations can be implemented by a transformer. Thinking in terms of the RASP model also allows us to shed light on empirical observation of transformer variants Press et al. (2020) and on concrete limitations of "efficient transformers" with restricted-attention Tay et al. (2020).

[7]Actually, the unbounded power of the processing functions `f` that RASP allows passing into `select` and `aggregate` technically renders `zipmap` unnecessary. Still, it is easy to imagine that `zipmap` would help if these functions were more strictly limited, as they are in 'real' transformers. Certainly we as programmers prefer to preprocess sequences using `zipmap` before passing them along for our `select` processors to handle.

REFERENCES

Joshua Ainslie, Santiago Ontañón, Chris Alberti, Philip Pham, Anirudh Ravula, and Sumit Sanghai. ETC: encoding long and structured data in transformers. *CoRR*, abs/2004.08483, 2020. URL https://arxiv.org/abs/2004.08483.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. URL http://arxiv.org/abs/1904.10509.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as soft reasoners over language, 2020.

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *CoRR*, abs/1906.06755, 2019. URL http://arxiv.org/abs/1906.06755.

Ofir Press, Noah A. Smith, and Omer Levy. Improving transformer models by reordering their sublayers. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pp. 2996–3005. Association for Computational Linguistics, 2020. URL https://www.aclweb.org/anthology/2020.acl-main.270/.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers, 2020.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions?, 2019.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. *CoRR*, abs/2007.14062, 2020. URL https://arxiv.org/abs/2007.14062.

A ADDITIONAL DETAILS ABOUT RASP

A note about types in RASP Technically, there is no one 'tokens', rather 5 options: tokens_str, tokens_int, tokens_float and tokens_bool cast the input sequence to the corresponding atomic types, and tokens_asis takes the input sequence as-is. For brevity, we refer to all of these as tokens here.

RASP Sequences RASP operates exclusively on sequence- and selector-generating functions, which we refer to as *sequences* and *selectors* respectively and RASP-*functions* together. All RASP-functions take as input exactly one non-empty sequence, and when describing them and how RASP manipulates them to create new RASP-functions we will do so in terms of the sequences and selectors that they and their manipulations generate from each input sequence. When it is clear from context, we will simply refer to them as sequences and selectors, and describe them directly in terms of their outputs. For example, if we say that a RASP operation applies +1 elementwise to each value in a sequence u, we actually mean that it returns a new sequence v such that for every input x and position $0 \leq i < |x|$, v(x)[i]=u(x)[i]+1.

Note In this section we will refer to the i-th value in a sequence s as s[i], such that s[i] is in one of the atomic types. We stress however that this is only for the discussion, and not a part of the language.

Additional operations For brevity in code, the RASP also comes with the following syntactic sugar:

- Anywhere that a tuple of sequences is passed into an operation, a single sequence may be passed in as-is as well. For example, `y=zipmap((indices,),(indices,),lambda a,b:a+b)` can also be written `y=zipmap(indices,indices,lambda a,b:a+b)`.
- The `zipmap` operation is accessible through a large range of operators, covering its application to all of the base binary and unary operations on the atomic types. For example, the above `y` can equivalently be defined as `y=indices+indices`. These operators can also be mixed with constants from the atomic primitives, such that an equal `y` be obtained using `y=2*indices`.
- In the `aggregate` operation, instead of the float- or integer-sequence `x`, you may pass in a tuple of sequences `xx` (or single sequence) and a function `f`, such that `zipmap(xx,f)` creates an input sequence `x` as originally expected by `aggregate`. The order of the parameters to aggregate is `s,xx,f,d`, where `f` and `d` are optional, and running `aggregate(s,xx,f,d)` is equivalent to running `x=zipmap(xx,f)` followed by `aggregate(s,x,default=d)`.
- Anywhere that a sequence-processing function is expected, (`aggregate` and `zipmap`), it may return more than one value (provided the number of values it returns is constant). In this case, the operation will sort the output values into the same number of output sequences. For example, `y1,y2=zipmap(x,lambda v:v+1,v+2)` is equivalent to writing `y1=x+1` and then `y2=x+2`.
- Anywhere that a processing function is expected, if one is not provided, the identity function is used[8].
- The function **`select1`** which takes only a single tuple of sequences `xx` and an index-computing function `fi`, and is syntactic sugar for `select(xx,(indices,),lambda *a:fi(a[:-1])==a[-1])`. `select1` is guaranteed to return a true select satisfying the "up-to-one" property, i.e., that can be successfully paired in aggregates with an `x` that does not contain numbers.

[8]This is consistent with `aggregate` allowing you to choose whether to pass a single sequence, or a tuple of sequences and a processing function.