# Compiling to transformers

Joey Velez-Ginorio

Department of Computer & Information Science, University of Pennsylvania

## Introduction

The representations of transformers are easy to measure but hard to understand; we cannot reliably predict or control behavior from them. To address this, recent work proposed compiling high-level programs to transformers as a tool for studying these representations. By compiling programs to transformers, the algorithms driving their representations are known, and techniques for understanding them can be verified against a ground truth.

However, prior compilers make a *closed program* assumption. They cannot compile programs with free variables. As a consequence, the program must specify the entire algorithm a transformer implements, restricting applicability to hand-constructed transformers rather than ones trained by gradient descent.

We resolve that limitation with Cajal, a language whose programs compile to transformers which are subsequently trained by gradient descent. **We prove its compiler correct, and conduct experiments where we compile a conditional reverse program into the attention head of a transformer, showing that transformers can learn to use compiled programs which specify part of their behavior.** Importantly, this enables verification of interpretability techniques against partial ground truths in transformers trained by gradient descent.

## Methods

- Using programming language theory, we provide a formal specification for a compiler from Cajal to differentiable and multilinear maps (Fig. 1).
- Using the compiler, we can embed algorithms into the attention heads of transformers to influence their learning dynamics and behavior (Fig. 3).

$$[\![-]\!] : \text{Program} \to k\text{-linear map}$$

$$[\![x : \tau \vdash x : \tau]\!](\vec{x}) = \vec{x}$$

$$[\![\varnothing \vdash * : \mathsf{1}]\!](0) = 1$$

$$[\![\Delta \vdash \mathsf{inj}_1\, e : \tau_1 \oplus \tau_2]\!](\vec{\sigma}) = ([\![e]\!], 0_{[\tau_2]})$$

$$[\![\Delta \vdash \mathsf{inj}_2\, e : \tau_1 \oplus \tau_2]\!](\vec{\sigma}) = (0_{[\tau_2]}, [\![e]\!])$$

$$[\![\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2]\!](\vec{\sigma}) = ([\![e_1]\!], [\![e_2]\!])$$

$$[\![\Delta_{\mathsf{k}} \circ \Delta_{\mathsf{v}} \vdash \{e_{\mathsf{k}} : e_{\mathsf{v}}\} : \tau_{\mathsf{k}} \rightharpoonup \tau_{\mathsf{v}}]\!](\vec{\sigma}) = \vec{x}_{\mathsf{q}} \mapsto \sum_{i=1}^{n} \langle [\![e_{\mathsf{k}}]\!]_i, \vec{x}_{\mathsf{q}} \rangle \cdot [\![e_{\mathsf{v}}]\!]_i$$

$$[\![\Delta_1 \circ \Delta_2 \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2]\!](\vec{\sigma}) = [\![e_2]\!]([\![e_1]\!])$$

$$[\![\Delta_1 \circ \Delta_2 \vdash \mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1\, x_1 \Rightarrow e_1; \mathsf{inj}_2\, x_2 \Rightarrow e_2\} : \tau_2]\!](\vec{\sigma}) = [\![e_1]\!]([\![e]\!]_1) \;+\; [\![e_2]\!]([\![e]\!]_2)$$

$$[\![\Delta \vdash \pi_i e : \tau_i]\!](\vec{\sigma}) = [\![e]\!]_i$$

$$[\![\Delta_{\mathsf{kv}} \circ \Delta_{\mathsf{q}} \vdash e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}} : \tau_{\mathsf{v}}]\!](\vec{\sigma}) = [\![e_{\mathsf{kv}}]\!]([\![\mathcal{R}]\!]([\![e_{\mathsf{q}}]\!]))$$

$$[\![\Delta \vdash e_1 \sqcap e_2 : \tau]\!](\vec{\sigma}) = [\![e_1]\!] \;+\; [\![e_2]\!]$$

$$[\![\Delta \vdash e_1; e_2 : \tau]\!](\vec{\sigma}) = [\![e_1]\!] \cdot [\![e_2]\!]$$

Fig. 1. Formal specification of the compiler

- We exploit *linear type theory* to define a programming language whose programs compile to multilinear maps.
- The compiler takes as input a program and returns a multilinear map, as shown below.

$$[\![x : \mathsf{Bool}, y : \mathsf{Bool} \vdash \mathsf{if}\, x\, \mathsf{then}\, y\, \mathsf{else}\, y : \mathsf{Bool}]\!]\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right) = \begin{bmatrix} x_1 y_1 + x_2 y_1 \\ x_1 y_2 + x_2 y_2 \end{bmatrix}$$

Fig. 2. Example output of the compiler

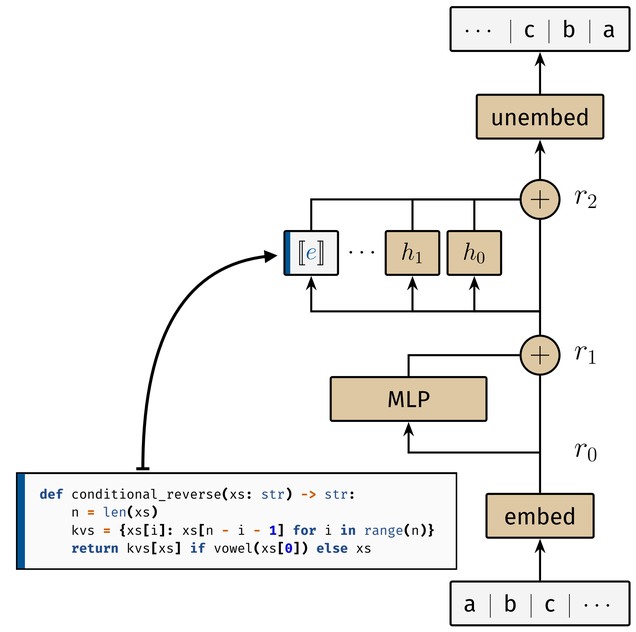


Fig. 3. Transformer with compiled attention head

## Future Work

- There are additional constraints needed on the compiler to ensure the multilinear maps we target are compatible with the constraints on functions leveraged in large-scale pretraining of LLMs.
- We believe these constraints are akin to the constraints by an *ordered type system*. We are working on a variant of Cajal with such a system, aiming to compile algorithms into LLMs with billions of parameters.
- We are additionally interested in what happens when we compile generically useful algorithms like *fold* or *scan* into an LLM. The hope is that, as with simpler algorithms, these generic algorithms act as generic inductive biases for learning.