Compiling to transformers

Joey Velez-Ginorio [1]

Abstract

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.

1. Introduction

Representations are *most* of what we can measure from a transformer. Yet leveraging these measurements to predict and control behavior remains difficult. For example, in an effort to both detect and mitigate social biases (e.g., gender bias) in Claude 3 Sonnet, Durmus et al. (2024) analyzed its representations. Their results were mixed. It is possible to detect certain social biases from representations. But intervening on these representations can impair general behavior—a result consistent with the systematic study in Da Silva et al. (2025). In short, our tools for understanding representations are not reliable enough to predict or control behavior.

To address this, Lindner et al. (2023) proposed compilers as a tool for studying representations. By compiling known algorithms into a transformer, techniques for understanding their representations can be verified against a ground truth. Consider the conditional string reversal algorithm in Fig. 1.

```
def conditional_reverse(xs: str) -> str:
    n = len(xs)
    kvs = {xs[i]: xs[n - i - 1] for i in range(n)}
    return kvs[xs] if vowel(xs[0]) else xs
```

Figure 1. Reversing a string if its first character is a vowel.

If we could compile this algorithm[1] into a transformer, we would have a rich model of the underlying representations, and can use that ground truth to verify whether techniques for understanding representations can recover it. Similar metascientific studies have been explored in other sciences: Jonas & Kording (2017) asked whether neuroscientists could understand a microprocessor, and Lazebnik (2002) whether a biologist could fix a radio.

However, compiling algorithms into transformers has so far been of limited use in verifying our understanding of representations. In part, this is because prior compilers make a *closed program* assumption (Perconti & Ahmed, 2014). They cannot compile programs with free variables. As a consequence, the program must specify the entire algorithm a transformer implements. This restricts their application to hand-constructed transformers rather than ones trained by gradient descent—which are ultimately the kind we want to understand.

Contributions This work resolves that limitation with Cajal, a language whose programs compile to transformers which can be subsequently trained by gradient descent. For example, the conditional string reversal program in Fig. 1 can be encoded in Cajal and then compiled into an attention

[1]Department of Computer Science, University of Pennsylvania, Philadelphia, PA, USA. Correspondence to: Joey Velez-Ginorio <joeyv@upenn.edu>.

Proceedings of the 43rd International Conference on Machine Learning, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

[1]The code is Python with parallel dictionary access. Passing a list of keys to a dictionary returns a list of values.

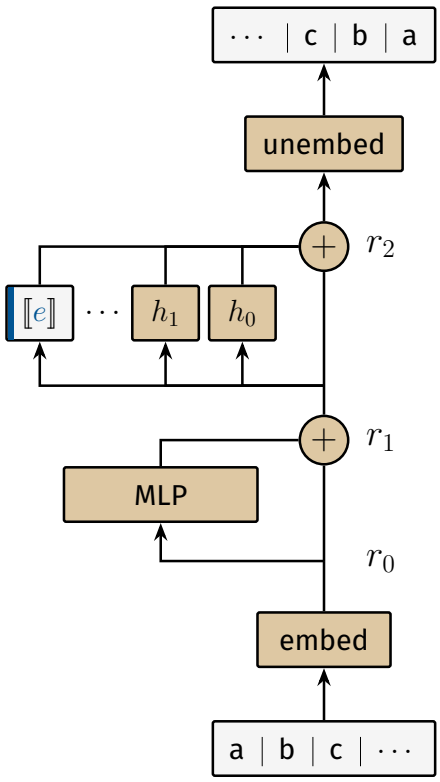


Figure 2. Compiling program $[\![e]\!]$ into an attention head of a transformer.

head of a transformer trained to solve the task, as shown in Fig. 2. More generally, our contributions are:

- We specify the syntax, evaluator, and compiler for Cajal$_{\searrow}$ programs (Section 3).
- We prove that Cajal$_{\searrow}$ programs compile correctly to transformers (Section 3).
- We implement Cajal$_{\searrow}$, programming a transformer to solve a conditional string reversal task using a compiled attention head (Section 4)
- We analyze its representations using principal components analysis (PCA), ablations, and steering to confirm it uses the compiled attention head to solve the task (Section 4).

As a result, Cajal$_{\searrow}$ enables verification of interpretability techniques against partial ground truths in transformers trained by gradient descent.

2. Background

The most important technical idea in this paper is that Cajal$_{\searrow}$ programs compile to k-linear maps. Because these k-linear maps are differentiable, we can embed them within a transformer.

Linear Algebra Linear maps model additive interactions between neurons. However, k-linear maps additionally model multiplicative interactions, while ensuring linearity in each argument. Consider the difference below, between the linear map f and the 2-linear map g.

$$f\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} 2x_1 + 3x_2 \\ 4x_1 \end{bmatrix} \quad g\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right) = \begin{bmatrix} 2x_1y_2 + 3x_1y_1 \\ 4x_2y_1 \end{bmatrix}$$

Multiplicative interactions (e.g., $2x_1y_2$) can implement more intricate behaviors, like gating. For example, the output of g is gated by its first input $\begin{bmatrix} x_1 & x_2 \end{bmatrix}$. When x_1 is 0, it turns off the activity of the first output neuron. Likewise when x_2 is 0, it turns off the activity of the second output neuron. This style of gating is how we will implement conditionals using k-linear maps, including for the conditional reverse program shown in Fig. 1.

Linear Types Most programming languages do not compile to k-linear maps. However, some languages with linear types can. These types annotate programs with information about their behavior, including constraints the behavior satisfies.[2] Consider the difference below, between the type annotation for the conditional on the left and right.

✓ $x : \mathsf{Bool}, y : \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y : \mathsf{Bool}$

✗ $x : \mathsf{Bool}, y : \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then}\ x\ \mathsf{else}\ x : \mathsf{Bool}$

Both are legitimate type annotations. They annotate the free variables x and y with types that make sense (e.g., x must be Bool for the conditional to have behavior), and also the type of value they evaluate to—when free variables are instantiated by values of their annotated type. But only the type annotation on the left is a linear type annotation: it ensures that evaluation of a program uses variables *exactly once*. It uses x when checking the condition and y when either branch executes. For the type annotation on the right, it uses x both in checking the condition and when either branch executes x, and does not use y at all.

A remarkable fact of linear types is that these syntactic constraints on permissible programs correspond to the constraints imposed on k-linear maps. This has been known for decades in programming language semantics (Bierman, 1995; Mellies, 2009), but only recently used to design programming languages whose programs compile to neural networks (Velez-Ginorio et al., 2026a;b). For example, the *linear* conditional above compiles to the following 2-linear map, which can exactly implement the behavior of the program.[3]

[2] In this paper, a program's behavior is the value produced by evaluating it.

[3] Assuming a typical one-hot encoding of booleans.

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

The goal of the following technical material is to formally establish this correspondence across all programs in Cajal$_\searrow$, and to develop a compiler which computes the correspondence for us. However, the most important intuitions are precisely those covered here in the background.

3. Defining Cajal$_\searrow$

3.1. Syntax and Evaluation

Cajal$_\searrow$ is a linearly typed programming language. Fig. 3 details its syntax, similar to a lambda calculus with sum and product types (Walker, 2005). The main differences are the absence of higher-order functions,[4] and the inclusion of dictionaries and nondeterministic choice.

Atom The simplest expression in Cajal$_\searrow$ is $*$, which introduces the unit type 1. To use $*$, we use sequencing. For example, $*; *$ sequences two $*$ together, and just returns the second. More generally, for an expression like $e_1; e_2$, sequencing evaluates e_1 and e_2, then returns the value of e_2. There is a formal specification of this behavior in Fig. 4, which defines how expressions evaluate. [5]

Character On its own $*$ is not very exciting. However, $*$ becomes powerful when paired with other expressions. With injections, $*$ can encode characters in an alphabet, and are how we introduce values of sum type. For example, the following injections introduce values of type $1 \oplus 1 \oplus 1 \oplus 1$, the type of a four letter alphabet Char. For convenience, we use n-ary forms of Cajal$_\searrow$ expressions, which are encodable using their binary form.

$$\mathsf{a} \triangleq \mathsf{inj}_0 * \qquad \mathsf{b} \triangleq \mathsf{inj}_1 * \qquad \mathsf{c} \triangleq \mathsf{inj}_2 * \qquad \mathsf{d} \triangleq \mathsf{inj}_3 *$$

To use expressions of sum type we pattern match, specifying what to do when we match against each character. For example, the following pattern match inspects x, returning a if the it is a vowel, and otherwise returning d.

$$\mathsf{isVowel} \quad \triangleq \quad \mathsf{case}\, x\, \mathsf{of} \begin{cases} \mathsf{inj}_0 x_0 \Rightarrow x; \mathsf{inj}_0 x_0 \\ \mathsf{inj}_1 x_1 \Rightarrow x; \mathsf{inj}_3 x_0 \\ \mathsf{inj}_2 x_2 \Rightarrow x; \mathsf{inj}_3 x_0 \\ \mathsf{inj}_3 x_3 \Rightarrow x; \mathsf{inj}_3 x_0 \end{cases}$$

Sequence Tuples introduce expressions of product type, which can encode sequences. For example, $(\mathsf{a}, \mathsf{b}, \mathsf{c})$ is a sequence of type $\mathsf{Char} \times \mathsf{Char} \times \mathsf{Char}$. To use these product types, we use projections to witness their elements. For example, $\pi_0(\mathsf{a}, \mathsf{b}, \mathsf{c})$ returns a.

Dictionary Dictionaries introduce expressions of dictionary type, which are sequences of keys and values. For example, $\{(\mathsf{a}, \mathsf{a}, \mathsf{b}) : (\mathsf{a}, \mathsf{b}, \mathsf{c})\}$ is a dictionary of type $\mathsf{Char} \rightharpoonup \mathsf{Char}$, comprised of three key-value pairs. To use this dictionary, we have a generalized lookup operator, which matches a query against keys in the dictionary, returning a nondeterministic choice between all matches. Since a maps both to a and b, querying the dictionary with a introduces nondeterminism. Interestingly, dictionaries compile to linear attention (Katharopoulos et al., 2020). They are an

[4] Adding higher-order functions is possible, but not not necessary for the applications shown in this paper.

[5] It is an inductively defined relation (Pierce, 2002). These are essentially implications. If the statement above the line holds, you may conclude the statement below the line.

Syntax

e	$:=$ $x \in \mathsf{Var}$	variable
	$\mid *$	unit
	$\mid (e_1, e_2)$	tuple
	$\mid \mathsf{inj}_i\, e$	injection
	$\mid \{e_\mathsf{k} : e_\mathsf{v}\}$	dictionary
	$\mid e_1; e_2$	sequencing
	$\mid \pi_i\, e$	projection
	$\mid \mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}$	case analysis
	$\mid e_1 \sqcap e_2$	nondeterminism
	$\mid e_1(e_2)_\mathcal{R}$	lookup
	$\mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$	let binding

τ	$:=$ 1	unit
	$\mid \tau_1 \times \tau_2$	product
	$\mid \tau_1 \oplus \tau_2$	sum
	$\mid \tau_1 \rightharpoonup \tau_2$	dictionary
Δ	$:=$ $\varnothing$	empty
	$\mid \Delta, x : \tau$	binder

Figure 3. Syntax of Cajal$_\searrow$ expressions e, types τ, and contexts Δ.

Evaluating

$$e \Downarrow v \iff \text{expression } e \text{ evaluates to } v$$

$$\frac{e \Downarrow v}{\mathsf{inj}_i\, e \Downarrow \mathsf{inj}_i\, v} \qquad \frac{e_\mathsf{k} \Downarrow v_\mathsf{k} \quad e_\mathsf{v} \Downarrow v_\mathsf{v}}{\{e_\mathsf{k} : e_\mathsf{v}\} \Downarrow \{v_\mathsf{k} : v_\mathsf{v}\}} \qquad \frac{e \Downarrow (e_1, e_2) \quad e_i \Downarrow v_i}{\pi_i\, e \Downarrow v_i} \qquad \frac{e_i \Downarrow v_i}{e_1 \sqcap e_2 \Downarrow v_i}$$

$$\frac{e_1 \Downarrow * \quad e_2 \Downarrow v_2}{e_1; e_2 \Downarrow v_2} \qquad \frac{e_1 \Downarrow v_1 \quad \{x \mapsto v_1\}(e_2) \Downarrow v_2}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Downarrow v_2} \qquad \frac{e \Downarrow \mathsf{inj}_i\, v \quad \{x_i \mapsto v\}(e_i) \Downarrow v_i}{\mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1\, x_1 \Rightarrow e_1; \mathsf{inj}_2\, x_2 \Rightarrow e_2\} \Downarrow v_i}$$

$$\frac{e_\mathsf{kv} \Downarrow \{v_\mathsf{k} : v_\mathsf{v}\} \quad e_\mathsf{q} \Downarrow v_\mathsf{q} \quad \mathcal{V} = \{(v_\mathsf{v})_i \mid (v_\mathsf{q}, (v_\mathsf{k})_i) \in \mathcal{R}\} \quad \bigsqcap_{\mathcal{V}} v \Downarrow v_\mathsf{r}}{e_\mathsf{kv}(e_\mathsf{q})_\mathcal{R} \Downarrow v_\mathsf{r}}$$

Figure 4. Evaluating Cajal$_{\searrow}$ expressions

abstract model of attention, from a programming language theoretic view.

In general, any expression in Cajal$_{\searrow}$ with a valid type annotation evaluates to some value. In other words, it has behavior. This is critical to prove, since what it will mean to correctly compile Cajal$_{\searrow}$ programs is to preserve their behavior as k-linear maps.

Theorem 1: (Programs have behavior)

If $\Delta \vdash e : \tau$ and $\Delta \vdash \sigma$, then $\exists v, \sigma(e) \Downarrow v$

Proof sketch. The argument is by induction on $\Delta \vdash e : \tau$, the type annotation of programs. Appendix A.4 contails a detailed proof. In short, the argument considers every way to construct a program $\Delta \vdash e : \tau$, and in each case shows how substituting its free variables with values $\sigma(e)$ and then evaluating the resulting program will always yield a value $\sigma(e) \Downarrow v$.

3.2. Compiling

To lift the closed program assumption, a compiler must work on programs with free variables. These programs specify only a part of the algorithm a system implements. The eventual behavior of those free variables is not determined by the programmer, but rather the target system—in a process known as *linking* (Cardelli, 1997). In this work, linking amounts to connecting the compiled k-linear maps within a neural network.

The compiler in Fig. 5 is a recursive function over programs. In general, it compiles a program with k free variables into a k-linear map. The domain and range of this map is determined by the type annotations. For example, a program $x{:}\mathsf{Char}, x{:}\mathsf{Char} \vdash e : \mathsf{Char}$ compiles to a 2-linear map $f : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$, with $[\![\mathsf{Char}]\!] = \mathbb{R}^4$. To illustrate how the compiler works, we will compile a few examples. There are additional examples in Appendix A.4.

Atom Because there is no way for $*$ to depend on a variable, it compiles to a constant 0-linear map, always producing 1. It is the simplest linear map the compiler produces. However, sequencing can depend on free variables, and help illustrate the simplest 2-linear map.

$$\begin{aligned}[\![x{:}\mathsf{1}, y{:}\mathsf{1} \vdash x; y : \mathsf{1}]\!](x, y) &= [\![x{:}\mathsf{1} \vdash x : \mathsf{1}]\!](x) \cdot [\![y{:}\mathsf{1} \vdash y : \mathsf{1}]\!](y) \\ &= xy\end{aligned}$$

Character The encoding of characters from before compile to one-hot encodings. Since they lack free variables, they compile to constant 0-linear maps.

$$\begin{aligned}[\![\varnothing \vdash \mathsf{a} : \mathsf{Char}]\!](0) &= [\![\varnothing \vdash \mathsf{inj}_0 * : \mathsf{1} \oplus \mathsf{1} \oplus \mathsf{1} \oplus \mathsf{1}]\!](0) \\ &= (1, 0_{[\![\mathsf{1}\oplus\mathsf{1}\oplus\mathsf{1}]\!]}) \\ &= \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}^\mathsf{T}\end{aligned}$$

With pattern matching, we can compile computation which conditions on character identity. In general, pattern matching across n cases compiles to a linear mixture of n experts, gated by the expression we are matching against (Jacobs et al., 1991). For example, isVowel compiles to a 1-linear map, returning a linear mixture of 4 experts.[6]

$$\begin{aligned}[\![\mathsf{isVowel}]\!]\left(\begin{bmatrix}x_0\\x_1\\x_2\\x_3\end{bmatrix}\right) &= [\![x; \mathsf{inj}_0 *]\!](x_0) + \sum_{i=1}^{3} [\![x; \mathsf{inj}_3 *]\!](x_i) \\ &= x_0 \cdot \begin{bmatrix}1\\0\\0\\0\end{bmatrix} + \sum_{i=1}^{3} x_i \cdot \begin{bmatrix}0\\0\\0\\1\end{bmatrix}\end{aligned}$$

[6] When clear from context, we omit the full type annotation when compiling.

Compiling

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 5. Compiling Cajal$_{\lrcorner}$ programs

Sequence Compiling a sequence is just concatenating the compiled entries of the sequence. Projection amounts to projecting out the i-th piece of that concatenated vector.

$$\begin{aligned}[\![\pi_1(\mathsf{a},\mathsf{b}) : \mathsf{Char}]\!] &= [\![(\mathsf{a},\mathsf{b}) : \mathsf{Char} \times \mathsf{Char}]\!]_1 \\ &= \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}_1^{\mathsf{T}} \\ &= \begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix}^{\mathsf{T}}\end{aligned}$$

Dictionary We model attention using dictionaries. These dictionaries compile to linear attention with an identity kernel (Katharopoulos et al., 2020), which is essentially a sum of outer products between keys and values. Lookup amounts to the application of a compiled query against the linear attention map.

$$[\![\{(\mathsf{a},\mathsf{a},\mathsf{b}) : (\mathsf{a},\mathsf{b},\mathsf{c})\}]\!] = [\![\mathsf{a}]\!] \otimes [\![\mathsf{a}]\!] + [\![\mathsf{b}]\!] \otimes [\![\mathsf{a}]\!] + [\![\mathsf{c}]\!] \otimes [\![\mathsf{b}]\!]$$

$$\simeq \begin{array}{c} \\ \mathsf{a} \\ \mathsf{b} \\ \mathsf{c} \\ \mathsf{d} \end{array} \begin{array}{c} \begin{array}{cccc}\mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{d}\end{array} \\ \begin{bmatrix}0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0\end{bmatrix}\end{array}$$

We know this compiled dictionary is *correct* because it mirrors the behavior of the original dictionary. A lookup on b returns c, and looking up $[\![\mathsf{b}]\!]$ in the compiled dictionary returns $[\![\mathsf{c}]\!]$. The key theoretical result in this paper proves that this sense of correctness holds for all programs in Cajal$_{\lrcorner}$.

Theorem 2: (Compiler preserves program behavior)

If $\Delta \vdash e : \tau$, $\Delta \vdash \sigma$, and $V = \{(D, v) \mid \sigma(e) \Downarrow_D v\}$, then $[\![e]\!]([\![\sigma]\!]) = \sum_{(D,v)\in V} [\![v]\!]$

Proof sketch. The argument is again by induction on $\Delta \vdash e : \tau$, the type annotation of programs. Appendix A.4 contails a detailed proof. In short, the argument considers every way to construct a program $\Delta \vdash e : \tau$, and in each case shows that the compiled program $[\![e]\!]$ mirrors the behavior V of the original program. Because programs have nondeterministic behavior, V captures the possible nondeterministic values. Beyond this, most cases rely on the following lemma.

Lemma 1: (Programs compile to k-linear maps)

If $\Delta, x{:}\tau_1 \vdash e : \tau_2$, then $[\![e]\!](\vec{\sigma}, \alpha_1 \cdot \vec{x_1} + \alpha_2 \cdot \vec{x_2}) = \alpha_1 \cdot [\![e]\!](\vec{\sigma}, \vec{x_1}) + \alpha_2 \cdot [\![e]\!](\vec{\sigma}, \vec{x_2})$

This lemma formally establishes the correspondence between programs and k-linear maps we have discussed throughout this paper. It is also the most important technical idea in this paper—it further shows the promise of linear types for compiling discrete algorithms into various kinds of neural networks (Velez-Ginorio et al., 2026a;b).

4. Experiments

Lifting the closed program assumption to compile programs with free variables introduces a potential complication. There is no guarantee a neural network will use the compiled k-linear map for its behavior. The goal of these experiments is to establish that transformers can learn to use these compiled k-linear maps. Our primary case study trains transformers to implement the conditional reverse algorithm from Fig. 1, using one-layer transformers structured like Fig. 2. Appendix B contains additional details and experiments, replicating our findings across three other tasks. **In summary, these findings show that transformers can learn to use the compiled k-linear maps.**

Models We train four one-layer causal transformers (Radford et al., 2018) on sequences of the form input|output, where | is a delimiter separating the input from the correct output. The input is a 15-character string sampled uniformly from lowercase letters in the alphabet. If the first character in input is a vowel, output is the reversed string; otherwise, output is equal to input. Training uses teacher-forced next-token prediction with cross-entropy loss applied to tokens after the delimiter. The four transformers share the same base architecture and training procedure, differing only in whether an additional compiled attention head is present and, if so, which program is compiled into it.

- Model I is our baseline model. It is a one-layer causal transformer with four attention heads, using learned token and position embeddings—essentially Fig. 1 without the compiled attention head. The I stands for indirect, as it is not *directly* programmed with Cajal.
- Model F is our primary model. It is Model I but with an additional attention head compiled from an encoding of the conditional reverse algorithm in Fig. 1. The F stands for full.
- Model P is an ablation model. It is Model F but with the additional attention head compiled from an encoding of a reverse algorithm. It is an ablation in the sense that it compiles a simpler algorithm. The P stands for partial.
- Model T is an ablation model. It is Model F but where the compiled attention head is randomized. It is an ablation in the sense that it keeps the signature of the compiled attention head, but randomizes the compiled k-linear map. The T stands for type-preserving.

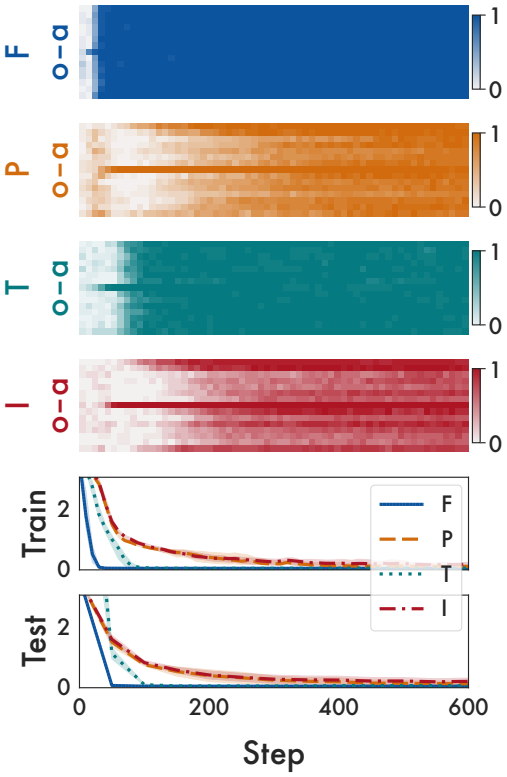


Figure 6. Behavioral dynamics.

4.1. Does compiling change learning?

Each model eventually learns to implement conditional reverse, but there are differences in how. Fig. 6 summarizes their early training dynamics. The heatmaps measure across 20 seeds the behavior of each model on the input: onmlkjihgfedcba. The y-axis tracks across seeds whether a model outputs a in its first position, b in its second position, and so forth—the correct output after reversal.

Because input strings are drawn uniformly from lowercase letters in the alphabet, the majority of the data do not contain a vowel in the first position. Therefore, a decent strategy for minimizing loss is to just learn identity, and not worry about reversal. This is essentially what Model I does. Its initial dip in loss comes from learning identity, and then the rest of learning is spent conditionally reversing for successively more vowels in the first position. Model P is similar. However, Model F never transitions into an overuse of identity,

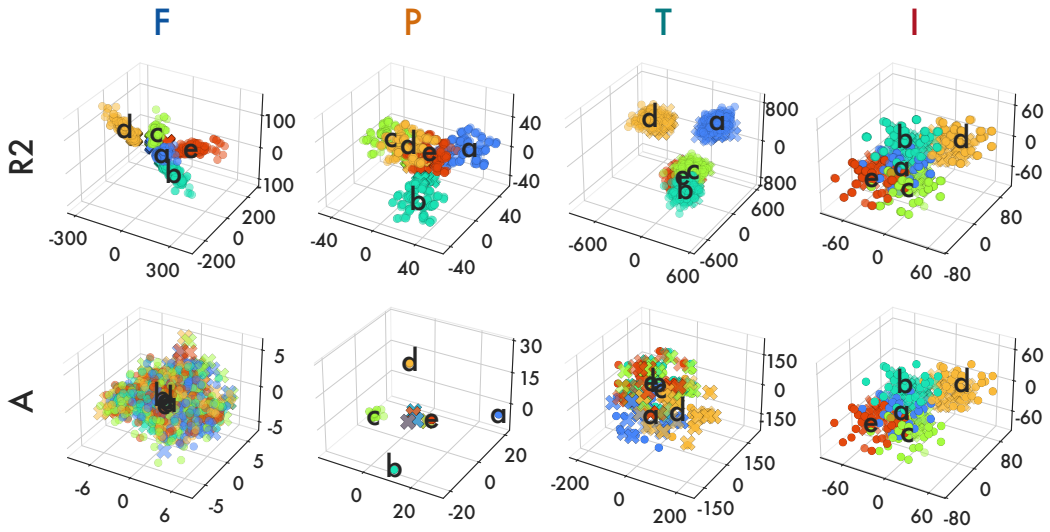


Figure 7. PCA on Representations

and instead outright learns the task the fastest across models, suggesting that the compiled attention head may be driving its behavior. Model T is similar.

4.2. Does compiling change representations?

The learning dynamics suggest Model F is using the compiled attention head to quickly learn the conditional reverse task. To check, we can observe whether there is task-relevant structure in the representations proceeding the compiled attention head versus the non-compiled attention heads. Fig. 7 displays the representations of each model projected onto their first three principal components, extracted using PCA. Row R2 depicts the residual stream after the attention layer, and A the outputs of the non-compiled attention heads. For clarity, only representations from position 16 are shown, and on a fragment of the alphabet. The points are labeled by target token identity, and use an X marker when the first token in the input sequence is a vowel. Crucially, there appears to be no task-relevant structure produced by non-compiled attention heads A for Model F.

4.3. Does compiling change the necessary representations?

To verify whether the compiled attention head is necessary for behavior, we resample ablate, for each relevant model, the representations produced by the compiled attention head R and non-compiled attention heads A. Resampling ablation replaces representations with represention generated by another input (Chan et al., 2022). Results are shown in Fig. 8, where accuracy is next-token prediction across all seeds. The ablations show Model F behaves essentially the same when ablating non-compiled attention heads, but other models rely on them to varying degrees. Interestingly,

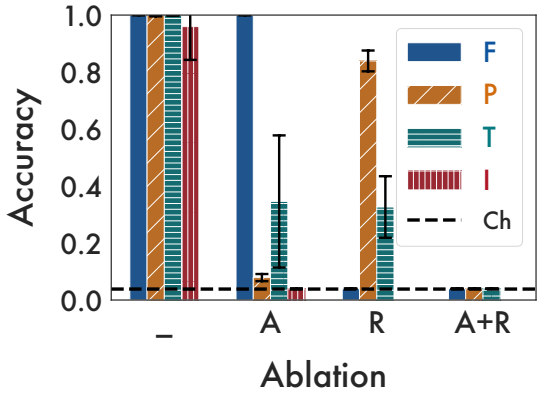


Figure 8. Resample Ablations

Model P, is fairly robust to ablating its compiled attention head, which contains a reversal algorithm. This shows that a transformer does not necessarily use any compiled program, even if intuitively relevant to the task.

4.4. Does compiling change the controllable representations?

An interesting feature of Model F is that we have a rich understanding of the compiled attention head, including what coordinates of which representations are *privileged*, whether they denote a human-intelligible concept (Elhage et al., 2023). As a final test of whether the transformer is using Model F as intended, we test whether gradient descent has preserved these privileged bases. The Cajal program compiled into Model F represents whether a character is a vowel in its first coordinate, and all other coordinates represent non-vowelness. By scaling these coordinates we should be able to reliably induce reversal (Only Rev) or identity (Only Id)—no matter if the first character is a vowel

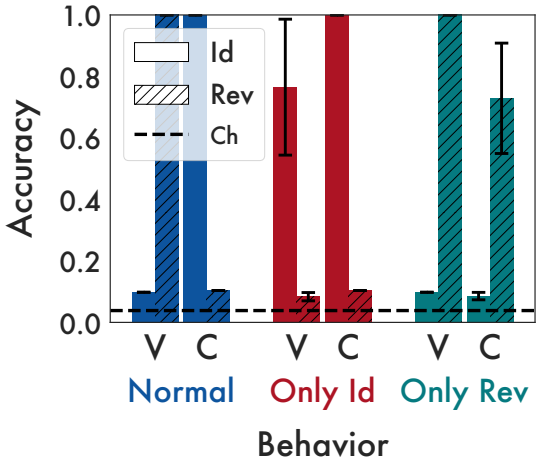


Figure 9. Steering Privileged Bases

V or a consonant C. Fig. 9 suggests these privileged bases are preserved.

5. Related Works

Cajal̀ was inspired by research in both formal language theory and programming language theory. Below we highlight their key insights, while also taking the opportunity to clarify where our work differs.

Formal Language Theory Weiss et al. (2021) introduced the programming language RASP as a model of transformers, and subsequent work built compilers for RASP-like languages, showing how to implement discrete algorithms as transformers (Lindner et al., 2023; Shaw et al., 2024). These papers directly inspired our efforts. They showed programming languages could serve as abstractions for understanding transformers.

However, the theory underpinning RASP is formal language theory; the proofs and theorems concerning RASP-like languages focus on the class of formal languages they recognize (Strobl et al., 2024). There are no formal specifications for compiling RASP-like programs with free variables to transformers. As a result, some principles underlying the design of previous compilers are difficult to understand or fix (e.g., closed program assumption).

Programming Language Theory There is a rich theory for specifying and proving the correctness of compilers (Patterson & Ahmed, 2019). This theory drives discovery of new programming languages which compile to various target architectures (e.g., quantum computers, GPUs, or x86) (Leroy et al., 2016; Hovgaard et al., 2018; Selinger, 2004). Recent work applied that theory in a new context: compiling to neural networks (Velez-Ginorio et al., 2026a;b). They showed that linear types could be used to constrain programming languages such that their programs compile into neural networks.

However, prior work focused on the theory of differentiable programming (i.e., expressing discrete structure in PyTorch), and on languages which compile to convolutional and recurrent neural networks. Cajal̀ instead compiles to transformers, introducing dictionaries which compile to linear attention heads. More importantly, we are introducing programming language theory as a tool for mechanistic interpretability. To our knowledge, this is the first paper to do so.

6. Discussion

Summary To validate new methods for understanding transformer representations, it helps to have transformers whose representations we already understand. Using those transformers, new techniques for analyzing representations can be verified against a ground truth. Prior work introduced languages for building such transformers but had a critical limitation—the closed program assumption. They could not compile programs with free variables. As a consequence, their programs must specify the entire algorithm a transformer implements, restricting applicability to hand-constructed transformers rather than ones trained by gradient descent. Using insights from programming language theory, Cajal̀ resolves this limitation. Its formal specification is proven correct, and our experiments show that transformers can learn to use compiled programs which only specify a part of their behavior. As a result, new techniques for analyzing representations can be verified against partial ground truths in transformers trained by gradient descent.

Limitations By compiling programs to k-linear maps, we can embed them within transformers. However, these linear maps are not necessarily *causal*. The i-th output coordinate of the linear map may be a function of the j-th input coordinate where $j \geq i$. In our experiments this causes no issue. Inputs look like abc|cba, and we construct compiled attention heads to only receive the sequence appearing before the delimiter. However, typical pretraining regimens of large language models possess no such delimiters, and therefore leveraging Cajal̀ in its current form would be more challenging: the *input* portion of the input sequence is not clear.

Future Directions We suspect these limitations can be resolved with *ordered* types, a more restrictive variant of linear types helpful for stream programming (Cutler et al., 2024). Programming language theory suggests that programs with ordered types may compile to causal k-linear maps (Mellies, 2009). If such a correspondence holds, we could design Cajal̀ so that partial specifications of behavior

can be injected into networks far larger than those considered here—it is our next step.

References

Axler, S. *Linear algebra done right*. Springer, 2024.

Bierman, G. M. What is a categorical model of intuitionistic linear logic? In *International Conference on Typed Lambda Calculi and Applications*, pp. 78–93. Springer, 1995.

Cardelli, L. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 266–277, 1997.

Chan, L., Garriga-Alonso, A., Goldowsky-Dill, N., Greenblatt, R., Nitishinskaya, J., Radhakrishnan, A., Shlegeris, B., and Thomas, N. Causal scrubbing: A method for rigorously testing interpretability hypotheses. In *AI Alignment Forum*, volume 2, pp. 19, 2022.

Cutler, J. W., Watson, C., Nkurumeh, E., Hilliard, P., Goldstein, H., Stanford, C., and Pierce, B. C. Stream types. *Proceedings of the ACM on Programming Languages*, 8 (PLDI):1412–1436, 2024.

Da Silva, P. Q., Sethuraman, H., Rajagopal, D., Hajishirzi, H., and Kumar, S. Steering off course: Reliability challenges in steering language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19856–19882, 2025.

Durmus, E., Tamkin, A., Clark, J., Wei, J., Marcus, J., Batson, J., Handa, K., Lovitt, L., Tong, M., McCain, M., Rausch, O., Huang, S., Bowman, S., Ritchie, S., Henighan, T., and Ganguli, D. Evaluating feature steering: A case study in mitigating social biases, 2024. URL https://anthropic.com/research/evaluating-feature-steering.

Elhage, N., Lasenby, R., and Olah, C. Privileged bases in the transformer residual stream. *Transformer Circuits Thread*, 24, 2023.

Hovgaard, A. K., Henriksen, T., and Elsman, M. High-performance defunctionalisation in futhark. In *International Symposium on Trends in Functional Programming*, pp. 136–156. Springer, 2018.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Jonas, E. and Kording, K. P. Could a neuroscientist understand a microprocessor? *PLoS computational biology*, 13 (1):e1005268, 2017.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Lazebnik, Y. Can a biologist fix a radio?—or, what i learned while studying apoptosis. *Cancer cell*, 2(3):179–182, 2002.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., and Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36:37876–37899, 2023.

MacIver, D. R., Hatfield-Dodds, Z., et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

Mellies, P.-A. Categorical semantics of linear logic. *Panoramas et syntheses*, 27:15–215, 2009.

Patterson, D. and Ahmed, A. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

Perconti, J. T. and Ahmed, A. Verifying an open compiler using multi-language semantics. In *European Symposium on Programming Languages and Systems*, pp. 128–148. Springer, 2014.

Pierce, B. C. *Types and programming languages*. MIT press, 2002.

Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al. Improving language understanding by generative pre-training. 2018.

Selinger, P. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

Shaw, P., Cohan, J., Eisenstein, J., Lee, K., Berant, J., and Toutanova, K. Alta: Compiler-based analysis of transformers. *arXiv preprint arXiv:2410.18077*, 2024.

Strobl, L., Merrill, W., Weiss, G., Chiang, D., and Angluin, D. What formal languages can transformers express? a survey. *Transactions of the Association for Computational Linguistics*, 12:543–561, 2024.

Velez-Ginorio, J., Amin, N., Kording, K., and Zdancewic, S. Compiling to linear neurons. *Proceedings of the ACM on Programming Languages*, 10(POPL):1010–1035, 2026a.

Velez-Ginorio, J., Amin, N., Kording, K., and Zdancewic, S. Compiling to recurrent neurons. *Proceedings of the ACM on Programming Languages*, (PLDI), 2026b.

Walker, D. Substructural type systems. *Advanced topics in types and programming languages*, pp. 3–44, 2005.

Weiss, G., Goldberg, Y., and Yahav, E. Thinking like transformers. In *International Conference on Machine Learning*, pp. 11080–11090. PMLR, 2021.

A. Theory

This appendix contains additional details on the formal specification and proofs in Cajal$_\searrow$.

A.1. Definitions

A.1.1. Syntax

There is an additional expression in the syntax not mentioned in the main text, an error expression which is convenient for the proofs but not necessary for introducing Cajal$_\searrow$. Similarly, there is a syntax for *values*, used in the formal specification of evaluation.

$$
\begin{array}{rcll}
e & := & \cdots & \\
 & | & \mathsf{error} & \text{error}
\end{array}
\qquad
\begin{array}{rcll}
v & := & * & \text{unit} \\
 & | & \mathsf{inj}_i\, v & \text{injection} \\
 & | & (e_1, e_2) & \text{tuple} \\
 & | & \{v_1 : v_2\} & \text{dictionary} \\
 & | & \mathsf{error} & \text{error}
\end{array}
$$

A.1.2. Evaluating

There are also additional evaluation rules not mentioned in the main text, again useful for proofs but unnecessary for introducing Cajal$_\searrow$.

$$
\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow \mathsf{error}}{E[e] \Downarrow \mathsf{error}}
$$

The E is an *evaluation context* (Pierce, 2002). It allows us to succinctly specify that if a subexpression evaluates to error, then the whole expression evaluates to error. The other rule simply says values evaluate to themselves. In the main text the lookup rule uses an operator $\sqcap_{\mathcal{V}}$, which we define below.

$$
\begin{aligned}
\sqcap_{\varnothing} &= \mathsf{error} \\
\sqcap_{\mathcal{V}} &= v_1 \sqcap v_2 \sqcap \cdots \sqcap v_n \text{ for all } v \in \mathcal{V}
\end{aligned}
$$

The operator $\sqcap_{\mathcal{V}}$ just generates a nondeterministic choice across all elements of $\mathcal{V}$, and if it is empty returns error.

Derivation of $e \Downarrow v$ The derivation D associated with $e \Downarrow v$ represents *the execution trace*. This is an important technical detail in our compiler correctness proofs.

$$
e \Downarrow_D v \iff D \text{ is a derivation of } e \Downarrow v
$$

A.2. Typing

The typing relation identifies expressions in Cajal$_\searrow$ with behavior. It is a cornerstone of the theory, as it allows us to consider in our proofs all possible program behaviors of interest.

$$
\frac{}{x : \tau \vdash x : \tau} \qquad \frac{}{\varnothing \vdash * : \mathsf{1}} \qquad \frac{}{\Delta \vdash \mathsf{error} : \tau} \qquad \frac{\Delta \vdash e : \tau_i}{\Delta \vdash \mathsf{inj}_i\, e : \tau_1 \oplus \tau_2}
$$

$$
\frac{\Delta_{\mathsf{k}} \vdash e_{\mathsf{k}} : \mathsf{List}_n(\mathsf{enum}(\tau_{\mathsf{k}})) \qquad \Delta_{\mathsf{v}} \vdash e_{\mathsf{v}} : \mathsf{List}_n(\tau_{\mathsf{v}})}{\Delta_{\mathsf{k}} \circ \Delta_{\mathsf{v}} \vdash \{e_{\mathsf{k}} : e_{\mathsf{v}}\} : \tau_{\mathsf{k}} \rightharpoonup \tau_{\mathsf{v}}} \qquad \frac{\Delta_1 \vdash e_1 : \tau_1 \qquad \Delta_2, x : \tau_1 \vdash e_2 : \tau_2}{\Delta_1 \circ \Delta_2 \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2}
$$

$$
\frac{\Delta_1 \vdash e : \tau_1 \oplus \tau_2 \qquad \Delta_2, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Delta_2, x_2 : \tau_2 \vdash e_2 : \tau}{\Delta_1 \circ \Delta_2 \vdash \mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1\, x_1 \Rightarrow e_1; \mathsf{inj}_2\, x_2 \Rightarrow e_2\} : \tau} \qquad \frac{\Delta \vdash e : \tau_1 \times \tau_2}{\Delta \vdash \pi_i e : \tau_i}
$$

$$
\frac{\Delta_{\mathsf{kv}} \vdash e_{\mathsf{kv}} : \tau_{\mathsf{k}} \rightharpoonup \tau_{\mathsf{v}} \qquad \Delta_{\mathsf{q}} \vdash e_{\mathsf{q}} : \mathsf{enum}(\tau_{\mathsf{q}}) \qquad \mathcal{R} \subseteq \mathsf{Val}(\tau_{\mathsf{q}}) \times \mathsf{Val}(\tau_{\mathsf{k}})}{\Delta_{\mathsf{kv}} \circ \Delta_{\mathsf{q}} \vdash e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}} : \tau_{\mathsf{v}}}
$$

$$\frac{\Delta \vdash e_1 : \tau \qquad \Delta \vdash e_2 : \tau}{\Delta \vdash e_1 \sqcap e_2 : \tau} \qquad \frac{\Delta_1 \vdash e_1 : \mathsf{1} \qquad \Delta_2 \vdash e_2 : \tau}{\Delta_1 \circ \Delta_2 \vdash e_1 ; e_2 : \tau}$$

$$\frac{\Delta \vdash e_1 : \tau_1 \qquad \Delta \vdash e_2 : \tau_2}{\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

Some rules reference an enum relation, which just checks that types are of the form $\mathsf{1} \oplus \mathsf{1} \oplus \cdots \oplus \mathsf{1}$.

A.2.1. Source environment

Source environments specify valid substitutions for expressions with free variables. Essentially these substitutions say that a free variable must be substituted by a value of the correct type (e.g., $x + 2$ requires a numeric value for x to make sense).

$$\frac{}{\varnothing \vdash \varnothing} \qquad \frac{\Delta \vdash \sigma \qquad \varnothing \vdash v : \tau}{\Delta, x{:}\tau \vdash \sigma\{x \mapsto v\}}$$

A.3. Compiling

Compiling types An important detail on compiling is how we compile *types*, which constrain the domain and codomain of the compiled k-linear maps. Because compiling dictionaries involves an inner product, types must compile to inner product spaces (Axler, 2024). The inner product in each case is always the Frobenius inner product, which generalizes the usual inner product on $\mathbb{R}^n$ to vector spaces of linear maps.

$$[\![_]\!] : \mathsf{Type} \to \mathsf{Inner\ Product\ Space}$$

$$\begin{aligned}
[\![\mathsf{1}]\!] &= \mathbb{R} \\
[\![\tau_1 \oplus \tau_2]\!] &= [\![\tau_1]\!] \oplus [\![\tau_2]\!] \\
[\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![\tau_\mathsf{k} \rightharpoonup \tau_\mathsf{v}]\!] &= \mathsf{Lin}([\![\tau_\mathsf{k}]\!], [\![\tau_\mathsf{v}]\!])
\end{aligned}$$

The operator $\oplus$ denotes the direct sum of vector spaces, $\times$ the direct product, and $\mathsf{Lin}(A, B)$ the space of linear maps f: $A \to B$.

Compiling contexts Contexts control the domain of the compiled k-linear maps. A context with k free variables is denoted by a k-tuple of vectors drawn from the vector spaces denoted by their types.

$$[\![_]\!] : \mathsf{Context} \to \mathsf{Tuple}(\mathsf{Vector})$$

$$\begin{aligned}
[\![\varnothing]\!] &= \{0\} \\
[\![\Delta, x{:}\tau]\!] &= [\![\Delta]\!] \times [\![\tau]\!]
\end{aligned}$$

Compiling programs There is one additional rule for compiling programs, for compiling error. It just returns the 0 vector from the vector space denoted by its type.

$$[\![\Delta \vdash \mathsf{error} : \tau]\!](\vec{\sigma}) = 0_{[\![\tau]\!]}$$

A.4. Proofs

The following proofs assume Barendregt's convention (Pierce, 2002), which states that variable names across bound and free variables are unique. This greatly simplifies proofs in a setting where binding mechanisms are not complicated. We additionally reference substitution $\{x \mapsto v\}(e)$ throughout the proofs—it is no more than replacing all instances of x with v in the expression e (Pierce, 2002).

A.4.1. THEOREM (PROGRAMS HAVE BEHAVIOR).

If $\Delta \vdash e : \tau$ and $\Delta \vdash \sigma$, then

$$\exists v, \sigma(e) \Downarrow v$$

Proof. By induction on $\Delta \vdash e : \tau$,

- **Case** $\varnothing \vdash * : \mathsf{1}$
  Let $\exists v = *$
  By evaluation $* \Downarrow *$
- **Case** $\Delta \vdash \mathsf{error} : \tau$
  Let $\exists v = \mathsf{error}$
  By substitution and evaluation $\sigma(\mathsf{error}) = \mathsf{error} \Downarrow \mathsf{error}$
- **Case** $x{:}\tau \vdash x : \tau$
  By assumption $x{:}\tau \vdash \{x \mapsto v\}$ where $v : \tau$
  Let $\exists v = \sigma(x)$
  By substitution and evaluation $\sigma(x) = v \Downarrow v$
- **Case** $\Delta \vdash \mathsf{inj}_1 e : \tau_1 \oplus \tau_2$
  By induction $\sigma(e) \Downarrow v_1$
  Let $\exists v = \mathsf{inj}_1 v_1$
  By substitution and evaluation $\sigma(\mathsf{inj}_1 e) = \mathsf{inj}_1 \sigma(e) \Downarrow \mathsf{inj}_1 v_1$
- **Case** $\Delta \vdash \mathsf{inj}_2 e : \tau_1 \oplus \tau_2$
  Similar to previous case
- **Case** $\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2$
  Let $\exists v = (\sigma(e_1), \sigma(e_2))$
  By substitution and evaluation $\sigma((e_1, e_2)) = (\sigma(e_1), \sigma(e_2)) \Downarrow (\sigma(e_1), \sigma(e_2))$
- **Case** $\Delta_1 \circ \Delta_2 \vdash \{e_1 : e_2\} : \tau_1 \rightharpoonup \tau_2$
  By splitting source environments $\sigma = \sigma_1 \circ \sigma_2$ where $\Delta_1 \vdash \sigma_1$ and $\Delta_2 \vdash \sigma_2$
  By induction and substitution $\sigma_1(e_1) = \sigma(e_1) \Downarrow v_1$
  By induction and substitution $\sigma_2(e_2) = \sigma(e_2) \Downarrow v_2$
  Let $\exists v = \{v_1 : v_2\}$
  By substitution and evaluation $\sigma(\{e_1 : e_2\}) = \{\sigma(e_1) : \sigma(e_2)\} = \{\sigma_1(e_1) : \sigma_2(e_2)\} \Downarrow \{v_1 : e_2\}$
- **Case** $\Delta_1 \circ \Delta_2 \vdash e_1; e_2 : \tau$
  Similar to previous case
- **Case** $\Delta_1 \circ \Delta_2 \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2$
  By splitting source environments $\sigma = \sigma_1 \circ \sigma_2$ where $\Delta_1 \vdash \sigma_1$ and $\Delta_2 \vdash \sigma_2$
  By induction and substitution $\sigma_1(e_1) = \sigma(e_1) \Downarrow v_1$
  By source environments $\Delta_2, x{:}\tau_1 \vdash \sigma_2\{x \mapsto v_1\}$
  By induction and substitution $\sigma_2\{x \mapsto v_1\}(e_2) = \sigma\{x \mapsto v_1\}(e_2) \Downarrow v_2$
  Because substitutions commute $\{x \mapsto v_1\}(\sigma(e_2)) \Downarrow v_2$
  Let $\exists v = v_2$
  By substitution and evaluation $\sigma(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) = (\mathsf{let}\, x = \sigma_1(e_1) \,\mathsf{in}\, \sigma_2(e_2)) \Downarrow v_2$
- **Case** $\Delta_1 \circ \Delta_2 \vdash \mathsf{case}\, e \,\mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\} : \tau$
  By splitting source environments $\sigma = \sigma_1 \circ \sigma_2$ where $\Delta_1 \vdash \sigma_1$ and $\Delta_2 \vdash \sigma_2$
  By induction and substitution $\sigma_1(e) = \sigma(e) \Downarrow v$
  Because evaluation preserves typing and by closing substitutions $v : \tau_1 \oplus \tau_2$
  By canonical forms either $v = \mathsf{inj}_1 v_1$ or $v = \mathsf{inj}_2 v_2$

Consider each possibility,

- **Case** $v = \mathsf{inj}_1\, v_1$
  By source environments $\Delta_2, x{:}\tau_1 \vdash \sigma_2\{x \mapsto v_1\}$
  By induction and substitution $\sigma_2\{x \mapsto v_1\}(e_1) = \sigma\{x \mapsto v_1\}(e_1) \Downarrow v_\bullet$
  Because substitutions commute $\{x \mapsto v_1\}(\sigma(e_1)) \Downarrow v_\bullet$
  Let $\exists v = v_\bullet$
  By substitution and evaluation $\mathsf{case}\, \sigma(e)\, \mathsf{of}\, \{\mathsf{inj}_1\, x_1 \Rightarrow \sigma(e_1); \mathsf{inj}_2\, x_2 \Rightarrow \sigma(e_2)\} \Downarrow v_\bullet$
- **Case** $v = \mathsf{inj}_2\, v_2$
  Similar to previous case

- **Case** $\Delta \vdash \pi_1 e : \tau_1$
  By induction $\sigma(e) \Downarrow v$
  Because evaluation preserves typing and by closing substitutions $v : \tau_1 \times \tau_2$
  By canonical forms $v = (v_1, v_2)$
  Let $\exists v = v_1$
  By substitution and evaluation $\sigma(\pi_1 e) = \pi_1 \sigma(e) \Downarrow v_1$
- **Case** $\Delta \vdash \pi_2 e : \tau_2$
  Similar to previous case
- **Case** $\Delta \vdash e_1 \sqcap e_2 : \tau$
  By induction $\sigma(e_1) \Downarrow v_1$
  By induction $\sigma(e_2) \Downarrow v_2$
  Let $\exists v = v_1$
  By substitution and evaluation $\sigma(e_1 \sqcap e_2) = \sigma(e_1) \sqcap \sigma(e_2) \Downarrow v_1$
- **Case** $\Delta_{\mathsf{kv}} \circ \Delta_{\mathsf{q}} \vdash e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}} : \tau_{\mathsf{v}}$
  By splitting source environments $\sigma = \sigma_{\mathsf{kv}} \circ \sigma_{\mathsf{q}}$ where $\Delta_{\mathsf{kv}} \vdash \sigma_{\mathsf{kv}}$ and $\Delta_{\mathsf{q}} \vdash \sigma_{\mathsf{q}}$
  By induction and substitution $\sigma_{\mathsf{kv}}(e_{\mathsf{kv}}) = \sigma(e_{\mathsf{kv}}) \Downarrow v_{\mathsf{kv}}$
  By induction and substitution $\sigma_{\mathsf{q}}(e_{\mathsf{q}}) = \sigma(e_{\mathsf{q}}) \Downarrow v_{\mathsf{q}}$
  Let $\mathcal{V} = \{(v_{\mathsf{v}})_i \mid (e_{\mathsf{q}}, (e_{\mathsf{k}})_i) \in \mathcal{R}\}$
  Consider whether $\mathcal{V}$ is empty,

  - **Case** $\mathcal{V} = \varnothing$
    By nondeterministic summation $\sqcap_\varnothing = \mathsf{error}$
    Let $\exists v = \mathsf{error}$
    By evaluation $\mathsf{error} \Downarrow \mathsf{error}$
    By substitution and evaluation $\sigma(e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}}) = \sigma_{\mathsf{kv}}(e_{\mathsf{kv}})(\sigma_{\mathsf{q}}(e_{\mathsf{q}}))_{\mathcal{R}} \Downarrow \mathsf{error}$
  - **Case** $\mathcal{V} \neq \varnothing$
    Let $v_{\mathsf{r}} \in \mathcal{V}$
    Let $\exists v = v_{\mathsf{r}}$
    Because nondeterministic sums of values return any summand $\sqcap_{\mathcal{V}}\, v \Downarrow v_{\mathsf{r}}$
    By substitution and evaluation $\sigma(e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}}) = \sigma(e_{\mathsf{kv}})(\sigma(e_{\mathsf{q}}))_{\mathcal{R}} \Downarrow v_{\mathsf{r}}$

■

A.4.2. THEOREM (COMPILER PRESERVES PROGRAM BEHAVIOR)

If $\Delta \vdash e : \tau$ and $\Delta \vdash \sigma$ and $V = \{(D, v) \mid \sigma(e) \Downarrow_D v\}$,

$$[\![e]\!]([\![\sigma]\!]) = \sum_{(D,v)\in V} [\![v]\!]$$

Proof. By induction on $\Delta \vdash e : \tau$

- **Case** $x : \tau \vdash x : \tau$
  Show $[\![x]\!]([\![v]\!]) = \sum_{(D,\,v)\in V} [\![v]\!]$
  By assumption $x : \tau \vdash \{x \mapsto v\}$
  By assumption $V = \{(D, v) \mid \{x \mapsto v\}(x) \Downarrow v\}$

$$\begin{aligned} & [\![x]\!]([\![v]\!]) \\ = \; & [\![v]\!] && \text{By compiling} \\ = \; & \sum_{(D,\,v)\in V} [\![v]\!] && \text{Because } V = \{(D, v)\} \end{aligned}$$

- **Case** $\varnothing \vdash * : \mathsf{1}$
  Show $[\![*]\!] = \sum_{(D,\,*)\in V} [\![*]\!]$
  By assumption $V = \{(D, *) \mid * \Downarrow *\}$

$$\begin{aligned} & [\![*]\!] \\ = \; & \sum_{(D,\,*)\in V} [\![*]\!] && \text{Because } V = \{(D, *)\} \end{aligned}$$

- **Case** $\Delta \vdash \mathsf{error} : \tau$
  Show $[\![\mathsf{error}]\!]([\![\sigma]\!]) = \sum_{(D,\,\mathsf{error})\in V} [\![\mathsf{error}]\!]$
  By assumption $V = \{(D, \mathsf{error}) \mid \mathsf{error} \Downarrow \mathsf{error}\}$

$$\begin{aligned} & [\![\mathsf{error}]\!]([\![\sigma]\!]) \\ = \; & \sum_{(D,\,\mathsf{error})\in V} [\![\mathsf{error}]\!] && \text{Because } V = \{(D, \mathsf{error})\} \end{aligned}$$

- **Case** $\Delta \vdash \mathsf{inj}_1\, e : \tau_1 \oplus \tau_2$
  Show $[\![\mathsf{inj}_1\, e]\!]([\![\sigma]\!]) = \sum_{(D,\,v)\in V} [\![v]\!]$
  By assumption $V = \{(D, v) \mid \sigma(\mathsf{inj}_1\, e) = \mathsf{inj}_1\, \sigma(e) \Downarrow_D v\}$
  Let $\bar{V} = \{(\bar{D}, \bar{v}) \mid \sigma(e) \Downarrow_{\bar{D}} \bar{v}\}$ by inversion on derivations in V
  Observe that $(D, v) \in V \iff (\bar{D}, \bar{v}) \in \bar{V}$ when $\bar{D}$ is obtained by inversion on D

$$\begin{aligned} & [\![\mathsf{inj}_1\, e]\!]([\![\sigma]\!]) \\ = \; & ([\![e]\!], 0) && \text{By compiling} \\ = \; & (\sum_{(\bar{D},\,\bar{v})\in \bar{V}} [\![\bar{v}]\!], 0) && \text{By induction} \\ = \; & \sum_{(\bar{D},\,\bar{v})\in \bar{V}} ([\![\bar{v}]\!], 0) && \text{Because } (a + b, 0) = (a, 0) + (b, 0) \\ = \; & \sum_{(D,\,v)\in V} ([\![v]\!], 0) && \text{Because } (D, v) \in V \iff (\bar{D}, \bar{v}) \in \bar{V} \\ = \; & \sum_{(D,\,v)\in V} [\![\mathsf{inj}_1\, v]\!] && \text{By compiling} \end{aligned}$$

- **Case** $\Delta \vdash \mathsf{inj}_2\, e : \tau_1 \oplus \tau_2$
  Similar to previous case

- **Case** $\Delta \vdash (e_1, e_2) : \tau_1 \times \tau_2$
  Show $[\![(e_1, e_2)]\!]([\![\sigma]\!]) = \sum_{(D,\, \sigma(e_1,e_2)) \in V} [\![\sigma(e_1, e_2)]\!]$
  By assumption $V = \{(D, (e_1, e_2)) \mid \sigma((e_1, e_2)) = (\sigma(e_1), \sigma(e_2)) \Downarrow_D (\sigma(e_1), \sigma(e_2))\}$
  Observe that $V = \{(D, (\sigma(e_1), \sigma(e_2))\} = \{(D, \sigma(e_1, e_2))\}$

$$
\begin{aligned}
& [\![(e_1, e_2)]\!]([\![\sigma]\!]) \\
= \; & [\![\sigma(e_1, e_2)]\!] && \text{Because compiling commutes with substitution} \\
= \; & \sum_{(D,\, \sigma(e_1,\, e_2)) \in V} [\![\sigma(e_1,\, e_2)]\!] && \text{Because } V = \{(D, \sigma(e_1, e_2))\}
\end{aligned}
$$

- **Case** $\Delta_{\mathsf{k}} \circ \Delta_{\mathsf{v}} \vdash \{e_{\mathsf{k}} : e_{\mathsf{v}}\} : \tau_{\mathsf{k}} \rightharpoonup \tau_{\mathsf{v}}$
  Show $[\![\{e_{\mathsf{k}} : e_{\mathsf{v}}\}]\!]([\![\sigma]\!]) = \sum_{(D,\, \{v_{\mathsf{k}}:v_{\mathsf{v}}\}) \in V} [\![\{v_{\mathsf{k}} : v_{\mathsf{v}}\}]\!]$
  By assumption $V = \{(D, \{v_{\mathsf{k}} : v_{\mathsf{v}}\}) \mid \sigma(\{e_{\mathsf{k}} : e_{\mathsf{v}}\}) \Downarrow_D \{v_{\mathsf{k}} : v_{\mathsf{v}}\}\}$
  Let $v_{\mathsf{k}} = \{(D_{\mathsf{k}}, v_{\mathsf{k}}) \mid \sigma(e_{\mathsf{k}}) \Downarrow_{D_{\mathsf{k}}} v_{\mathsf{k}}\}$ by inversion on derivations in V
  Let $v_{\mathsf{v}} = \{(D_{\mathsf{v}}, v_{\mathsf{v}}) \mid \sigma(e_{\mathsf{v}}) \Downarrow_{D_{\mathsf{v}}} v_{\mathsf{v}}\}$ by inversion on derivations in V
  Observe that $(D, \{v_{\mathsf{k}} : v_{\mathsf{v}}\}) \in V \iff (D_{\mathsf{k}}, v_{\mathsf{k}}) \in v_{\mathsf{k}} \;\wedge\; (D_{\mathsf{v}}, v_{\mathsf{v}}) \in v_{\mathsf{v}}$ when D_{k} and D_{v} are obtained by inversion on D

$$
\begin{aligned}
& [\![\{e_{\mathsf{k}} : e_{\mathsf{v}}\}]\!]([\![\sigma]\!]) \\
= \; & \left(\vec{x}_{\mathsf{q}} \mapsto \sum_{i=1}^{n} \langle [\![e_{\mathsf{k}}]\!]_i,\, \vec{x}_{\mathsf{q}} \rangle \cdot [\![e_{\mathsf{v}}]\!]_i\right) && \text{By compiling} \\
= \; & \left(\vec{x}_{\mathsf{q}} \mapsto \sum_{i=1}^{n} \langle \sum_{(D_{\mathsf{k}}, v_{\mathsf{k}}) \in v_{\mathsf{k}}} [\![v_{\mathsf{k}}]\!]_i,\, \vec{x}_{\mathsf{q}} \rangle \cdot \left(\sum_{(D_{\mathsf{v}}, v_{\mathsf{v}}) \in v_{\mathsf{v}}} [\![v_{\mathsf{v}}]\!]_i\right)\right) && \text{By induction} \\
= \; & \sum_{(D_{\mathsf{k}}, v_{\mathsf{k}}) \in v_{\mathsf{k}}} \sum_{(D_{\mathsf{v}}, v_{\mathsf{v}}) \in v_{\mathsf{v}}} \left(\vec{x}_{\mathsf{q}} \mapsto \sum_{i=1}^{n} \langle [\![v_{\mathsf{k}}]\!]_i,\, \vec{x}_{\mathsf{q}} \rangle \cdot [\![v_{\mathsf{v}}]\!]_i\right) \\
& \text{Because } [\![\{e_{\mathsf{k}} : e_{\mathsf{v}}\}]\!] \text{ is linear in } [\![e_{\mathsf{k}}]\!] \text{ and } [\![e_{\mathsf{v}}]\!] \\
= \; & \sum_{(D,\, \{v_{\mathsf{k}}:v_{\mathsf{v}}\}) \in V} \left(\vec{x}_{\mathsf{q}} \mapsto \sum_{i=1}^{n} \langle [\![v_{\mathsf{k}}]\!]_i,\, \vec{x}_{\mathsf{q}} \rangle \cdot [\![v_{\mathsf{v}}]\!]_i\right) \\
& \text{Because } (D, \{v_{\mathsf{k}} : v_{\mathsf{v}}\}) \in V \iff (D_{\mathsf{k}}, v_{\mathsf{k}}) \in v_{\mathsf{k}} \;\wedge\; (D_{\mathsf{v}}, v_{\mathsf{v}}) \in v_{\mathsf{v}} \\
= \; & \sum_{(D,\, \{v_{\mathsf{k}}:v_{\mathsf{v}}\}) \in V} [\![\{v_{\mathsf{k}} : v_{\mathsf{v}}\}]\!] && \text{By compiling}
\end{aligned}
$$

- **Case** $\Delta_1 \circ \Delta_2 \vdash e_1; e_2 : \tau$
  Similar to previous case
- **Case** $\Delta_1 \circ \Delta_2 \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2$
  Show $[\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!]([\![\sigma]\!]) = \sum_{(D,\, v_2) \in V} [\![v_2]\!]$
  By assumption $V = \{(D, v_2) \mid \sigma(\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2) \Downarrow_D v_2\}$
  Let $v_1 = \{(D_1, v_1) \mid \sigma(e_1) \Downarrow_{D_1} v_1\}$ by inversion on derivations in V
  Let $v_2 = \{(D_2, v_2) \mid (_, v_1) \in v_1,\, \{x \mapsto v_1\}(\sigma(e_2)) \Downarrow_{D_2} v_2\}$ by inverting derivations in V
  Observe that $(D, v_2) \in V \iff (D_1, v_1) \in v_1 \;\wedge\; (D_2, v_2) \in v_2$ when D_1 and D_2 are obtained by inversion on D

$$
\begin{aligned}
& [\![\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2]\!]([\![\sigma]\!]) \\
= \; & [\![e_2]\!]([\![e_1]\!]) && \text{By compiling} \\
= \; & [\![e_2]\!]\left(\sum_{(D_1,\, v_1) \in v_1} [\![v_1]\!]\right) && \text{By induction} \\
= \; & \sum_{(D_1,\, v_1) \in v_1} [\![e_2]\!]([\![v_1]\!]) && \text{Because } [\![e_2]\!] \text{ is linear} \\
= \; & \sum_{(D_1,\, v_1) \in v_1} [\![\{x \mapsto v_1\}(e_2)]\!] && \text{Because compiling commutes with substitution} \\
= \; & \sum_{(D_1,\, v_1) \in v_1} \sum_{(D_2,\, v_2) \in v_2} [\![v_2]\!] && \text{By induction} \\
= \; & \sum_{(D,\, v_2) \in V} [\![v_2]\!]
\end{aligned}
$$

- **Case** $\Delta_1 \circ \Delta_2 \vdash \mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\} : \tau_2$
  By assumption $V = \{(D, v) \mid \sigma(\mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}) \Downarrow v\}$
  Let $v_{12} = \{(D_{12}, v_{12}) \mid \sigma(e) \Downarrow_{D_{12}} v_{12}\}$ by inversion on derivations in V

$$
\begin{aligned}
& [\![\mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}]\!]([\![\sigma]\!]) \\
&= [\![e_1]\!]([\![e]\!]_1) + [\![e_2]\!]([\![e]\!]_2) && \text{By compiling} \\
&= [\![e_1]\!]\Big(\Big(\textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![v_{12}]\!]\Big)_1\Big) + [\![e_2]\!]\Big(\Big(\textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![v_{12}]\!]\Big)_2\Big) && \text{By induction} \\
&= [\![e_1]\!]\Big(\textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![v_{12}]\!]_1\Big) + [\![e_2]\!]\Big(\textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![v_{12}]\!]_2\Big) \\
& \quad \text{Because first and second projection are linear} \\
&= \Big(\textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![e_1]\!]([\![v_{12}]\!]_1)\Big) + \Big(\textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![e_2]\!]([\![v_{12}]\!]_2)\Big) && \text{Because } [\![e_i]\!] \text{ is linear} \\
&= \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} \Big([\![e_1]\!]([\![v_{12}]\!]_1) + [\![e_2]\!]([\![v_{12}]\!]_2)\Big) \\
& \quad \text{Because } \textstyle\sum_a f(a) + \sum_a g(a) = \sum_a \Big(f(a) + g(a)\Big)
\end{aligned}
$$

  For $(D_{12}, v_{12}) \in v_{12}$, by inversion on D_{12} we know v_{12} is either error, $\mathsf{inj}_1 v_{22}$, or $\mathsf{inj}_2 v_{22}$.

  - **Case** $v_{12} = \mathsf{inj}_1 v_{22}$
    Let $v_{22} = \{(D_{22}, v) \mid (_, \mathsf{inj}_1 v_{22}) \in v_{12},\ \{x_1 \mapsto v_{22}\}(\sigma(e_1)) \Downarrow v\}$
    Observe that $(D, v) \in V \iff (D_{12}, v_{12}) \in v_{12} \ \wedge\ (D_{22}, v) \in v_{22}$ when D_{12} and D_{22} are obtained by inversion on D

$$
\begin{aligned}
& \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} \Big([\![e_1]\!]([\![\mathsf{inj}_1 v_{22}]\!]_1) + [\![e_2]\!]([\![\mathsf{inj}_1 v_{22}]\!]_2)\Big) \\
& \quad \text{Because } \textstyle\sum_a f(a) + \sum_a g(a) = \sum_a \Big(f(a) + g(a)\Big) \\
&= \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} \Big([\![e_1]\!](([\![v_{22}]\!], 0)_1) + [\![e_2]\!]([\![e_1]\!](([\![v_{22}]\!], 0)_2))\Big) && \text{By compiling} \\
&= \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} \Big([\![e_1]\!]([\![v_{22}]\!]) + [\![e_2]\!]([\![e_1]\!](0))\Big) && \text{By first and second projection} \\
&= \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![e_1]\!]([\![v_{22}]\!]) && \text{By linearity } [\![e_i]\!](0) = 0 \\
&= \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![\{x_1 \mapsto v_{22}\}(\sigma(e_1))]\!] \\
& \quad \text{Because compiling commutes with substitution} \\
&= \textstyle\sum_{(D_{12},\, v_{12}) \in v_{12}} \sum_{(D_{22},\, v) \in v_{22}} [\![v]\!] && \text{By induction} \\
&= \textstyle\sum_{(D,\, v) \in V} [\![v]\!]
\end{aligned}
$$

  - **Case** $v_{12} = \mathsf{inj}_2 v_{22}$
    Similar to previous case
  - **Case** $v_{12} = \mathsf{error}$
    Similar to previous case

- **Case** $\Delta \vdash \pi_1 e : \tau_1$
  Show $[\![\pi_1 e]\!]([\![\sigma]\!]) = \sum_{(D,\, v_1) \in V} [\![v_1]\!]$
  By assumption $V = \{(D, v) \mid \sigma(\pi_1 e) \Downarrow v\}$

Let $v_{12} = \{(D_{12}, v_{12}) \mid \sigma(e) \Downarrow_{D_1} v_{12}\}$ by inversion on derivations in V

$$\begin{aligned}
& [\![\pi_1 e]\!]([\![\sigma]\!]) \\
&= [\![e]\!]_1 && \text{By compiling} \\
&= \left(\sum_{(D_{12},\, v_{12}) \in v_{12}} [\![v_{12}]\!]\right)_1 && \text{By induction} \\
&= \sum_{(D_{12},\, v_{12}) \in v_{12}} [\![v_{12}]\!]_1 && \text{Because } [\![v_{12}]\!]_1 \text{ is linear}
\end{aligned}$$

For $(D_{12}, v_{12}) \in v_{12}$, by inversion on D_{12} we know v_{12} is either (e_1, e_2) or error.

- **Case** $v_{12} = (e_1, e_2)$
  Let $v_{22} = \{(D_{22}, v) \mid (_, (e_1, e_2)) \in v_{12},\ \sigma(e_1) \Downarrow v\}$
  Observe that $(D, v) \in V \iff (D_{12}, (e_1, e_2)) \in v_{12} \ \wedge\ (D_{22}, v) \in v_{22}$ when D_{12} and D_{22} are obtained by inversion on D

$$\begin{aligned}
& \sum_{(D_{12},\, (e_1, e_2)) \in v_{12}} [\![(e_1, e_2)]\!]_1 \\
&= \sum_{(D_{12},\, (e_1, e_2)) \in v_{12}} ([\![e_1]\!], [\![e_2]\!])_1 && \text{By compiling} \\
&= \sum_{(D_{12},\, (e_1, e_2)) \in v_{12}} [\![e_1]\!] && \text{By first projection} \\
&= \sum_{(D_{12},\, (e_1, e_2)) \in v_{12}} \sum_{(D_{22},\, v) \in v_{22}} [\![v]\!] && \text{By induction} \\
&= \sum_{(D,\, v) \in V} [\![v]\!]
\end{aligned}$$

- **Case** $v_{12} = \mathsf{error}$
  Similar to previous case

- **Case** $\Delta \vdash \pi_2 e : \tau_2$
  Similar to previous case
- **Case** $\Delta \vdash e_1 \sqcap e_2 : \tau$
  Show $[\![e_1 \sqcap e_2]\!]([\![\sigma]\!]) = \sum_{(D,\, v) \in V} [\![v]\!]$
  By assumption $V = \{(D, v) \mid \sigma(e_1 \sqcap e_2) \Downarrow_D v\}$
  Let $v_1 = \{(D_1, v) \mid \sigma(e_1) \Downarrow_{D_1} v\}$ by inversion on derivations in V
  Let $v_2 = \{(D_2, v) \mid \sigma(e_2) \Downarrow_{D_2} v\}$ by inversion on derivations in V
  Observe that $(D, v) \in V \iff (D_1, v) \in v_1 \ \vee\ (D_2, v) \in v_2$ when D_1 and D_2 are obtained by inversion on D

$$\begin{aligned}
& [\![e_1 \sqcap e_2]\!]([\![\sigma]\!]) \\
&= [\![e_1]\!] + [\![e_2]\!] && \text{By compiling} \\
&= \sum_{(D_1,\, v) \in v_1} [\![v]\!] + \sum_{(D_2,\, v) \in v_2} [\![v]\!] && \text{By induction} \\
&= \sum_{(_,\, v) \in v_1 \cup v_2} [\![v]\!] && \text{Because } \sum_{a \in A} a + \sum_{b \in B} b = \sum_{c \in A \cup B} c \\
&= \sum_{(D,\, v) \in V} [\![v]\!] && \text{Because } (D, v) \in V \iff (D_1, v) \in v_1 \ \vee\ (D_2, v) \in v_2
\end{aligned}$$

- **Case** $\Delta_{\mathsf{k}} \circ \Delta_{\mathsf{v}} \vdash e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}} : \tau_{\mathsf{v}}$
  Show $[\![e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}}]\!]([\![\sigma]\!]) = \sum_{(D,\, v) \in V} [\![v]\!]$
  By assumption $V = \{(D, v) \mid \sigma(e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}}) \Downarrow_D v\}$
  Let $v_{\mathsf{q}} = \{(D_{\mathsf{q}}, v_{\mathsf{q}}) \mid \sigma(e_{\mathsf{q}}) \Downarrow_{D_{\mathsf{q}}} v_{\mathsf{q}}\}$ by inversion on derivations in V
  Let $v_{\mathsf{kv}} = \{(D_{\mathsf{kv}}, v_{\mathsf{kv}}) \mid \sigma(e_{\mathsf{kv}}) \Downarrow_{D_{\mathsf{kv}}} v_{\mathsf{kv}}\}$ by inversion on derivations in V

$$
\begin{aligned}
& [\![e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}}]\!]([\![\sigma]\!]) \\
&= [\![e_{\mathsf{kv}}]\!]([\![\mathcal{R}]\!]([\![e_{\mathsf{q}}]\!])) && \text{By compiling} \\
&= [\![e_{\mathsf{kv}}]\!]([\![\mathcal{R}]\!](\textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}[\![v_{\mathsf{q}}]\!])) && \text{By induction} \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}[\![e_{\mathsf{kv}}]\!]([\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])) && \text{Because } [\![\mathcal{R}]\!] \text{ and } [\![e_{\mathsf{kv}}]\!] \text{ are linear}
\end{aligned}
$$

For $(D_{\mathsf{kv}}, v_{\mathsf{kv}}) \in v_{\mathsf{kv}}$, by inversion on D_{kv} we know v_{kv} is either $\{v_{\mathsf{k}} : v_{\mathsf{v}}\}$ or error.

- **Case** $v_{\mathsf{kv}} = \{v_{\mathsf{k}} : v_{\mathsf{v}}\}$
  Let $v_{\mathsf{r}} = \{(D_{\mathsf{r}}, v) \mid (_, v_{\mathsf{q}}) \in v_{\mathsf{q}},\ (_, \{v_{\mathsf{k}} : v_{\mathsf{v}}\}) \in v_{\mathsf{kv}},\ \sqcap_{((v_{\mathsf{k}})_i, v_{\mathsf{q}})\in\mathcal{R}}(v_{\mathsf{v}})_i \Downarrow_{D_{\mathsf{r}}} v\}$ by inversion on derivations in V
  Observe that $(D, v) \in V \iff (D_{\mathsf{q}}, v_{\mathsf{q}}) \in v_{\mathsf{q}} \ \wedge\ (D_{\mathsf{kv}}, \{v_{\mathsf{k}} : v_{\mathsf{v}}\}) \in v_{\mathsf{kv}} \ \wedge\ (D_{\mathsf{r}}, v) \in v_{\mathsf{r}}$ when D_{kv} and D_{kv} and D_{r} are obtained by inversion on D

$$
\begin{aligned}
& \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}[\![\{e_{\mathsf{k}} : e_{\mathsf{v}}\}]\!]([\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])) \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\Big(\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}[\![\{v_{\mathsf{k}} : v_{\mathsf{v}}\}]\!]\Big)([\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])) \\
& \quad \text{By induction} \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}([\![\{v_{\mathsf{k}} : v_{\mathsf{v}}\}]\!])([\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])) \\
& \quad \text{Because } \textstyle\sum_a(f_1(a) + f_2(a)) = \sum_a\sum_i f_i(a) \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}\Big(\vec{x}_{\mathsf{q}} \mapsto \sum_{i=1}^{n}\langle [\![v_{\mathsf{k}}]\!]_i, \vec{x}_{\mathsf{q}}\rangle \cdot [\![v_{\mathsf{v}}]\!]_i\Big)([\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])) \\
& \quad \text{By compiling} \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}\Big(\sum_{i=1}^{n}\langle [\![v_{\mathsf{k}}]\!]_i, [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])\rangle \cdot [\![v_{\mathsf{v}}]\!]_i\Big) \\
& \quad \text{By function application} \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}\Big(\sum_{((v_{\mathsf{k}})_i,v_{\mathsf{q}})\in\mathcal{R}}[\![v_{\mathsf{v}}]\!]_i\Big) \\
& \quad \text{Because } \mathcal{R} \text{ is implemented by } \langle_,_\rangle \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}\Big(\sum_{((v_{\mathsf{k}})_i,v_{\mathsf{q}})\in\mathcal{R}}[\![(v_{\mathsf{v}})_i]\!]\Big) \\
& \quad \text{By compiling} \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}\Big([\![\sqcap_{((v_{\mathsf{k}})_i,v_{\mathsf{q}})\in\mathcal{R}}(v_{\mathsf{v}})_i]\!]\Big) \\
& \quad \text{By compiling} \\
&= \textstyle\sum_{(D_{\mathsf{q}},v_{\mathsf{q}})\in v_{\mathsf{q}}}\sum_{(D_{\mathsf{kv}},\{v_{\mathsf{k}}:v_{\mathsf{v}}\})\in v_{\mathsf{kv}}}\Big(\sum_{(D_{\mathsf{r}},v)\in v_{\mathsf{r}}}[\![v]\!]\Big) \\
& \quad \text{By induction} \\
&= \textstyle\sum_{(D,v)\in V}[\![v]\!]
\end{aligned}
$$

- **Case** $v_{\mathsf{kv}} = \mathsf{error}$
  Similar to previous case

■

A.4.3. LEMMA ($\mathcal{R}$ IS IMPLEMENTED BY $\langle_,_\rangle$)

If $(v_{\mathsf{k}})_i : \mathsf{enum}(\tau_{\mathsf{k}})$ and $v_{\mathsf{q}} : \mathsf{enum}(\tau_{\mathsf{q}})$, then

- $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \in \mathcal{R} \implies \langle [\![v_{\mathsf{k}}]\!]_i, [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])\rangle \cdot [\![v_{\mathsf{v}}]\!]_i = [\![v_{\mathsf{v}}]\!]_i$
- $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \notin \mathcal{R} \implies \langle [\![v_{\mathsf{k}}]\!]_i, [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])\rangle = 0$

Proof. Consider each implication,

- **Case** $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \in \mathcal{R} \implies \langle [\![v_{\mathsf{k}}]\!]_i, [\![R]\!]([\![v_{\mathsf{q}}]\!])\rangle \cdot [\![v_{\mathsf{v}}]\!]_i = [\![v_{\mathsf{v}}]\!]_i$
  By assumption $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \in \mathcal{R}$
  Because $(v_{\mathsf{k}})_i : \mathsf{enum}(\tau_{\mathsf{k}})$, $[\![(v_{\mathsf{k}})_i]\!]$ is a one-hot vector, where $[\![(v_{\mathsf{k}})_i]\!]_j = 1$
  Since all other entries of $[\![(v_{\mathsf{k}})_i]\!]_j$ are 0, $\langle [\![v_{\mathsf{k}}]\!]_i, [\![R]\!]([\![v_{\mathsf{q}}]\!])\rangle = \langle [\![(v_{\mathsf{k}})_i]\!], [\![R]\!]([\![v_{\mathsf{q}}]\!])\rangle = [\![(v_k)_i]\!]_j \cdot [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])_j$
  Observe that $[\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])_j = 1$ when $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \in \mathcal{R}$

$$
\begin{aligned}
& \langle [\![v_{\mathsf{k}}]\!]_i, [\![R]\!]([\![v_{\mathsf{q}}]\!])\rangle \cdot [\![v_{\mathsf{v}}]\!]_i \\
= & \langle [\![(v_{\mathsf{k}})_i]\!], [\![R]\!]([\![v_{\mathsf{q}}]\!])\rangle \cdot [\![v_{\mathsf{v}}]\!]_i && \text{By compiling} \\
= & [\![(v_k)_i]\!]_j \cdot [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])_j \cdot [\![v_{\mathsf{v}}]\!]_i && \text{Because } [\![(v_{\mathsf{k}})_i]\!] \text{ is one-hot} \\
= & 1 \cdot [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])_j \cdot [\![v_{\mathsf{v}}]\!]_i && \text{Because } [\![(v_{\mathsf{k}})_i]\!]_j = 1 \\
= & 1 \cdot 1 \cdot [\![v_{\mathsf{v}}]\!]_i && \text{Because } [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])_j = 1 \\
= & [\![v_{\mathsf{v}}]\!]_i && \text{By multiplication}
\end{aligned}
$$

- **Case** $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \notin \mathcal{R} \implies \langle [\![v_{\mathsf{k}}]\!]_i, [\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])\rangle = 0$
  Similar to previous case but using $[\![\mathcal{R}]\!]([\![v_{\mathsf{q}}]\!])_j = 0$ when $((v_{\mathsf{k}})_i, v_{\mathsf{q}}) \notin \mathcal{R}$

■

A.4.4. LEMMA (COMPILER COMMUTES WITH SUBSTITUTION)

If $\Delta, x{:}\tau_1 \vdash e : \tau_2$,

$$[\![\{x \mapsto v\}(e)]\!](\vec{\sigma}) = [\![e]\!](\vec{\sigma}, [\![v]\!])$$

Proof. By induction on e. The argument follows a similar structure to the substitution lemma proved in (Velez-Ginorio et al., 2026b).

■

A.4.5. LEMMA (COMPILER MAPS PROGRAMS TO MULTILINEAR MAPS)

If $\Delta, x{:}\tau_1 \vdash e : \tau_2$,

$$[\![e]\!](\vec{\sigma}, \alpha_1 \cdot \vec{x_1} + \alpha_2 \cdot \vec{x_2}) = \alpha_1 \cdot [\![e]\!](\vec{\sigma}, \vec{x_1}) + \alpha_2 \cdot [\![e]\!](\vec{\sigma}, \vec{x_2})$$

Proof. Immediate from compiler mapping programs to additive and homogeneous maps.

■

A.4.6. LEMMA (COMPILER MAPS PROGRAMS TO ADDITIVE MAPS)

If $\Delta, x{:}\tau_1 \vdash e : \tau_2$,

$$[\![e]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) = [\![e]\!](\vec{\sigma}, \vec{x_1}) + [\![e]\!](\vec{\sigma}, \vec{x_2})$$

Proof. By induction on e,

- **Case** y
  Consider whether $x = y$,
  - **Case** $x = y$
    By inversion $\Delta = \varnothing$

$$
\begin{aligned}
& [\![x]\!](\vec{x} + \vec{y}) \\
= & \vec{x} + \vec{y} && \text{By compiling} \\
= & [\![x]\!](\vec{x}) + [\![y]\!](\vec{y}) && \text{By compiling}
\end{aligned}
$$

- **Case** $x \neq y$
  Vacuous because we assume a contradiction $\Delta, x : \tau_1 \vdash y : \tau_2$

- **Case** $*$
  Vacuous because we assume $\Delta, x : \tau_1 \vdash * : \tau_2$
- **Case** error

$$
\begin{aligned}
& [\![\mathsf{error}]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \\
&= 0 && \text{By compiling} \\
&= 0 + 0 && \text{Because } 0 + 0 = 0 \\
&= [\![\mathsf{error}]\!](\vec{\sigma}, \vec{x_1}) + [\![\mathsf{error}]\!](\vec{\sigma}, \vec{x_2}) && \text{By compiling}
\end{aligned}
$$

- **Case** $\mathsf{inj}_1\, e$

$$
\begin{aligned}
& [\![\mathsf{inj}_1\, e]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \\
&= ([\![e]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}), 0) && \text{By compiling} \\
&= ([\![e]\!](\vec{\sigma}, \vec{x_1}) + [\![e]\!](\vec{\sigma}, \vec{x_2}), 0) && \text{By induction} \\
&= ([\![e]\!](\vec{\sigma}, \vec{x_1}), 0) + ([\![e]\!](\vec{\sigma}, \vec{x_2}), 0) && \text{Because } (a + b, 0) = (a, 0) + (b, 0) \\
&= [\![\mathsf{inj}_1\, e]\!](\vec{\sigma}, \vec{x_1}) + [\![\mathsf{inj}_1\, e]\!](\vec{\sigma}, \vec{x_2}) && \text{By compiling}
\end{aligned}
$$

- **Case** $\mathsf{inj}_2\, e$
  Similar to previous case
- **Case** (e_1, e_2)

$$
\begin{aligned}
& [\![(e_1, e_2)]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \\
&= ([\![e_1]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}), [\![e_2]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2})) && \text{By compiling} \\
&= ([\![e_1]\!](\vec{\sigma}, \vec{x_1}) + [\![e_1]\!](\vec{\sigma}, \vec{x_2}), [\![e_2]\!](\vec{\sigma}, \vec{x_1}) + [\![e_2]\!](\vec{\sigma}, \vec{x_2})) && \text{By induction} \\
&= ([\![e_1]\!](\vec{\sigma}, \vec{x_1}), [\![e_2]\!](\vec{\sigma}, \vec{x_1})) + ([\![e_1]\!](\vec{\sigma}, \vec{x_2}), [\![e_2]\!](\vec{\sigma}, \vec{x_2})) && \text{Because } (a + b, c + d) = (a, c) + (b, d) \\
&= [\![(e_1, e_2)]\!](\vec{\sigma}, \vec{x_1}) + [\![(e_1, e_2)]\!](\vec{\sigma}, \vec{x_2}) && \text{By compiling}
\end{aligned}
$$

- **Case** $\{e_\mathsf{k} : e_\mathsf{v}\}$
  Consider whether $x \in \mathsf{fv}(e_\mathsf{k})$ or $x \in \mathsf{fv}(e_\mathsf{v})$,

- **Case** $x \in \mathsf{fv}(e_\mathsf{k})$

$$
\begin{aligned}
& [\![\{e_\mathsf{k} : e_\mathsf{v}\}]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) \\
= \; & \vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_1} + \vec{x_2})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \\
& \text{By compiling} \\
= \; & \vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_1})_i + [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_2})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \\
& \text{By induction} \\
= \; & \vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} (\langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_1})_i, \vec{x}_\mathsf{q} \rangle + \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_2})_i, \vec{x}_\mathsf{q} \rangle) \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \\
& \text{Because inner product is bilinear} \\
= \; & \vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_1})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i + \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_2})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \\
& \text{Because } \cdot \text{ distributes over } + \\
= \; & \vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_1})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \\
& \quad + \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_2})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \\
& \text{Because } \sum_a f(a) + f(b) = \sum_a f(a) + \sum_a f(b) \\
= \; & \left(\vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_1})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \right) \\
+ \; & \left(\vec{x}_\mathsf{q} \mapsto \sum_{i=1}^{n} \langle [\![e_\mathsf{k}]\!](\vec{\sigma_\mathsf{k}}, \vec{x_2})_i, \vec{x}_\mathsf{q} \rangle \cdot [\![e_\mathsf{v}]\!](\vec{\sigma_\mathsf{v}})_i \right) \\
& \text{By addition of functions} \\
= \; & [\![\{e_\mathsf{k} : e_\mathsf{v}\}]\!](\vec{\sigma}, \vec{x_1}) + [\![\{e_\mathsf{k} : e_\mathsf{v}\}]\!](\vec{\sigma}, \vec{x_2}) \\
& \text{By compiling}
\end{aligned}
$$

- **Case** $x \in \mathsf{fv}(e_\mathsf{v})$
  Similar to previous case

- **Case** $e_1; e_2$
  Similar to previous case
- **Case** $\pi_i e$

$$
\begin{aligned}
& [\![\pi_i e]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \\
= \; & [\![e]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2})_i & \text{By compiling} \\
= \; & [\![e]\!](\vec{\sigma}, \vec{x_1})_i + [\![e]\!](\vec{\sigma}, \vec{x_2})_i & \text{By induction} \\
= \; & [\![\pi_i e]\!](\vec{\sigma}, \vec{x_1}) + [\![\pi_i e]\!](\vec{\sigma}, \vec{x_2}) & \text{By compiling}
\end{aligned}
$$

- **Case** $\mathsf{case}\ e\ \mathsf{of}\ \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}$
  Consider whether $x \in \mathsf{fv}(e)$ or $(x \in \mathsf{fv}(e_1)) \wedge (x \in \mathsf{fv}(e_2))$,

- **Case** $x \in \mathsf{fv}(e)$

$$
\begin{aligned}
& [\![\mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) \\
= {} & [\![e_1]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1} + \vec{x_2})_1) + [\![e_2]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1} + \vec{x_2})_2) \\
& \text{By compiling} \\
= {} & [\![e_1]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1})_1 + [\![e]\!](\vec{\sigma_1}, \vec{x_2})_1) + [\![e_2]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1})_2 + [\![e]\!](\vec{\sigma_1}, \vec{x_2})_2) \\
& \text{By induction and linearity of projection} \\
= {} & [\![e_1]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1})_1) + [\![e_1]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_2})_1) \\
& + [\![e_2]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1})_2) + [\![e_2]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_2})_2) \\
& \text{By induction} \\
= {} & [\![e_1]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1})_1) + [\![e_2]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_1})_2) \\
& + [\![e_1]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_2})_1) + [\![e_2]\!](\vec{\sigma_2}, [\![e]\!](\vec{\sigma_1}, \vec{x_2})_2) \\
& \text{Because } + \text{ is commutative} \\
= {} & [\![\mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}]\!](\vec{\sigma}, \vec{x_1}) \\
& + [\![\mathsf{case}\, e\, \mathsf{of}\, \{\mathsf{inj}_1 x_1 \Rightarrow e_1; \mathsf{inj}_2 x_2 \Rightarrow e_2\}]\!](\vec{\sigma}, \vec{x_2}) \\
& \text{By compiling}
\end{aligned}
$$

- **Case** $(x \in \mathsf{fv}(e_1)) \wedge (x \in \mathsf{fv}(e_2))$
  Similar to previous case

- **Case** $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$
  Consider whether $x \in \mathsf{fv}(e_1)$ or $x \in \mathsf{fv}(e_2)$,

  - **Case** $x \in \mathsf{fv}(e_1)$

$$
\begin{aligned}
& [\![\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \\
= {} & [\![e_2]\!](\vec{\sigma_2}, [\![e_1]\!](\vec{\sigma_1}, \vec{x_1} + \vec{x_2})) & \text{By compiling} \\
= {} & [\![e_2]\!](\vec{\sigma_2}, [\![e_1]\!](\vec{\sigma_1}, \vec{x_1}) + [\![e_1]\!](\vec{\sigma_1}, \vec{x_2})) & \text{By induction} \\
= {} & [\![e_2]\!](\vec{\sigma_2}, [\![e_1]\!](\vec{\sigma_1}, \vec{x_1})) + [\![e_2]\!]([\![e_1]\!](\vec{\sigma_1}, \vec{x_2})) & \text{By induction} \\
= {} & [\![\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]\!](\vec{\sigma}, \vec{x_1}) + [\![\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]\!](\vec{\sigma}, \vec{x_2}) & \text{By compiling}
\end{aligned}
$$

  - **Case** $x \in \mathsf{fv}(e_2)$
    Similar to previous case

- **Case** $e_{\mathsf{kv}}(e_{\mathsf{q}})_{\mathcal{R}}$
  Similar to previous case, but also using linearity of $[\![\mathcal{R}]\!]$

- **Case** $e_1 \sqcap e_2$

$$
\begin{aligned}
& [\![e_1 \sqcap e_2]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \\
= {} & [\![e_1]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) + [\![e_2]\!](\vec{\sigma}, \vec{x_1} + \vec{x_2}) & \text{By compiling} \\
= {} & [\![e_1]\!](\vec{\sigma}, \vec{x_1}) + [\![e_1]\!](\vec{\sigma}, \vec{x_2}) + [\![e_2]\!](\vec{\sigma}, \vec{x_1}) + [\![e_2]\!](\vec{\sigma}, \vec{x_2}) & \text{By induction} \\
= {} & [\![e_1]\!](\vec{\sigma}, \vec{x_1}) + [\![e_2]\!](\vec{\sigma}, \vec{x_1}) + [\![e_1]\!](\vec{\sigma}, \vec{x_2}) + [\![e_2]\!](\vec{\sigma}, \vec{x_2}) & \text{Because } + \text{ is commutative} \\
= {} & [\![e_1 \sqcap e_2]\!](\vec{\sigma}, \vec{x_1}) + [\![e_1 \sqcap e_2]\!](\vec{\sigma}, \vec{x_2}) & \text{By compiling}
\end{aligned}
$$

■

A.4.7. LEMMA (COMPILER MAPS PROGRAMS TO HOMOGENEOUS MAPS)

If $\Delta, x : \tau_1 \vdash e : \tau_2$,

$$[\![e]\!](\vec{\sigma}, \alpha \cdot \vec{x}) = \alpha \cdot [\![e]\!](\vec{\sigma}, \vec{x})$$

Proof. By induction on e. The argument follows a similar structure to proving that the compiler maps programs to additive maps.

■

B. Experiments

This appendix contains additional details on experiments from the main paper, in addition to other experiments ran to supplement the findings in this work. For the conditional reverse task, we take the opportunity to highlight insights we omitted from the main paper in light of space constraints. For the other experiments, we give brief summaries of the findings.

The code for analyses and generating the figures are available in the supplementary material. They are a useful reference for finer details (Appendix C).

B.1. Conditional Reverse

B.1.1. Models

The transformers we use (similar to Fig. 1) have their MLP layer before attention. This is atypical. But for the conditional reverse task it is the more natural ordering. The MLP can tag tokens with information about whether they are vowels, and attention can leverage that information to perform the conditional reverse. We tried the typical ordering but `Model I` struggles to solve the task when limited to a one-layer transformer.

Subsequent experiments (e.g., rotate and conditional rotate) test transformers with a natural ordering, and with two layers of transformers.

B.1.2. Does compiling change learning?

In Fig. 6, the heatmaps for each model show a horizontal line toward the midpoint of the y-axis. Each model learns the correct output token at the 8th position faster than the rest. This is because for the conditional reverse task on a sequence of 15 characters, the 8th position is always identity. The task at that position is simpler.

B.1.3. Does compiling change representations?

In Fig. 7, each PCA plot shows representations projected onto the first 3 principal components (PCs). In general, these three components tend to explain around 50% of variance in each plot. PC1 is the bottom-left axis. PC2 is the bottom-right axis. PC3 is the vertical axis. The exact amount of variance explained by each is below:

Model	Representation	PC1	PC2	PC3
F	R2	45.4%	19.7%	14.1%
	A	47.9%	23.3%	10.1%
P	R2	21.1%	16.8%	13.7%
	A	20.6%	15.3%	14.1%
T	R2	25.4%	20.4%	18.6%
	A	22.9%	19.6%	18.1%
I	R2	18.3%	16.2%	14.4%
	A	18.4%	16.2%	14.4%

B.1.4. Does compiling change the necessary representations?

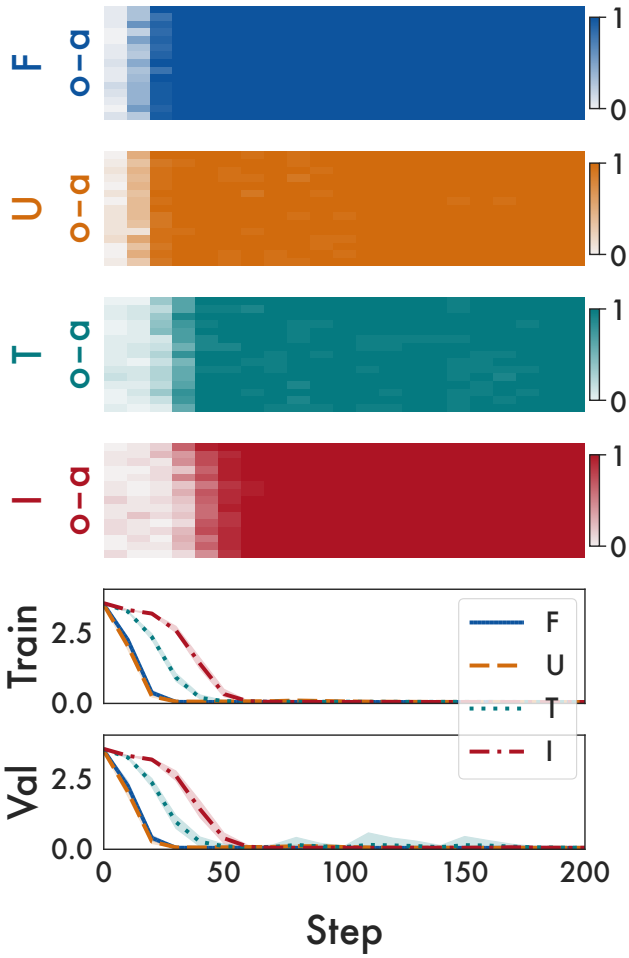
One observation we did not mention about Fig. B.1 is that `Model T` seems to use both the compiled and non-compiled attention heads. Recall that its compiled head is the k-linear map denoted by an encoding of the conditional reverse program in Fig. 1—but where the coefficients of that map are randomized (e.g., think randomly initialized matrix or tensor). It does not relearn the solution denoted by `Model F`, even though there exists a configuration of its coefficients which implement `Model F`.

B.1.5. Does compiling change the controllable representations?

An interesting detail about our encoding of characters in Model F is that it is *not* a one-hot encoding. We encode characters using the type $\oplus_{i=0}^{9} 1 = 1 \oplus 1 \oplus \cdots \oplus 1$. This type can exactly implement an alphabet of 10 characters using one-hot encodings. But for an alphabet of 26 characters, our encoding uses the first coordinate to represent whether a character is a vowel, and the other 9 coordinates are essentially left to the neural network to decide how to use. An important consequence of our steering results in Fig. 9 is that transformers can learn to respect this encoding.

B.2. Reverse

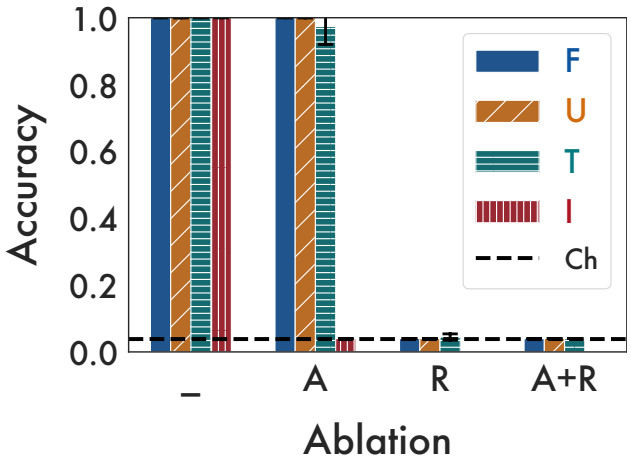
The findings of this experiment are consistent with what is observed in the main paper. There are two main distinctions worth noting. First, Model F is now compiled from a reverse program, essentially Model P from the main paper. Moreover, there is a different ablation Model U, which is essentially Model F but where we *unfreeze* the coefficients of its k-linear map. We were interested in whether gradient descent would decide to preserve the compiled circuit, or opt to rewire it into something else. For this experiment, it appeared to preserve the compiled circuit. But in subsequent experiments (e.g., rotate and conditional rotate) Model U exhibited extremely unstable learning dynamics.



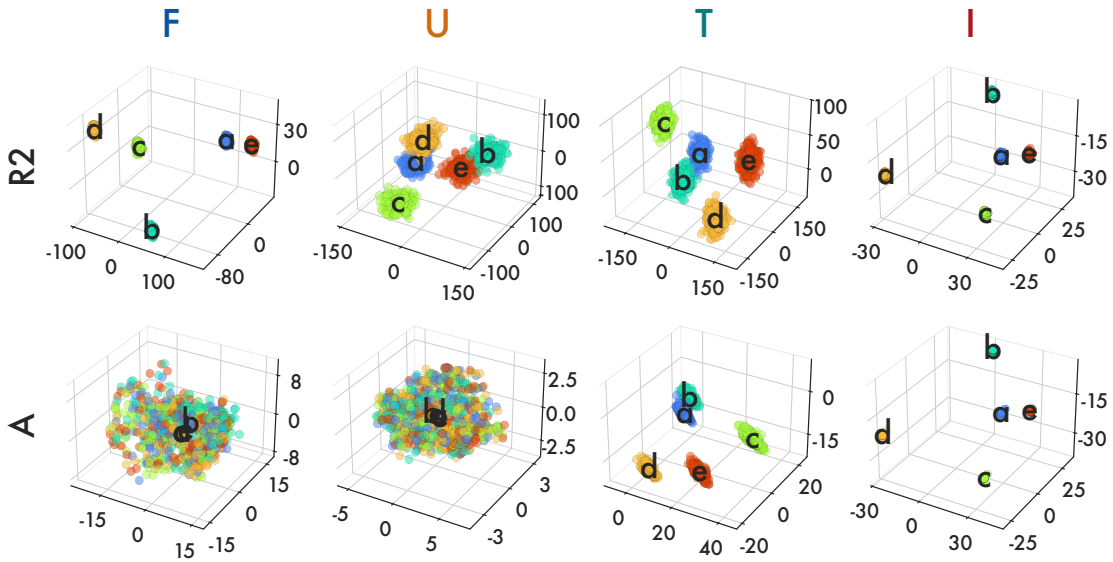
(a) Behavioral Dynamics

qahftrxckafnafq|qfanfakcxrtfhaq
ofpvausieyiccwp|pwcciyeisuavpfo
usnzjovqwpsbfhc|chfbspwqvojznsu
gchqjjfgyqpesej|jesepqygfjjqhcg

(b) Task



(c) Resample Ablations



(d) PCA on Representations

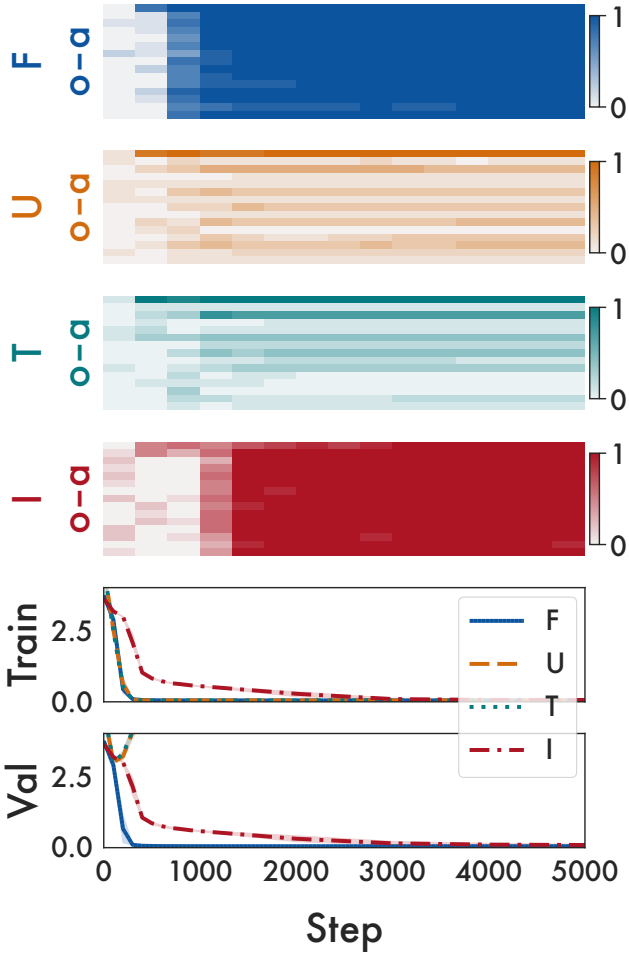
Model	Representation	PC1	PC2	PC3
F	R2	46.8%	35.5%	8.6%
	A	55.4%	22.5%	6.2%
U	R2	29.0%	27.5%	22.2%
	A	56.7%	13.0%	10.5%
T	R2	42.0%	38.3%	15.0%
	A	35.0%	32.0%	15.9%
I	R2	26.8%	21.5%	20.8%
	A	26.9%	21.6%	21.0%

(e) Variance explained by PCs

B.3. Rotate

This experiment concerns an entirely new task: rotate. The transformers must learn to shift the input sequence to the left n times, where n is the position of the first character in the alphabet (0-indexed). For example, a string starting with a is just identity because a is in the 0-th position in the alphabet. A string starting with b moves to the left once (e.g., bac becomes acb). This task is more complicated than reverse, so each model is a two-layer transformer with the standard ordering, attention layers coming before MLP layers. The compiled attention head encoding rotate is only present in the first layer of the transformer.

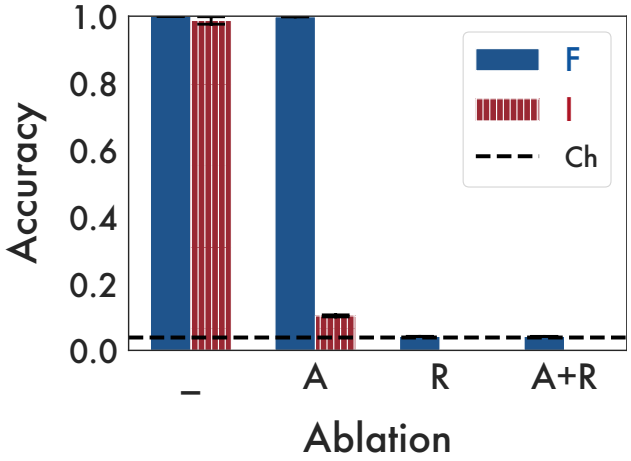
Because Model T and Model U do not learn the task, our analyses focus on Model F and Model I.



(a) Behavioral Dynamics

qahftrxckafnafq|ahftrxckafnafqq
ofpvausieyiccwp|pofpvausieyiccw
usnzjovqwpsbfhc|ovqwpsbfhcusnzj

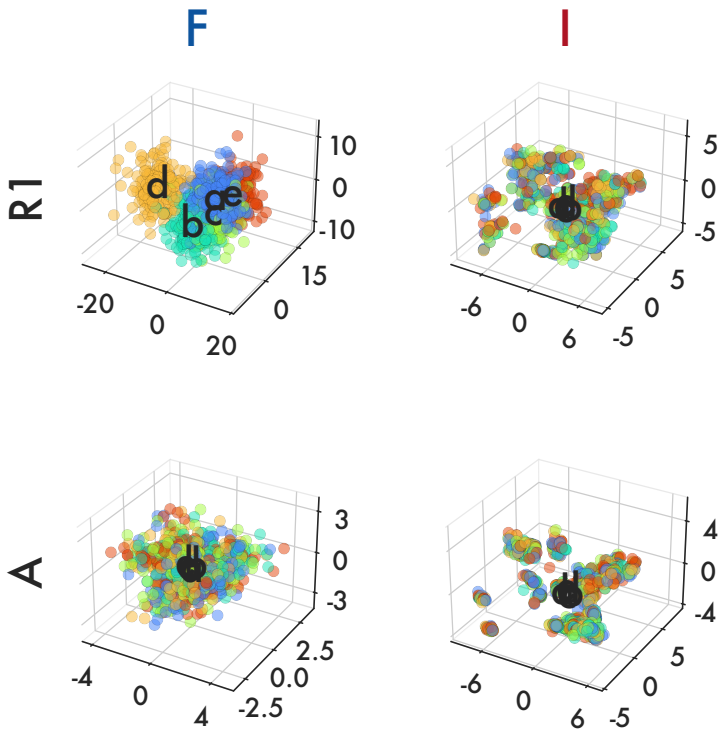
(b) Task



(c) Resample Ablations

Model	Representation	PC1	PC2	PC3
F	R1	12.7%	10.8%	10.0%
	A	30.9%	20.8%	16.5%
I	R1	22.6%	12.7%	9.5%
	A	38.6%	21.4%	15.3%

(e) Variance explained by PCs

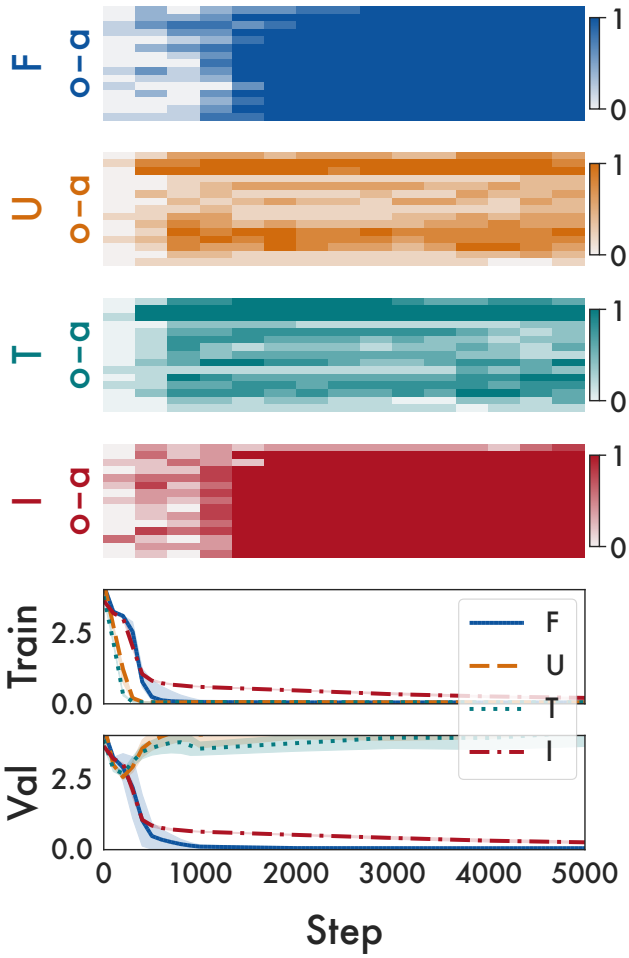


(d) PCA on Representations

B.4. Conditional Rotate

The final experiment is the most difficult. It is a variant of the previous rotate task. If the last character is a vowel, you rotate left, and otherwise rotate right by n positions where n is the position of the first character in the alphabet (0-indexed). Model architectures are similar to the rotate experiment. However, this time a program encoding the conditional rotation is compiled into the attention head of the first transformer layer.

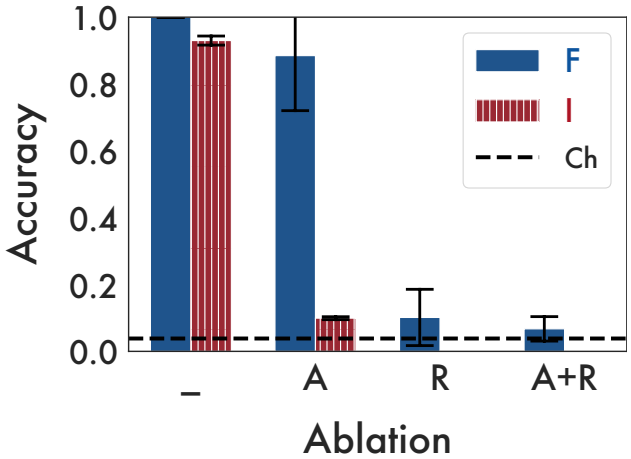
Because Model T and Model U do not learn the task, our analyses focus on Model F and Model I.



(a) Behavioral Dynamics

qahftrxckafnafq|qqahftrxckafnaf
ofpvausieyiccwp|fpvausieyiccwpo
usnzjovqwpsbfhc|sbfhcusnzjovqwp

(b) Task



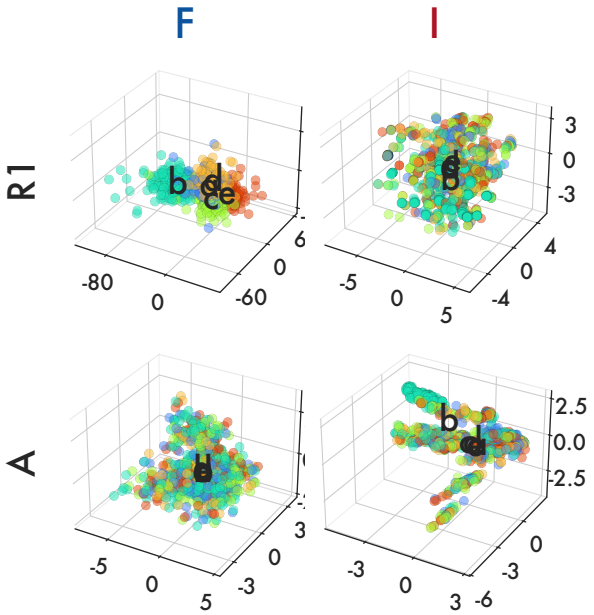
(c) Resample Ablations

C. Implementation

We implemented Cajal in Python, embedding it within PyTorch. Every Cajal program compiles to a PyTorch module implementing the k-linear map. These are then freely used within the transformers in our experiments. The implementation of Cajal and its compiler is transliterated directly from the formal specifications in this paper.

Our implementation uses property-based tests (MacIver et al., 2019) to check that the implementation satisfies key theorems from this paper (i.e., that programs have behavior, and that compiling preserves their behavior). These tests essentially randomly sample Cajal programs and check that these properties hold of sampled programs.

The supplementary material contains an artifact which includes our implementation of Cajal and all code used to run and analyze experiments. We have packaged the artifact as a Docker container for reproducibility. On running the artifact (see README), the entrypoint will be a Jupyter Notebook walking through examples of how to use the code.



(d) PCA on Representations

Model	Representation	PC1	PC2	PC3
F	R1	14.8%	12.6%	11.1%
	A	39.2%	15.6%	11.9%
I	R1	14.8%	10.2%	9.2%
	A	32.8%	23.4%	14.1%

(e) Variance explained by PCs