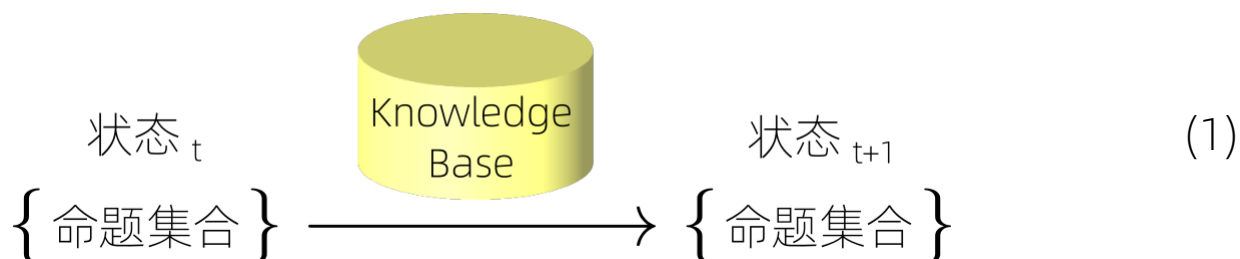


①

逻辑与深度学习的关系

这是经典逻辑 AI 的最基本运作模式：



它其实包含了两个算法：

- **matching** (unification):
逻辑 rules 是包含变量的条件命题，
例如 $\forall x. \text{是人}(x) \Rightarrow \text{会死}(x)$.
Unification 判定一条 rule 是否可以 apply 到某逻辑命题上，
例如：是 **人(苏格拉底)** 可以跟上式的左边 unify.
Matching 的结果是得到一推 instantiated（特例化，即不包含变量）的命题。
- **forward- or backward-chaining** (resolution):
由已知事实 推导出新结论，或反过来，判断某给定的新结论是否成立。
例如：是 **人(苏格拉底)** \Rightarrow 会死 **(苏格拉底)** \wedge 是 **人(苏格拉底)**
可以推出：会死 **(苏格拉底)**。

深度学习的特点，就是将

$$\text{状态}_t \vdash \text{状态}_{t+1} \quad (2)$$

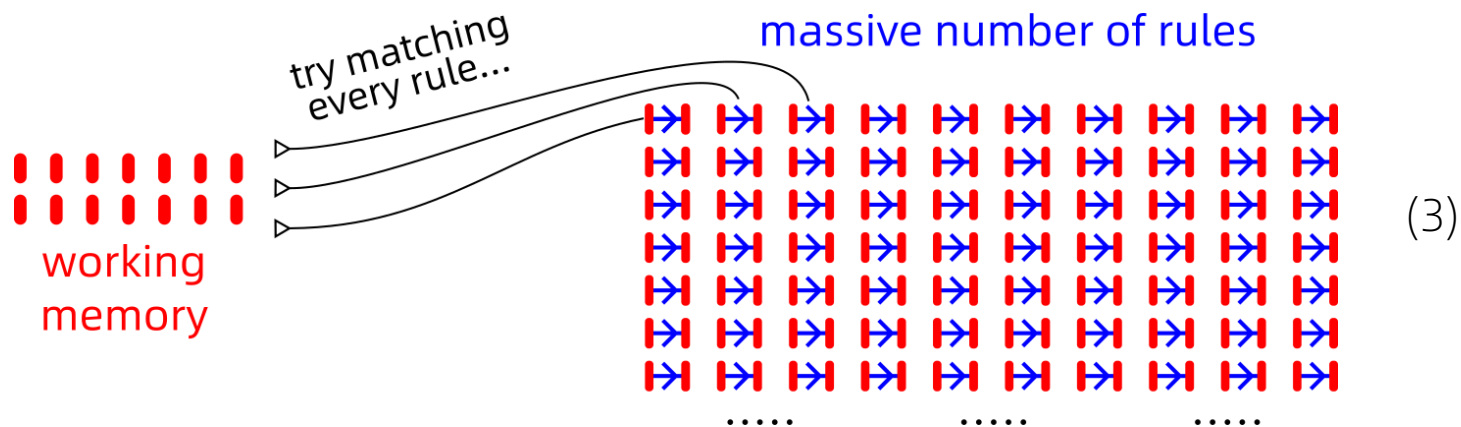
的逻辑推导过程，通通纳入进去一个非常复杂的非线性函数（= 深度神经网络）里面。这样做以后，上述的逻辑结构被“mingled”在一起，以至于很难分辨了。但也正是由于这种「大杂烩」，深度神经网络 将一套复杂的组合算法压缩成数量不算太多的一层层参数。它同时可以做 learning 和 inference 这两个动作。这种简单粗暴的方法，其实非常有效率，要超越它的速度并不容易！

我们知道（或推测）一个智能系统 应该具有 符号逻辑的结构。这点知识可不可以用来 约束/加速 深度神经网络？答案似乎是有可能的。现时 state-of-the-art 处理视觉的 CNN 和处理文字的 GPT，它们都有内部结构，**而不是 fully-connected**，而且这内部结构 对应于 被处理的资料的结构。因此我们有理由相信，逻辑结构 可以用来约束 深度神经网络的结构，达到加速。

②

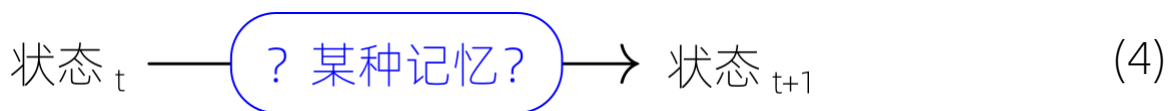
接下来我们详细一点看逻辑系统的结构：

Knowledge Base 里面有很多 rules，系统要将这些 rules 逐一 match with 系统状态 (= working memory) 里面的命题：

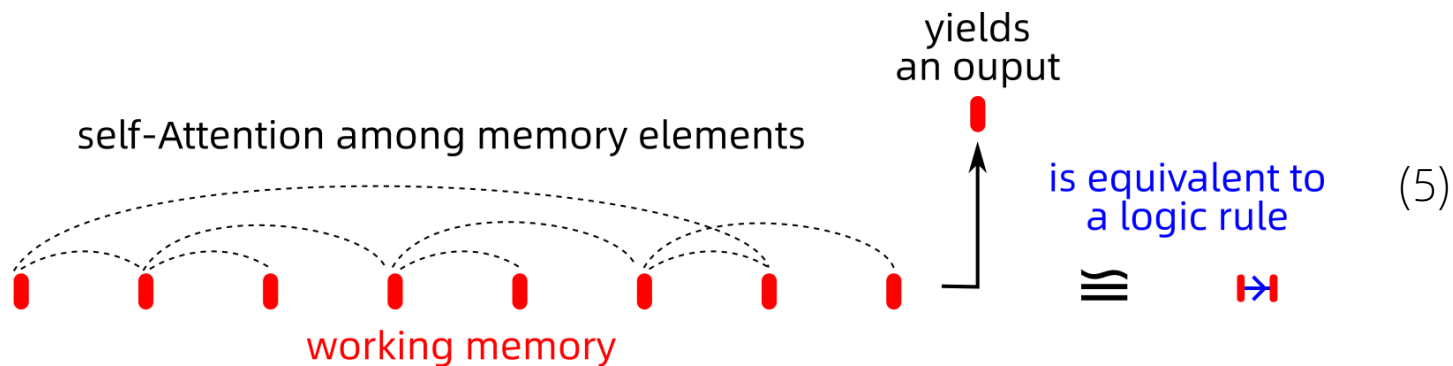


成功 matched 的 rules 可以导出新的结论，加进 working memory 的状态 里面。

这个复杂的操作，完全被一个神经网络取代。或者可以更抽象地说：



而以 Transformer 来说，它是一种 输入元 之间 的记忆体（这记忆就储存在 Q, K, V 矩阵里），而它 **implicitly** 做到了 rules 的作用：

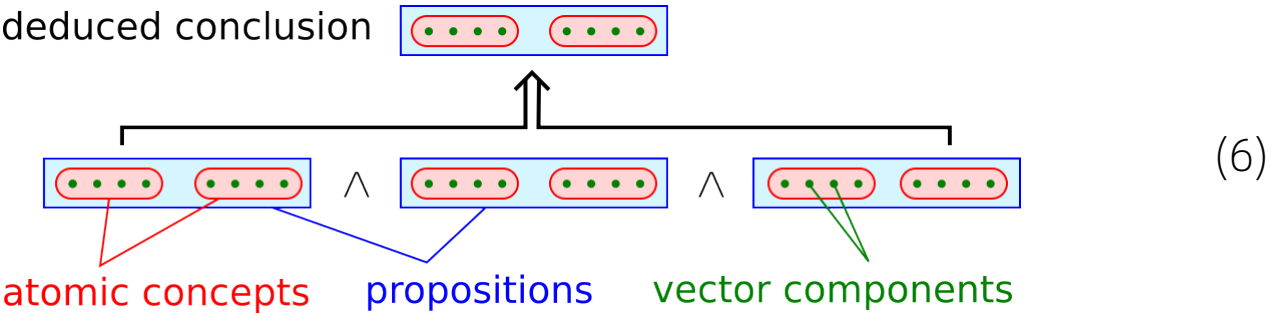


换句话说，Transformer 内部有某种（扭曲了的）逻辑 rules 的结构。那么很自然的问题就是：能否发掘更多 逻辑 / 逻辑系统 的结构？也就是说，公式 (4) 可以有怎样的代数结构约束？这个问题 可以参考 范畴逻辑 的理论，还有 经典 logic-based AI 系统的理论。

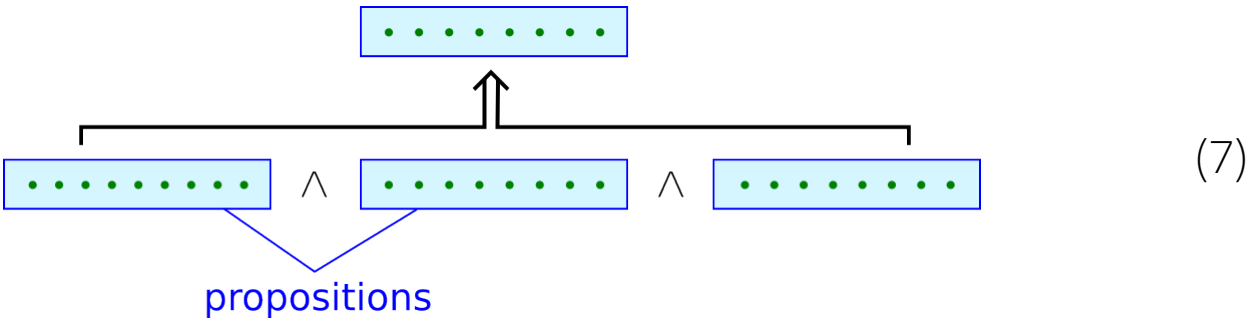
我们希望 勾画出公式 (4) 需要具备的代数约束，但暂时先用文字描述比较容易：


- 状态是 **颗粒化** 的，它是某集合的元素，元素之间可交换，也就是 Transformer 的 equivariance. （注意：Transformer 有 equivariance, 但 equivariance 未必一定要用 Transformer 实现）
- **深度结构**：例如多层网络，每层是函数的复合 (composition). Transformer 也用了深度结构。
- 逻辑 包括了 **命题**层次 和 **命题内部**层次的 颗粒化。后者是**谓词** (predicate) 逻辑的结构，例如：*loves(John, Mary)*，也可以简单地将它视为 **代数元**之间的**乘积**，例如：*John · loves · Mary*, 后者也叫做 “word”. （不同类别的代数元之间不一定容许乘积，因此有 groupoid 的概念，但暂时来说这细节不重要。）现时重点是如何将 这两层的 颗粒化 结构 施加到深度神经网络上。
- 逻辑推导 每步只产生**一个**新的结论（或其概率分布），然后这个新的结论，再加入到旧的状态中，作为一个命题集合的元素，而旧状态也要 **遗忘** 一些命题，否则需要无限记忆。这跟 Transformer 每次输出一列的 tokens 有点不同（虽然我们不太肯定 Transformer tokens 究竟对应于命题 还是 谓词 / 原子概念）。
- 逻辑 rule 通常只跟某几个前提有关，其它前提是**无关**的，例如：**眼睛好看** \wedge **鼻子好看** \wedge **嘴巴好看** \Rightarrow **帅**，跟 **有钱** 或 **穷** 无关。Transformer 的 **softmax** 结构似乎也可以排除一些无关的 tokens 的影响。
- (可能还有其他的结构特征.....)

根据我的理论，理想的逻辑形式是这样的（各种元素的个数纯粹示意）：

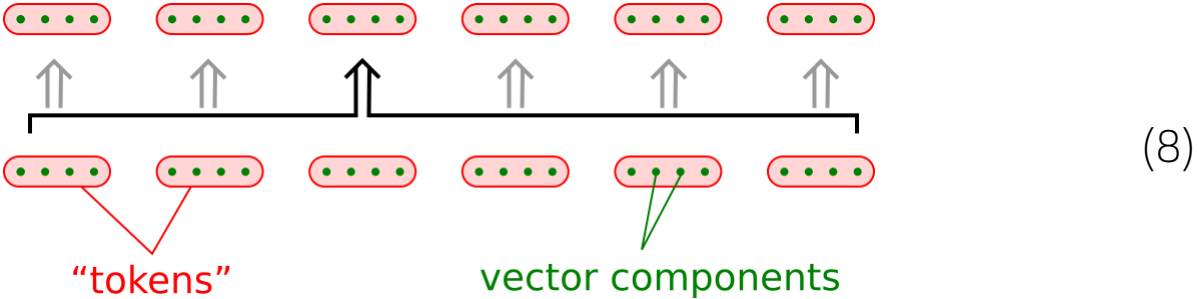


相比之下，我们目前可以写出来的 代数关系 $p \wedge q = p \wedge q$ 只表达了这种结构：



相比于 图 (7)，图 (6) 添加了  的约束，但这约束怎样用代数表示？

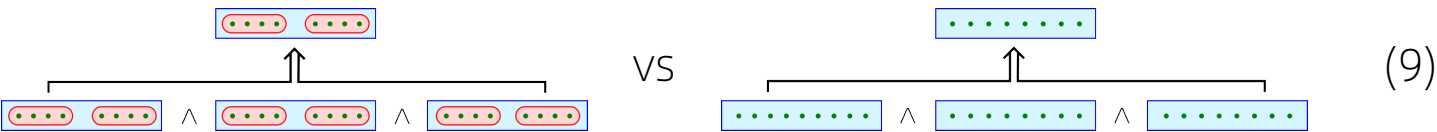
而 **Transformer** 处理 命题 和 概念 的方式是这样：



它没有 explicit 的命题结构，而是用特别的 token 表示句子的**终结**，当然还有 positional encoding 这些「伎俩」。所以，Transformer 是一种比较 *ad hoc* 的设计，我们应该可以改进它。

现在我们企图回答那最重要的问题：怎样用代数形式表达「命题是由概念原子组成的」？

亦即是说，以下这两个结构的分别在哪里？如何用代数表达这不同？



这就像问 $0...9 \times 0...9$ 跟 $00...99$ 的分别（基本上没有分别，它们是 isomorphic）。

类似地，

$$\{ \text{John, Mary} \} \times \{ \text{human, god, worm} \} \tag{10}$$

跟

$$\{ \text{John is human, Mary is human,} \} \tag{11}$$

等 $2 \times 3 = 6$ 个命题 也是同构的。但前者是由两组不同的概念结合而成，它的成分可以被 \forall 或 \exists 量化；后者是 命题逻辑，那些命题是不可分割的，也不可以拆开来量化。

但由于 $\dots \in$ 非交换自由群（最少结构的群），它没有像 $a \cdot b = b \cdot a$ 那样的对称性公式。

经过一番分析之后 我得到了「命题是由概念原子构成的」以下条件：

Atomic Condition (AC). *Each proposition P_i is made up of K atoms:*

$$P_i = a_{i1} \cdot \dots \cdot a_{iK} \tag{12}$$

where optionally some atoms can be copied to other locations (with a non-linear transformation τ , if they are copied to the output layer) via:

$$a_{ih} = a_{jk} \quad \text{or} \quad a_{ih} = \tau(a_{jk}) \tag{13}$$

and the transformation τ has to accord with \forall or \exists as adjunctions to a substitution functor.

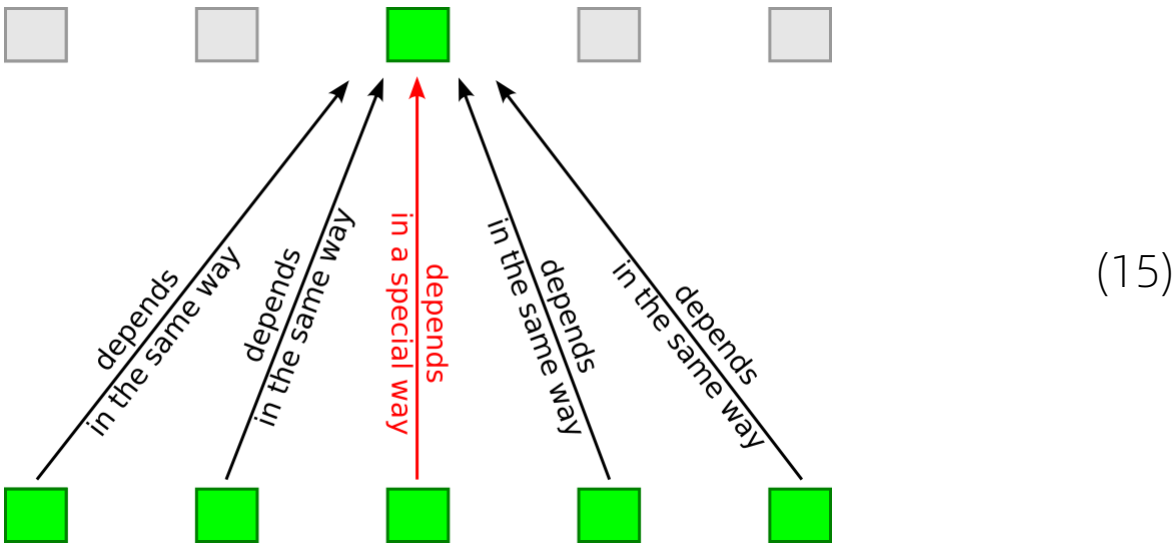
其实 τ 只需要是连续函数，就可以符合上述条件。所以 Atomic Condition 的重点在于 (12) 和 (13) 这两个等式，其实是非常简单的。 \forall 和 \exists 作为伴随函子的范畴论描述 比较复杂，我们在 附录 A 里解释。

那么 等式 (13) 里面的 “=” 是来自哪里？其实太明显了，它就是逻辑 rule 里面将变量「syntactically 搬动」的动作：

$$\forall X, Y, Z. \text{ grandfather}(\textcolor{red}{X}, \textcolor{red}{Z}) \leftarrow \text{father}(\textcolor{red}{X}, \textcolor{red}{Y}) \wedge \text{father}(\textcolor{red}{Y}, \textcolor{red}{Z}) \tag{14}$$

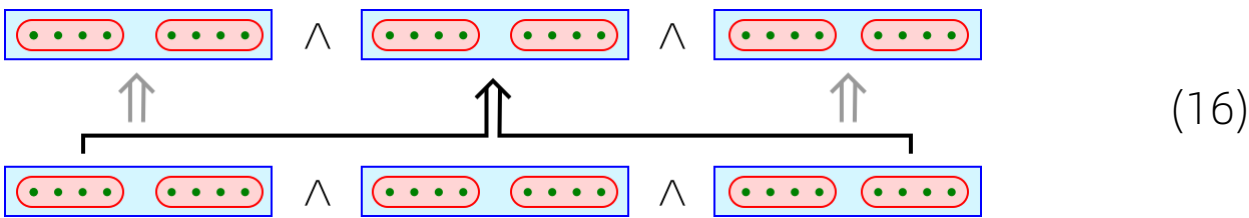
正是 这些「搬动」，构成了「命题是由概念组成的」结构。

Self-Attention 的本质 可以这样理解（抽象注意力结构）：



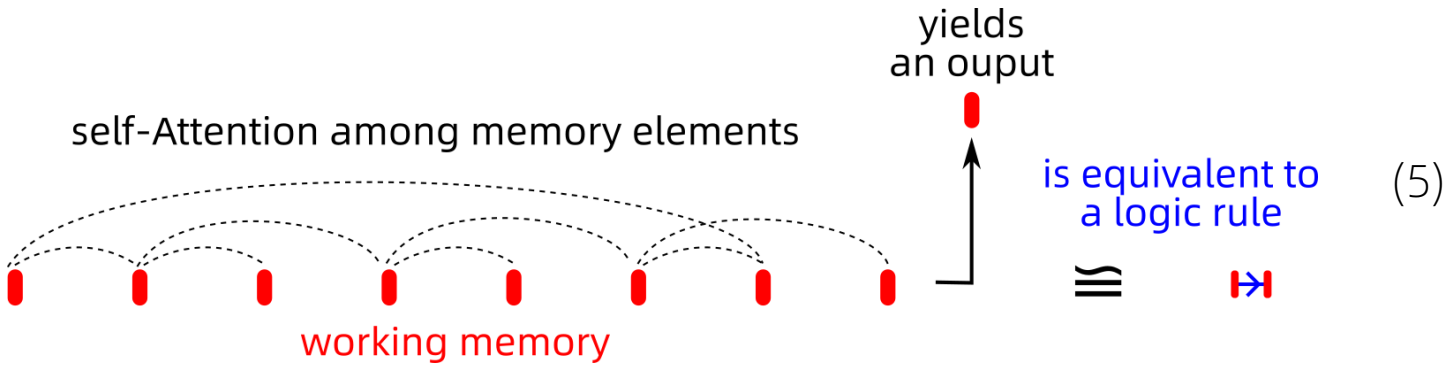
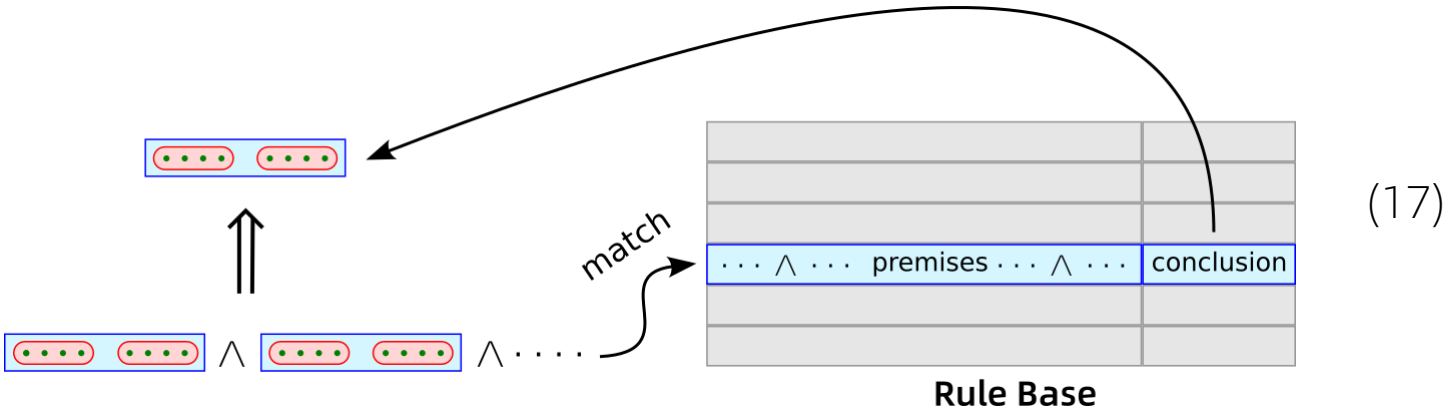
这垂直的「軸」结构 (红色), 重复在每一条轴上。所以, 当 输入的元素 交换时, 输出也随着交换。这就是 self-Attention 能达到 **equi-variant** 效果的原因。

而我们想用 类似以上 self-Attention 的方法, 解决 逻辑结构 的问题：



这里有一个很重要的重点¹：self-Attention 的前身是来自 Graves *et al.* 的《Neural Turing Machine》的 **content-addressable memory**. 我们很有理由将它看成是一种 **记忆体**。

我们要比较 两种做法, 前者是简单直接的经典 rule base 结构, 后者是以 self-Attention 代替 rule base：



大家要感受到 Transformer 是一种非常 “twisted” 的处理 rules matching 的方式。这个对应很不明显, 以至于我们很难分辨出 Transformer 那边的 rules 长什么样子。然而我觉得 Transformer 的设计者们 或许多少有意识到它跟 rule-based systems 的相似性。特别地, 看看以下这条逻辑 rule：

$$\forall X, Y, Z. \text{ grandfather}(X, Z) \leftarrow \text{father}(X, Y) \wedge \text{father}(Y, Z)$$

(14)

这 rule 的前提有两个条件, 出现两次的变量 Y 必须相等 (红色), matching 才算成功。而这种在 rule 的前提内部进行的 **比较** (comparison) 运作, 正是 self-Attention 可以方便地做到的。但 self-Attention 也忽略了 $A \wedge B$ 的对称性, 可能还有改进的空间。

¹谢谢 “子鱼” 告诉我这个重要信息。

我觉得目前要回答的几个关键问题是：

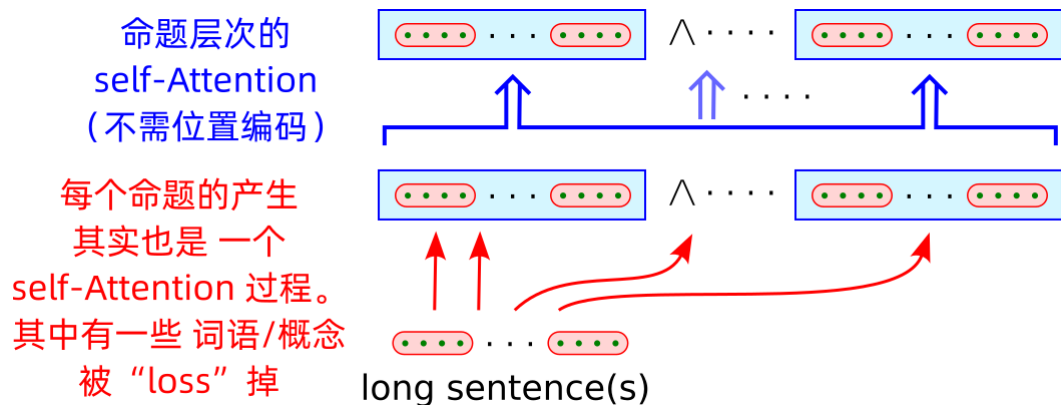
- 根据例如 刘乾 *et al* 的论文¹，Transformer 做不到某些 逻辑语法上的运作，问题出在哪里？似乎不是 Transformer 根本无法学习那种语法，而是它不能纯粹靠 prompt 做到。但其实 prompt 是否具有深层意义，还是它只是一个 hack？我们没有 explicitly「告诉」Transformer 它应该怎么做，那它做不到是不是一个真的缺憾？我觉得很难判断，令 prompt 的研究方向笼罩在迷雾之中。
- 现在考虑图 (17) 即 rule base 的 naïve 学习算法。这个算法当然是很慢的，因为要比较两个集合（working memory 和 rule head）的相似性。假设两个集合的大小固定为 N ，那需要 $N \times N$ 次的 dot products，而这只是比较了一条 rule。所有 rules 还要用 softmax 相加。当 rule base 很大的时候，这个算法似乎不太实际。
- 图 (17) 还有可能出现 旧有的逻辑 rule learning 算法的问题，亦即“plateau problem”。举例来说，用 Prolog 语言写 append 函数：

```
append(X,Y,Z) :-  
    list(X), head(X,X1), tail(X,X2), append(X2,Y,W), cons(X1,W,Z).
```

这个 rule 有 5 个前提。当 rule 被学习时，前提被逐个加进去，但 rule 的「得分值」一直是零，直到最后的前提加进去了，得分才突然升到 100%。对于机器学习来说，这情况是很糟的。而 Transformer 将 rules「扭曲地」缠在一起，这做法会不会反而有利于避免困在 local minima 呢？
- 可能存在介乎 Transformer 与 naïve rule base 之间的算法，它比 Transformer 有更强的逻辑结构，但比 naïve rule base 用了更多类似 self-Attention 的矩阵运算来加速？

¹Qian Liu, Shengnan An, Jian-Guang Lou, Bei Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. *Compositional Generalization by Learning Analytical Expressions*. Advances in Neural Information Processing Systems 33 (2020).

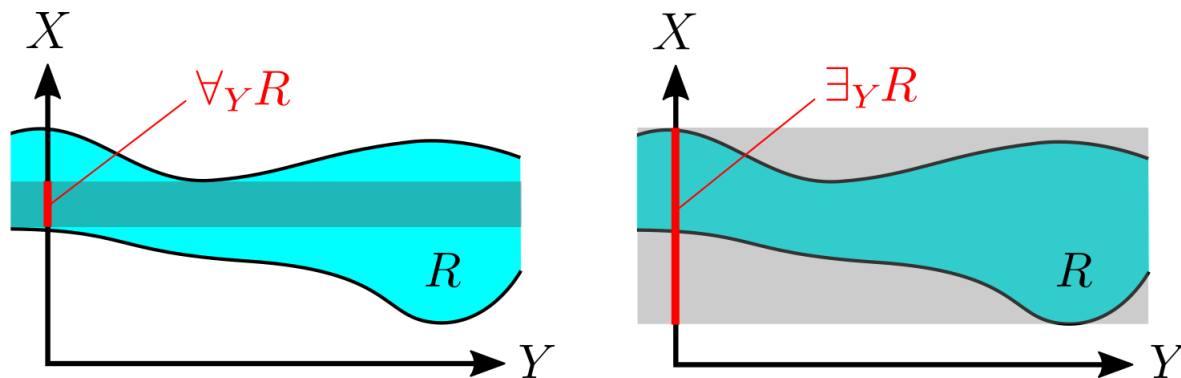
现时想到的一个比较可行的 architecture:



- 在蓝色结构里，每个命题要由 fixed # of 概念原子组成。
- 但抽象的 self-Attention 结构也可以容许 可变长度 的命题，只要能计算到 命题之间的 dot product (similarity)，还有 读取 固定长度的 矩阵 记忆体。
- 红色结构：如何从 不定长度的句子 产生数量和长度不同的命题？这就是以前谈过的 sequence to sequence-of-sequences 问题。这也可以用 “lossy” self-Attention 解决。

在这个附录里，我们用简单的方式解释一下 \forall 和 \exists 的理论。

以下是一个关系 R ，例如「 Y 爱 X 」， X, Y 都是「人」的集合；两个相同的拷贝。例如 对角线 代表爱自己（有些人不爱自己）。注意这个图也不是 对角线 对称的，否则就没有「失恋」了。将关系 R 的图像 **投影** 到 X 轴上，可以得到 $\forall_Y R$ 和 $\exists_Y R$ 集合。 $\forall_Y R$ 集合代表那些「人人都爱」的 X ， $\exists_Y R$ 集合代表那些「有人爱他」的 X ：



(19)

范畴论大师 Lawvere 发现， \forall 和 \exists 是一个所谓 **weakening functor** 的「伴随函子」，所谓 weakening 就是从一个只有 X 变量的**论域** 扩充到有 X, Y 两个变量的论域，而这里 Y 纯粹是一个 **dummy** variable:

$$\begin{array}{ccc}
 & \xleftarrow{\forall_Y} & \\
 (X) & \xrightarrow{\text{weakening}} & (X, Y) \\
 & \xleftarrow{\exists_Y} &
 \end{array}
 \quad (20)$$

例如 $\text{Love}(Y, X)$ 是带有两个变量的逻辑式子，但 $\forall Y. \text{Love}(Y, X)$ 其实没有了 Y 这个变量，因为它被 \forall **绑定**了。

所谓 **伴随** (adjoint) 的意思是：有两个范畴：左边和右边。你可以将左边的东西搬到右边，在右边的范畴里做「**比较**」，而这个比较 同样地可以 把东西搬到左边做，两个比较是**等价**的。这里「比较」的意思就是范畴内的 morphism，例如在 **Set** 范畴里，比较就是 set inclusion.

伴随函子 并不是唯一的，所以 weakening 分别有 \forall 和 \exists 两个伴随。

Lawvere 将 量词 \forall 和 \exists 推广得更一般：「简单」的 weakening functor 是基于 笛卡尔积 $X \times Y$ ；Lawvere 将它扩充到任何 **substitution** functor. 但我暂时缺乏这方面的例子，不清楚它对我们的应用有什么益处。

在 范畴逻辑 里面有 **Beck-Chevalley** 条件和 **Frobenius** 条件，或许是我们所需的对称性？但细看之后，发觉还是不能解决问题..... For completeness, 我还是描述一下，没兴趣的可以略过。

首先考虑比较容易明白的 **Frobenius** 条件。在逻辑上，它等于说：

$$\exists x. [\phi \wedge \psi(x)] \equiv \phi \wedge \exists x. \psi(x). \quad (21)$$

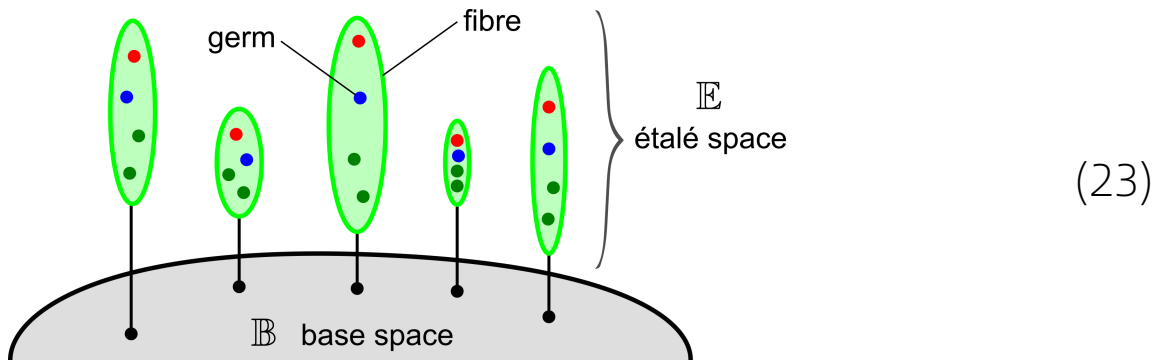
由于 经典逻辑 AI 普遍使用 \forall 而忽略 \exists ，我将上式改写成：

$$\forall x. [\phi \vee \psi(x)] \equiv \phi \vee \forall x. \psi(x). \quad (22)$$

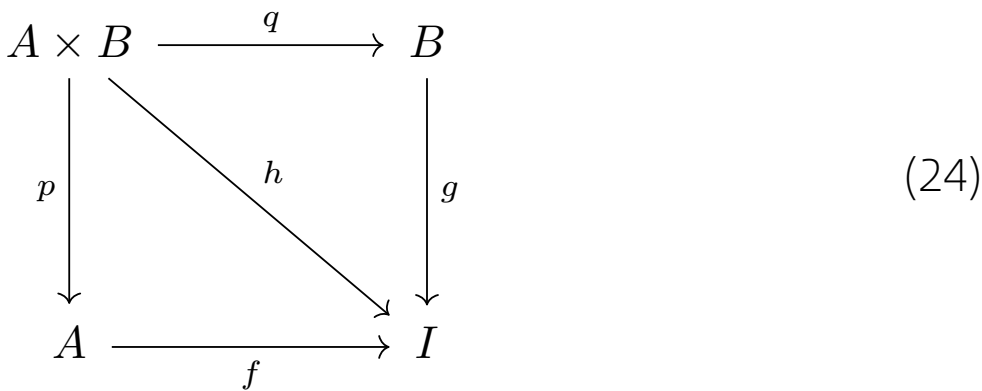
但问题是，(22) 式的左边和右边，其对应的神经网络 (6) 是一样的（看不出有分别）。也就是说这个差别可能太 subtle 了，它并不影响我们实际 implement 的神经网络。

以前说过，谓词逻辑 带来 **fibration** 或 **indexing** 结构。Beck-Chevalley 和 Frobenius 条件 基本上是说，这 纤维结构 是 “preserved by re-indexing functors”。

这是 fibration 结构的示意图：

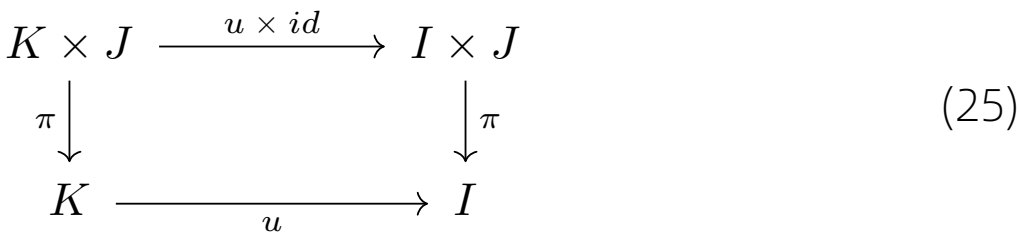


这整个结构 叫 **bundle**，而 **sheaf** 是 bundle 加上某个特殊的 拓扑结构。
 在 (A, f) 和 (B, g) 两个 bundle 之上可以定义 **fibred product** of A and B over I , 记作 $A \times_I B$:



其中 $h = f \circ p = g \circ q$. 这也是一个 **pullback**.

Beck-Chevalley 条件是说 下面这幅图 commute:



其中 π 就是代表 量词 \forall 或 \exists 的 投影，它们是 weakening map π^* 的伴随映射。

Beck-Chevalley 条件并不完全是空洞的；它有可能不成立。有一个反例是 Pitts 提出的：考虑 $X \times Y$ ，其中 $X = Y = \mathbb{N} \cup \{\infty\}$ 亦即 自然数加上 ∞ 作为 top element；但 Y 是用 discrete order，亦即所有 order 都是 $=$ 关系。 A 是 $X \times Y$ 上的关系： $A = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq y\}$. 那么 $\exists y.(x, y) \in A$ 会是整个 X 集合。如果考虑 DCPO 范畴，我们要求 fibration of Scott-closed subsets (ordered by inclusion) over DCPO. $\exists y.A$ 的 Scott closure 的条件是 它是一个 lower set closed under directed joins; 而这个 Scott closure 条件似乎不成立，因而导致 图(25) 不 commute. (我对 Scott closure 的细节不太理解)

首先以函数的方式表达 self-Attention 结构：

输出命题 O_i 由原子 b_i 组成

$$O_i = [b_1 \dots b_K] \tag{26}$$

输入命题 P_i 由原子 a_i 组成

$$P_i = [a_1 \dots a_K] \tag{27}$$

Self-Attention

$$O_i = \alpha(P_i ; P_1 \dots \hat{P}_i \dots P_N) \tag{28}$$

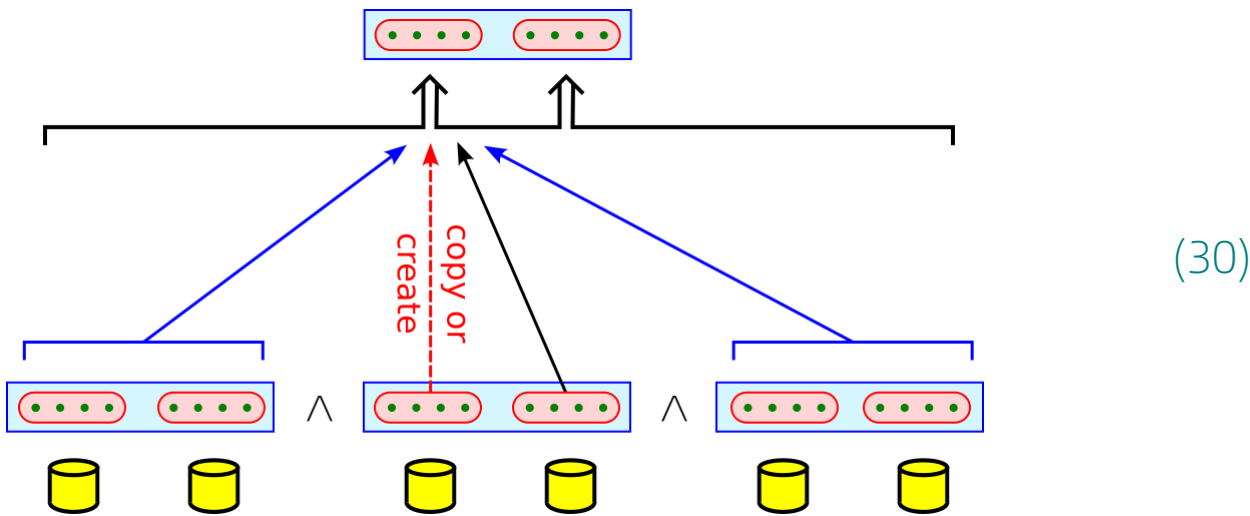
$P_1 \dots \hat{P}_i \dots P_N$ 的意思是 $P_1 \dots P_N$ 除了 P_i .
 $\alpha(P_i ; \dots)$ 是图 (15) 的函数结构，也可以理解为以 P_i 为 query 的 self Attention.

How to measure similarity between (P_i, Q_i) ?

$$\arg \min_i \sum_i \min_j \langle P_i, Q_j \rangle \tag{29}$$

- 首先留意到，轴心结构 只对 命题层次有用，它的作用不会延伸到 概念原子层次。然而，有鉴于 Transformer 也没有充分利用 交换不变性，然而它却因为用了矩阵乘法而变得很有效率，所以从效率的角度看，也有将 轴心结构 延伸到 原子层面的理由。
- 另一个可以尝试的想法是：直接 hard-code copying mechanism. 这可以怎样用 Attention 做到？以前分析过了，copy 并不容易，因为需要 winner takes all.
- 但纯粹用 Hopfield network 又缺乏了 深度。但似乎在 RL 场景下，**广度**也是重要的。
- 其实 只要有 输入/输出 的函数关系，就等价于有 逻辑 rules 的 KB 库。问题是它用什么方法给出结论。

Self-Attention 已经符合我们的要求，但我们想改进它。主要有两个 idea：其一是更直接的 copy mechanism，其二是 更细致的 函数 dependence.



Copy:

- Copy 不需要什么特别机制，输出就是输入。但问题是怎样 combine with “create” operation.
- 当然 可以用大家都熟悉的 softmax: $\alpha \text{ copy} + \beta \text{ create}$, where α, β are outputs of softmax.
- 另一个想法是 content-addressable. 用 table-lookup 的形式，找可执行的 rules。但有 variable matching 的问题。

Create:

- 需要的是怎样的函数？