

Hey Pentti, We Did It Again!:

Differentiable vector-symbolic types that prove polynomial termination

Eilene Tomkins-Flanagan (eilenetomkinsflanaga@cmail.carleton.ca)

Connor Hanley (connorhanley@cmail.carleton.ca)

Mary Alexandria Kelly (mary.kelly4@carleton.ca)

Department of Cognitive Science, Carleton University
1125 Colonel By Drive, Ottawa, ON, K1S 5B6, Canada

Abstract

We present a typed computer language, Doug, in which all typed programs may be proved to halt in polynomial time, encoded in a vector-symbolic architecture (VSA). Doug is just an encoding of the light linear functional programming language (LLFPL) described by Schimanski (2009, ch. 7). The types of Doug are encoded using a slot-value encoding scheme based on holographic declarative memory (HDM; Kelly, Arora, West, & Reitter, 2020). The terms of Doug are encoded using a variant of the Lisp VSA defined by Tomkins-Flanagan and Kelly (2024). Doug allows for some points on the embedding space of a neural network to be interpreted as types, where the types of nearby points are similar both in structure and content. Types in Doug are therefore learnable by a neural network. Following Chollet (2019), Card, Moran, and Newell (1983), and Newell and Rosenbloom (1981), we view *skill* as the application of a procedure, or program of action, that causes a goal to be satisfied. Skill acquisition may therefore be expressed as *program synthesis*. Using Doug, we hope to describe a form of learning of skilled behaviour that follows a human-like pace of skill acquisition (i.e., *substantially faster than brute force*; Heathcote, Brown, & Mewhort, 2000), exceeding the efficiency of all currently existing approaches (Kaplan et al., 2020; A. L. Jones, 2021; Chollet, 2024). Our approach brings us one step closer to modeling human mental representations, as they must actually exist in the brain, and those representations’ acquisition, as they are actually learned.

Keywords: polynomial time type system; representation learning; vector-symbolic architecture

Chollet (2019) proposed a novel model for intelligence: an *intelligent agent* does not just *solve problems* in its environment in service to its goals (as we might infer it does in the Markov decision process model found in *reinforcement learning*, Sutton & Barto, 2018, pp. 46-53). Instead, an intelligent agent *acquires the skills necessary to the task*, and uses its *acquired skills* to solve problems in service to its goals. Chollet’s model of “skill” is an extremely general one. In his view, a skill is a *program*: a procedure made up of well-defined steps, which, when followed, will transform the environment from an initial state into a desired state. By separating the intelligent agent into an *intelligent system* that generates skills, and *skill programs* that realize the agent’s competencies, Chollet frames the problem of intelligence as one of an ability to acquire skills from sparse examples in a great variety of problem types. Following Chollet, a *greater intelligence* within some scope of problems is one that can acquire the skills to solve problems in that scope with a minimum of foreknowledge and training, and a *more general intelligence* is intelligent, up to some degree, in a greater variety of scopes, more exhaustive of the space of possible problems.

The Chollet model may have a familiar ring for cognitive modelers, however. The *Goals, Operators, Methods, and Selection rules* (GOMS; Card et al., 1983, ch. 5) model describes skilled behaviour as a deployment of procedures that will tend to satisfy an agent’s goals. In GOMS, an agent with some *goal G* uses its *selection rules* to choose a *method* that, the agent expects, will cause *G* to be satisfied, if followed. A *method* is a procedure in the sense we used above: a sequence of well-defined steps. In the case of GOMS, the steps are *operators*, discrete actions the agent can take that transform the state of the environment, or the agent’s internal state. Following the procedure laid out by the selected method, the environment is transformed into a state whereby the goal *G* is satisfied. Newell (1990) situates GOMS within the *Soar* cognitive architecture, stating that GOMS “posits a set of mechanisms and structures” (p. 285) that can be formalized with the *Soar* architecture, and goes on to express skill acquisition in *Soar* (ch. 6.5) as a kind of search for increasingly efficient procedures that satisfy goals.

Chollet refers to the problem of acquiring skills as “program synthesis” (p. 30), that is, search for a procedure, with some given effect. Hutter (2005, ch. 1.5) showed that synthesizing an optimal program that *predicts or represents* the environment allows an agent to behave in an optimally goal-directed (i.e., rational) way, with minimal extra machinery. In this way, the problems of *skill acquisition* and *representation learning* can be thought of as transformable into one another.

Hutter, unfortunately, also finds (ch. 1.7), that program synthesis is profoundly nontrivial. In fact, finding a program that optimally predicts an arbitrary environment is incomputable, because it involves comparing programs in light of their computational effects, and some programs cannot be proven either to eventually halt or run forever (Turing, 1936), so their computational effects cannot be known either until they halt or until the end of time, whichever comes first. Even if we restrict ourselves to considering only programs up to some maximum finite length, if we are to be certain we have found the optimal representation, we must do an ugly exhaustive search through all possible programs up to our maximum length (which is intractable, in the sense of taking an exponentially long amount of time as a function of the maximum length) and, for each program, find its shortest representation that halts within some maximum time (which is also intractable). Hutter’s finding of a doubly-exponential rate of

skill acquisition is consistent with the “power law” described by Newell and Rosenbloom (1981), however, Heathcote et al. (2000) show that their law is an artifact of population averaging, and humans in fact acquire skill *exponentially faster* than Newell and Rosenbloom believed. Unsurprisingly so, as Newell and Rosenbloom modeled skill acquisition as a simple, heuristic-informed (but in the worst case, like Hutter’s model, brute-force) search, and managed to satisfy their power law. It seems obvious that, in order to be useful, a method of program synthesis that exhibits intelligence should be substantially better than this sort of brute-force approach.

Chollet (2019)’s ARC task was designed to demonstrate the efficiency of human intelligence relative to existing AI approaches, and the importance of efficient program synthesis. A model approaching the ARC task that showed some early success was DreamCoder (Ellis et al., 2021), which used a similar method to Hutter’s, albeit synthesizing programs using a form of heuristic informed search involving a neural network. DreamCoder has since been surpassed by the GPT o-series models (Chollet, 2024), but Chollet (2025) notes that both approaches seem to behave like exhaustive search:

It’s always possible to ... logarithmically improve your performance by throwing more compute at the problem. And of course this is true for o1, but even before that it was also true for brute force program search systems. Assuming you have the right [domain-specific language], then extremely crude, basic, brute-force program iteration can solve ARC at human level. (49:55)

We would like to constrain program synthesis so that search is restricted only to programs likely to be useful to solving a given problem. If we make good assumptions about the set of programs that are to be searched over, then, for many problems, searching through the constrained set of possible programs for their solution should become tractable. As a first pass, we should prohibit from consideration intractable programs, as we do not want to bother trying to evaluate them to discover their computational effects. This prohibition is achieved using a polynomial time type system (Girard, 1998).

A *type system* is a programming language that assigns, to each expression in a program, a *type* that describes what kind of data it is, if it is a variable, and what it operates on and produces, if it is a function, accompanied with a set of *inference rules* for how to demonstrate that some expression has some type. A *polynomial time type system* is a type system that prohibits programs which do not have polynomial time complexity. A polynomial time type system is therefore not Turing-complete, but the set of problems polynomial time type systems are capable of representing still includes essentially all useful programs. The type system we will be considering is the *Light Linear Functional Programming Language* (LLFPL; Schimanski, 2009, ch. 7).

How can we make use of a polynomial time type system to constrain program synthesis? LLFPL, as an example, encodes the maximum recursion depth of a function in its type.

We envision that types should be learnable, such that a learning agent should be able to acquire the ability to guess at the sort of structure a skill program should have, before it sets to work determining the program’s specifics. The agent may need to revise its guess, but it should at least be capable of acquiring the ability to make good guesses. When a learning agent makes good guesses as to the structure of the program it should be synthesizing, its search is immediately restricted to just the programs of the appropriate structure. This set may still be quite large, but, if the appropriate structure restricts recursion depth, it is definitely much smaller than the set of all programs, or even all well-typed programs in the type system.

Our goal, then, is to make LLFPL *learnable by a neural network*. To do so, we will need to make it possible to express types as *points in a vector space*, and we should make it so that structurally similar types are nearby in the space, such that a small change in position in the space connotes a small change in the structure of the type at the new position. In a word, we would like our types to be *differentiable*.

Any arbitrary syntactic structure (of which types are a sort) can be encoded over a vector space in this way (Tomkins-Flanagan & Kelly, 2024), using a *vector-symbolic architecture* (VSA). In a VSA, we expect structures composed of distinct elements to be nearby in the vector space if (1) they have the same structure, and (2) the elements of which they are composed are also nearby. We will go beyond Tomkins-Flanagan and Kelly, however, and take proper advantage of the features of a VSA to make distinct but similar structures spatially nearer to one another. We define a language, based on Tomkins-Flanagan and Kelly’s Lisp VSA and LLFPL, called the *vector-symbolic Lisp representation of the light linear functional programming language* (VSLRLLFPL), or Doug¹, as the other name is very long.

A body of many organs

Doug builds on the work of Tomkins-Flanagan and Kelly (2024) in order to (1) allow points on a neural network’s *embedding space* to encode *systematically* decodable types, (2) constrain possible programs typed by any point on the embedding space to include only those that halt in polynomial time, and (3) for nearby points to encode types that are both *structurally similar* and *comprised of similar elements*. Therefore, the surface induced by the embedding space and some loss function on decoded types will be differentiable, where traversing the surface by *gradient descent* will cause relatively smooth changes in the structure and content of types encoded at nearby points, even if not all points are decodable. In other words, *the structure and content of types in Doug is learnable by a neural network*.

In order to achieve the three preceding goals, we must first select a polynomial time type system; as above, we use LLFPL. We will then recapitulate Tomkins-Flanagan and Kelly’s definition of a VSA for the benefit of the novice

¹Our implementation may be found at <https://github.com/eilene-ftf/doug>

reader, supplementing the discussion with the additional elements necessary to encode Doug, including Kymn et al. (2023)’s *residue numbers*. We will then select a VSA-based encoding scheme whereby *syntactically* similar structures are encoded as *spatially* similar vectors, if they are similar in content. The encoding scheme we choose is the one employed by Kelly et al. (2020)’s *holographic declarative memory* (HDM). The preceding parts will come together so that, in the next section, we can encode the *types* of Doug using the slot-value encoding of HDM with residue numbers, and the *terms* using a variant of Tomkins-Flanagan and Kelly (2024)’s Lisp VSA.

Lightly linear flipout

In this section, we will introduce some of the history and motivations behind linear type systems, polynomial time type systems, as well our language of choice, the *Light Linear Functional Programming Language* (Schimanski, 2009).

History and intuitions A type system is a full language specification that constrains the kinds of expressions in the language based on the notion of a *type*. Types introduce primitives into the system which regiment the kinds of things that our language deals with and puts constraints on what we can do with those things (Nederpelt & Geuvers, 2014).

One constraint that we might like to put on our programming language is that to constrain the amount of usages of some item. Consider the following: suppose we are writing a program for a resource-constrained old computer, and we want to ensure both (a) efficient (i.e., as little as possible) usage of the limited amount of memory available, and (b) ensure that all memory that we allocate for the program, if we do so, is neatly “put back” in place as quickly as possible. Motivations like these inspired Girard (1987) to formulate *Linear Logic* (LL) which captures the aforementioned goals. Simply put, it does this by restricting the number of usages of items by requiring that they are used *exactly* once in programs. But we say that a type system is *affine* if and only if it requires that terms be used *at most* once.

LLFPL Schimanski (2009) is a systematic study of polynomial time type systems. The language that we will be encoding here is Schimanski (2009)’s own contribution: *Light Linear Functional Programming Language* (LLFPL_l), which extends the *Linear Functional Programming Language* (LFPL; Hofmann, 2003) by combining it with elements from *Light Linear Logic* (LLL; Girard, 1998). Here we will lay out the language definition in the standard way, using Backus-Naur form. After, we will give a natural language explanation of what the constants are meant to denote.

Definition 1 (Types of LLFPL; Levels of types) *The set of types of LLFPL are defined by the following expression,*

$$\sigma, \tau ::= B^n \mid \sigma \multimap \tau \mid \sigma \otimes \tau \mid L^n(\sigma) \mid !^n \sigma \mid \diamond^n. \quad (1)$$

The level of a type is defined recursively by the function,

$$\ell(\rho) := \begin{cases} n & \text{if } \rho \in \{B^n, L^n(\sigma), !^n \sigma, \diamond^n\}, \\ \min\{\ell(\sigma), \ell(\tau)\}, & \text{otherwise.} \end{cases} \quad (2)$$

More naturally, B^n is the type of *Booleans*, $\sigma \multimap \tau$ the *linear function* type. $\sigma \otimes \tau$ is the tuple type of pairs, $L^n(\sigma)$ is the type of lists of type σ . $!^n \sigma$ is a modal operator denoting that the embedded type σ can be used “arbitrarily often” (Schimanski, 2009). $\diamond^n$ is a *credit* type from Hofmann (2003), which is used to limit recursion depth.

Definition 2 (The constants of LLFPL) *The constant terms of LLFPL, which are constructors and destructors for the types given in Definition 1.*

$$\mathbf{tt}^n, \mathbf{ff}^n : B^n, \quad (3)$$

$$\mathbf{Case}_{\sigma}^n : B^n \multimap \sigma \multimap \sigma \multimap \sigma, \quad (4)$$

$$\mathbf{Case}_{\tau, \sigma}^n : L^n(\tau) \multimap (\diamond^n \multimap \tau \multimap L^n(\tau) \multimap \sigma) \multimap \sigma \multimap \sigma, \quad (5)$$

$$\mathbf{cons}_{\tau}^n : \diamond^n \multimap \tau \multimap L^n(\tau), \quad (6)$$

$$\mathbf{nil}_{\tau}^n : L^n(\tau), \quad (7)$$

$$d^n : \diamond^n, \quad (8)$$

$$\otimes_{\tau, \rho}^n : \tau \multimap \rho \multimap \tau \otimes \rho, \quad (9)$$

$$\pi_{\sigma}^n : \tau \otimes \rho \multimap (\tau \multimap \rho \multimap \sigma) \multimap \sigma. \quad (10)$$

Where for Eq. ns (4-5), we have the constraint that $\ell(\sigma) \geq n$, for Eq. (9) that $\ell(\sigma \otimes \tau) = n$, and finally for Eq. (10) that $\ell(\sigma) \geq \ell(\tau \otimes \rho) = n$.

Here, the intuitive “meaning” behind each constant is listed: $\mathbf{tt}^n, \mathbf{ff}^n$ are *true* and *false* constants, or $\top$ and $\perp$. $\mathbf{Case}_{\sigma}^n$ and $\mathbf{Case}_{\tau, \sigma}^n$ are *destructors* for boolean types and list types (respectively); $\mathbf{cons}_{\tau}^n$ and $\mathbf{nil}_{\tau}^n$ are the constructor and base element of list types; d^n is a *chit* of the *credit* type, a sort of token you have to give to recursive procedures that is consumed on usage in order to limit the depth of recursion; $\otimes_{\tau, \rho}^n$ is the constructor for tuple types; and finally, π_{σ}^n is the *projection function*, which is the destructor for tuples types.

The language is not just a collection of types and constants, but also terms which form the object-level of the language. Terms are what are used to express the procedures which an interpreter, computer, or agent, follows.

Definition 3 (Terms of LLFPL) *The terms of LLFPL are defined inductively,*

$$s, t ::= x : \tau \mid c \mid (\lambda x : \tau. t) \mid (t \ s) \mid !^n \boxed{x = \{s\}_{!n} \text{ in } t} \quad (11)$$

$$\mid !^n \boxed{t} \mid \{t\}_{!n} \mid \left(s \overset{n x_1}{\underset{x_2}{\triangleleft}} t \right).$$

where we have types τ , constants c , natural numbers n , and variables $x_1, \dots, x_m$. Constants of LLFPL are terms.

More naturally, x ’s are variables, c ’s constants, $(\lambda x : \tau. t)$ is a λ -abstraction (Nederpelt & Geuvers, 2014, p. 2), and $(t \ s)$ an application of a function t to s . The special terms here are the *boxed terms* $!^n \boxed{\cdot}$. A boxed term is “closed” (i.e., has no free variables), except for those bound by terms in holes $\{ \cdot \}_{!n}$ (Schimanski, 2009, p. 200). A term t enclosed by a box $!^n \boxed{t}$ has level $n + 1$. When a hole is filled in, as in $!^n \boxed{x = \{s\}_{!n} \text{ in } t}$, we bind x to the value s in the term t , similarly to a function application. When a boxed term is en-

closed in a hole of the same level $\{!^n \boxed{t}\}_{!n}$, the bang and box are eliminated, and t is lowered from level $n+1$ to level n .

A type system is not complete without accompanying inference rules, which are a collection of rewriting rules specifying when and under what conditions terms can be properly said to have some type. We will not be enumerating the inference rules of the type system. For a complete presentation of LLFPL's inference rules see Schimanski (2009, pp. 209-210)

How LLFPL is polynomial Schimanski (2009, ch. 7.3.5) proves that LLFPL can only express programs that halt in polynomial time. Many useful functions, like *quicksort*, may be expressed in LLFPL, but intractable functions cannot be. LLFPL achieves its polynomial restriction by a careful interplay of the credit and boxed expressions during the evaluation of recursive expressions. The key evaluation rule to consider is that defined for *folded* expressions, where a function mapped over a list has a variable bound to a holed term.

$$\left((\text{cons}_{n+1}^{\tau} t^{\circ n+1} v l) f[z := \{r\}_{!n}] g \right) \mapsto_{n+1}^l \left(r \triangleleft_{r_2}^{n r_1} (f[z := \{r_1\}_{!n}] t v (l f[z := \{r_2\}_{!n}] g)) \right) \quad (12)$$

The left side of the above should be read as the *application of a list* consisting of a head v and a tail l to a recursive case f and a base case g , where the variable z is bound to the holed value r in f . When a list is “applied”, it just means that f is to be folded over each value of the list in turn, until the base case. On the right hand side, we have that the *multiplexer* $\triangleleft$ copies r into r_1, r_2 , which must be done explicitly since copying is restricted. Then, f is applied to t , the credit, v , the head, and the result of the recursive map over the tail l .

Because applying f consumes a chit, which are stored in lists, there must be linearly many calls to f in the length of the list. Iteration calls f multiple times, but it can't make exponentially proliferating recursive calls. A variable in f may be bound to another recursive term, allowing nested recursion, but since it's a holed term, it must be one level below f . As a result, linear recursive calls can be nested only up to the maximum level of a term. *In order to increase the maximum polynomial order of a term, one must increase its level.*

What's a VSA again?

Tomkins-Flanagan and Kelly (2024) define a VSA as:

A vector-symbolic architecture is an algebra (i.e., a vector space with a bilinear product),

1. that is closed under the product $\otimes : V \times V \rightarrow V$ (i.e., if $u \otimes v = w$, then $u, v, w \in V$)
2. whose product has an “approximate inverse” $\overline{\otimes}$ that, given a product w and one of its operands u or v , yields a vector correlated with the other operand
3. for which there is a dogma for selecting vectors from the space to be treated as atomic “symbols” (yielding themselves, thereby, to syntactic manipulations defined in terms of the algebra),

4. that is paired with a memory system $\mathcal{M}$ that stores an inventory of known symbols for retrieval after lossy operations (e.g., involution), that can be recalled from $\mathcal{M}(p)$, and which is appendable $\mathcal{M} \leftarrow t$, and
5. possesses a measure of the correlation (a.k.a., similarity) of two vectors, $\text{sim}(u, v) \in [-1, 1]$, where 1 and -1 imply that u, v are colinear, 0 that they are linearly independent.

Certain VSAs relax the above properties, but all behave in a manner that approximates these properties. Tomkins-Flanagan and Kelly also show that VSAs are *Cartesian closed* under these properties, meaning that a VSA can express an arbitrary *Turing-complete language* over vectors of fixed dimension (so long as the memory may be arbitrarily large). For our convenience, we extend the definition of a generic VSA with *permutations*, used in HDM, as well as a *second* product operator $\star$ and *resonator networks*, used in the *residue numbers* we employ to encode the natural numbers.

A *permutation* $\mathbf{P}_c(v)$ is a function that reorders the dimensions of a vector v . That is, for a finite vector $v \in V$, $\mathbf{P}_c(v) = v' \in V$ where $v'_j = v_i$, where all values of v_i are mapped to exactly one v'_j . Permutations are invertible, so $\mathbf{P}_c^{-1}(\mathbf{P}_c(v)) = v$. The *second product* $\star$ behaves as $\otimes$, except that $a \star b \neq a \otimes b$ in general, and each product has a different multiplicative unit. Given some composite representation $v = \bigotimes_{i=1}^k a_i$, where $a_i \in A_i$, a resonator network decomposes v into a tuple of the elements of which it is composed: $R(v, A_1, \dots, A_k) = (a_1, \dots, a_k)$.

Permutative concerns Permutations are typically applied together with the first product operator $\otimes$ in order to achieve *asymmetric binding*. That is, where $a \star b = \mathbf{P}_{\text{right}}(a) \otimes \mathbf{P}_{\text{left}}(b)$ for constant permutations $\mathbf{P}_{\text{left}}, \mathbf{P}_{\text{right}}, \mathbf{P}_{\text{left}} \neq \mathbf{P}_{\text{right}}$, we have that $a \star b \neq b \star a$. This allows for the inductive encoding of sequences $a \star (b \star (c \star \dots))$.

Residual notes Residue numbers use complex-domain holographic reduced representations (Plate, 2003). We will try to make the treatment as generic as possible for the purpose of our formalism. Given a function ζ that generates a VSA representation of a natural number n , $\zeta(n) = v$, we define the sum as $\zeta(n+m) = \zeta(n) \otimes \zeta(m)$ and the product as $\zeta(nm) = \zeta(n) \star \zeta(m)$. Numbers are encoded using a sum of modular vector representations, so $\zeta(n) = z_p(n) \otimes z_q(n) \otimes z_r(n) \otimes \dots$, where $p, q, r, \dots$ are positive coprime integers, and $z_s(n) = z_s(n \bmod s)$. See Kymn et al. (2023).

Resonant decoding As above, a resonator network is defined as $R(\zeta(n), P, Q, R, \dots) = (z_p(n), z_q(n), z_r(n), \dots)$ for moduli $p, q, r, \dots$. Each set S used in the decoding contains all the possible values of the corresponding function z_s , so $S = \{z_s(1), \dots, z_s(s)\}$, since $z_s(n) = z_s(n \bmod s)$. When a numeric representation is decoded into its modular constituents, the exact value n can be decoded. Given we decode the tuple $(z_p(n \bmod p), z_q(n \bmod q), z_r(n \bmod r), \dots)$, we may further infer that the tuple of natural numbers $(n \bmod p, n$

$\text{mod } q, n \text{ mod } r, \dots$) identifies the encoded number n . That tuple uniquely encodes n up to the least common multiple of $p, q, r, \dots$. See Frady, Kent, Olshausen, and Sommer (2020).

Certain holographic declarations

Kelly et al. (2020) describes Holographic Declarative Memory (HDM), a *declarative memory* module for the ACT-R cognitive architecture (Anderson, 1993) that uses a VSA (specifically, holographic reduced representations; Plate, 2003) to encode *memory chunks*. In ACT-R, a chunk is a data structure that can be held in memory. In ACT-R, declarative memory contains both semantic and episodic memory, and, when *probed* with the appropriate cue, will *recall* whatever *chunk* it stores that is most similar to the cue.

There are two types of information stored in an ACT-R chunk. They are *sequential* information, and *slot-value* information. Each kind of information consists of *symbols* in the sense intended by Newell (1980); namely, they *refer to* some *meaning* (i.e., another object of cognition), and that having a symbol confers *distal access* to whatever meaning it symbolizes. *Sequential* information is coded in a list-like format, where the symbols are ordered, and the position of each symbol matters. In the *slot-value* format, symbols are stored in named, unordered *slots*, and the chunk can be decomposed by retrieving symbols from it in some known slot. We are only interested in the slot-value encoding.

In HDM, each slot of a chunk has a permutation associated with it, $\mathbf{P}_{\text{slot}}$. Objects of the same kind are stored in chunks that contain the same slots, so, if one were representing shapes of various colours and sizes, one might have chunks like (shape:circle colour:red size:large) or (shape:square colour:blue size:small). A special placeholder value Φ is stored and held constant across all chunks. HDM is interested in the semantic content of values, for use in retrieval from declarative memory. As declarative memory is cued with chunks similar to what was stored, and probing it yields a chunk similar to the probe, *values* are thought of as answers to the question “what goes in the empty spot of my incomplete chunk?” Given one is storing information about redness, instead of storing information about the value *red* directly, HDM will store information about the context in which *red* appears, and use a placeholder to stand in for *red*. That is, supposing one has a large red circle in working memory, and one is storing information about redness, one stores $\mathbf{q}_{\text{red}} = \mathbf{P}_{\text{colour}}(\Phi) \otimes \mathbf{P}_{\text{shape}}(v_{\text{circle}}) \otimes \mathbf{P}_{\text{size}}(v_{\text{large}})$. The resulting stored value corresponding to the colour red is just the sum of all the stored contexts in which red occurs.

Because we can think of $\mathbf{q}_{\text{red}}$ as a question to which the colour red is an answer (i.e., an incomplete chunk where red fills the colour slot), a chunk may be constructed as $\mathbf{c}(\text{large red square}) = \mathbf{P}_{\text{colour}}(v_{\text{red}}) \otimes \mathbf{P}_{\text{shape}}(v_{\text{circle}}) \otimes \mathbf{P}_{\text{size}}(v_{\text{large}})$, and, should one be interested in describing a neural network that is able to complete type signatures given partial information, one can train a network to associate v_{red} with a distribution of questions like $\mathbf{q}_{\text{red}}$. For our purposes, we are more interested

in chunks $\mathbf{c}$. In the simple scheme presented by Kelly et al. for HDM, chunks that are alike in structure and content will be spatially nearby. However, the HDM scheme is slightly too constraining, as chunks that are alike in structure, and similar in content, but unlike in one value, will be very dissimilar. Accordingly, we iterate on the HDM chunk representation by following a BEAGLE-like formula (M. N. Jones & Mewhort, 2007), and representing a chunk as the sum of the products of all subsets the chunk. In the preceding example, with $C = \mathcal{P}(\{\text{size : large, shape : circle, colour : red}\})$, $\mathbf{c}(\text{size : large colour : red shape : square}) = \sum_{c \in C} \bigotimes_{\text{slot: value} \in c} \mathbf{P}_{\text{slot}}(v_{\text{value}})$, where $\mathcal{P}$ denotes a powerset. Chunks are normalized to a magnitude of 1. Because the unary subsets of a chunk are encoded in its representation, a value may be decoded from a chunk $\mathbf{c}$ by $\mathcal{M}(\mathbf{P}_{\text{slot}}^{-1}(\mathbf{c}))$, provided values are stored in the memory system.

VSLRLLFPL, or, Doug

Tomkins-Flanagan and Kelly (2024) provide a method for encoding any arbitrary syntax into a VSA using traditional *role-filler pairs* commonly used in both symbolic (Ritter, Tehranchi, & Oury, 2019) and vector-symbolic systems (Plate, 2003; Smolensky, 1990). Therefore, in order to capture the polynomial time type system within the VSA, we propose the following encoding.

Definition 4 (Doug Types) *For the following definition symbols marked in **bold** will denote vector-symbols of a sufficiently high dimensionality D sampled according to the dogma of the chosen VSA, except $\mathbf{c}$, which denotes the chunk constructor. The tags of the encoding will be **Boolean**, **Map**, **Tuple**, **List**, **Bang**, and **Credit**.*

Recalling the encoding scheme we derived from HDM above, a chunk of slot-value pairs is encoded as, $\mathbf{c}(\text{slot}_1 : \text{value}_1 \text{ slot}_2 : \text{value}_2 \dots) = \sum_{c \in C} \bigotimes_{\text{slot: value} \in c} \mathbf{P}_{\text{slot}}(\text{value})$, where $C = \mathcal{P}(\{\text{slot}_1 : \text{value}_1, \text{slot}_2 : \text{value}_2, \dots\})$, we construct the following types inductively.

For each type in Def. 1, we proceed step-wise with an encoding function:

$$\begin{aligned} \text{boolean}(n) &:= \mathbf{c}(\text{kind : } \mathbf{Boolean} \text{ type : } \mathbf{B} \text{ level : } n), \\ \text{map}(d, c) &:= \mathbf{c}(\text{kind : } \mathbf{Map} \\ &\quad \text{type : } \mathbf{c}(\text{dom : } d \text{ codom : } c), \text{ level : } n), \\ \text{tuple}(l, r) &:= \mathbf{c}(\text{kind : } \mathbf{Tuple} \\ &\quad \text{type : } \mathbf{c}(\text{left : } l \text{ right : } r), \text{ level : } n), \\ \text{list}(n, s) &:= \mathbf{c}(\text{kind : } \mathbf{List} \text{ type : } s, \text{ level : } n), \\ \text{bang}(n, s) &:= \mathbf{c}(\text{kind : } \mathbf{Bang} \text{ type : } s, \text{ level : } n), \\ \text{credit}(n) &:= \mathbf{c}(\text{kind : } \mathbf{Credit} \text{ type : } \mathbf{D}, \text{ level : } n). \end{aligned}$$

Where n uniformly denotes a residue natural number.

The encoding allows for (1) structuring the different sub-elements of the types in a compositional manner, (2) querying a representation for a specific sub-element, and (3) a notion of *similarity* between two types.

We encode the constants found in Def. 2 as follows:

Definition 5 (Doug Constants) We maintain the same convention as Def. 4 for denoting vector-symbols. Let us have tag vector-symbols **TT**, **FF**, **Case_{bool}**, **Case_{list}**, **Cons**, **Nil**, **Dollar**, **Pair**, and **Proj**.

We proceed step-wise for each item in Def. 2,

$$\begin{aligned}
tt(n) &:= \mathbf{TT} + (\mathbf{level} \otimes n), \\
ff(n) &:= \mathbf{FF} + (\mathbf{level} \otimes n), \\
case_{bool}(n, s) &:= \mathbf{Case}_{bool} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes s), \\
case_{list}(n, t, s) &:= \mathbf{Case}_{list} + (\mathbf{level} \otimes n) + (\mathbf{from} \otimes t) + (\mathbf{to} \otimes s), \\
cons(n, t) &:= \mathbf{Cons} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes t), \\
nil(n, t) &:= \mathbf{Nil} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes t), \\
dollar(n) &:= \mathbf{Dollar} + (\mathbf{level} \otimes n), \\
pair(n, l, r) &:= \mathbf{Pair} + (\mathbf{level} \otimes n) + (\mathbf{left} \otimes l) + (\mathbf{right} \otimes r), \\
proj(n, s) &:= \mathbf{Proj} + (\mathbf{level} \otimes n) + (\mathbf{type} \otimes s),
\end{aligned}$$

where again n is some natural number encoding.

Constant terms of the language represent *constructors* and *destructors* of types, which are ways we can express type introduction and elimination (Univalent Foundations Program, 2013, pg. 27).

Types and constants do not make up the whole language: we must also have a way of encoding arbitrary expressions.

Definition 6 (Doug Terms) We adopt the same vector-symbolic conventions above and sample vectors of the same dimensionality. Following Def. 3, let the tag symbols be **annotation**, **Const**, **Lambda**, **App**, **Box**, **Brackets**, and **Sub**.

Step-wise, the encoding of terms is as follows:

$$\begin{aligned}
annotation(x, \tau) &:= \mathbf{Annotation} + (\mathbf{var} \otimes x) + (\mathbf{type} \otimes \tau), \\
const(c) &:= \mathbf{Const} + (\mathbf{val} \otimes c), \\
\\
lambda(x, \tau, t) &:= \mathbf{Lambda} + (\mathbf{var} \otimes x) + (\mathbf{type} \otimes \tau) \\
&\quad + (\mathbf{body} \otimes t), \\
app(t, s) &:= \mathbf{App} + (\mathbf{rator} \otimes t) + (\mathbf{rand} \otimes s), \\
box(n, x, s, t) &:= \mathbf{Box} + (\mathbf{let} \otimes x) + (\mathbf{this} \otimes s) \\
&\quad + (\mathbf{that} \otimes t) + (\mathbf{level} \otimes n), \\
brace(n, t) &:= \mathbf{Brace} + (\mathbf{level} \otimes n) + (\mathbf{term} \otimes t), \\
sub(s, n, x_1, x_2, t) &:= \mathbf{Sub} + (\mathbf{this} \otimes t) + (\mathbf{that} \otimes s) \\
&\quad + (\mathbf{level} \otimes n) + (\mathbf{x1} \otimes x_1) + (\mathbf{x2} \otimes x_2),
\end{aligned}$$

where n is some natural number encoded in HRR.

Discussion

Doug allows us to encode types over a vector space. We view types as a sort of *constraint* on program synthesis; given a type, one narrows a search space of possible programs. If a type is chosen reasonably, and the type system is sufficiently constraining, the search space may be constrained to such an extent that searching for a program satisfying some goal can be done in polynomial, not exponential time. Finding an optimal program, we expect, will remain computationally hard.

But finding any program that satisfies a goal, and furthermore, constraining a search to only consider programs of constant behaviour should not be, as, during skill acquisition, humans tend to do so with ease: human skill acquisition is not doubly exponential, connoting exhaustive search both for more efficient programs and programs of the correct behaviour (as suggested by Hutter, 2000, ch. 1), but singly-exponential, connoting a hard search for more efficient procedures, but an easier search for correct behaviour (Heathcote et al., 2000).

Dehaene, Al Roumi, Lakretz, Planton, and Sablé-Meyer (2022) found that the neural representations for at least some simple skills seem to have the information content of optimal programs. There are two suggestions that follow from their finding: First, that human mental representations must be expressive enough to store arbitrary programs, at least up to some maximum complexity permitted by memory capacity. Second, humans are fairly good at finding optimal representations, up to the limits of what is tractable. This result is unsurprising: Hutter found that, given an optimal representation of the world, rational goal-directed behaviour is simple to achieve. Humans tend to behave rationally, given the resources to do so. Under the commitments of the common model of cognition (Laird, Lebiere, & Rosenbloom, 2017), humans are “boundedly rational”, or, one might say, *rational, up to the limits of what is tractable*, and that *the degree of rational behaviour one can achieve is a skill issue*.

Tomkins-Flanagan and Kelly (2024) argued that VSAs provide the machinery necessary to interpret brain states as syntactically structured representations; in other words, if humans solve problems by generating *skill programs* that *may be optimal* for some tasks, human brain states must encode programs. Furthermore, humans must learn to generate those programs efficiently, and, in order to do so, their search for those programs must be constrained to a subset of programs that may be useful. Those constraints must be sufficiently strict to radically accelerate program synthesis relative to brute-force search, and must *also* be interpretable as brain states, and must be learnable, as humans somehow acquire knowledge of the constraints appropriate to novel problems.

Doug is the first step in describing *learnable, provably strong constraints* that might limit the complexity of program synthesis. By constraining a type system to express only programs that run in polynomial time, programs typed with Doug may express only tractable solutions to given problems. However, it remains to be shown which, if any, type systems can make program synthesis polynomial, and whether learning over those types does not take so long as to cancel any benefits gleaned from constraint. Nevertheless, Doug captures important intuitions about what human skill acquisition should be like, and provides a methodology by which future type systems that really constrain program synthesis like humans can be devised.

References

Anderson, J. R. (1993). *Rules of the mind*. Lawrence Erlbaum

- Associates, Inc.
- Card, S. K., Moran, S. P., & Newell, A. (1983). *The psychology of human-computer interaction*. Lawrence Erlbaum Associates, inc., Publishers.
- Chollet, F. (2019). *On the measure of intelligence*. Retrieved from <https://arxiv.org/abs/1911.01547>
- Chollet, F. (2024). *OpenAI o3 breakthrough high score on ARC-AGI-Pub*. <https://arcprize.org/blog/oai-o3-pub-breakthrough>. (Accessed: 2025-01-22)
- Chollet, F. (2025). *François chollet on openai o-models and arc*. Retrieved from <https://www.youtube.com/watch?v=w9WE1aOPjHc>
- Dehaene, S., Al Roumi, F., Lakretz, Y., Planton, S., & Sablé-Meyer, M. (2022). Symbols and mental programs: a hypothesis about human singularity. *Trends in Cognitive Sciences*, 26(9), 751-766. doi: 10.1016/j.tics.2022.06.010
- Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., ... Tenenbaum, J. B. (2021). Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation* (p. 835–850). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3453483.3454080
- Frady, E. P., Kent, S. J., Olshausen, B. A., & Sommer, F. T. (2020, December). Resonator Networks, 1: An Efficient Solution for Factoring High-Dimensional, Distributed Representations of Data Structures. *Neural Computation*, 32(12), 2311–2331. doi: 10.1162/neco_a.01331
- Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50(1), 1–101.
- Girard, J.-Y. (1998). Light Linear Logic. *Information and Computation*, 143(2), 175–204. doi: 10.1006/inco.1998.2700
- Heathcote, A., Brown, S., & Mewhort, D. J. K. (2000, Jun 01). The power law repealed: The case for an exponential law of practice. *Psychonomic Bulletin & Review*, 7(2), 185-207. doi: 10.3758/BF03212979
- Hofmann, M. (2003, May). Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1), 57–85. doi: 10.1016/S0890-5401(03)00009-9
- Hutter, M. (2000). *Towards a universal theory of artificial intelligence based on algorithmic probability and sequential decision theory*.
- Hutter, M. (2005). *Universal artificial intelligence*. Springer.
- Jones, A. L. (2021). *Scaling scaling laws with board games*. Retrieved from <https://arxiv.org/abs/2104.03113>
- Jones, M. N., & Mewhort, D. J. (2007). Representing word meaning and order information in a composite holographic lexicon. *Psychological review*, 114(1), 1-37. doi: 0.1037/0033-295X.114.1.1
- Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... Amodei, D. (2020). *Scaling laws for neural language models*. Retrieved from <https://arxiv.org/abs/2001.08361>
- Kelly, M. A., Arora, N., West, R. L., & Reitter, D. (2020). Holographic declarative memory: Distributional semantics as the architecture of memory. *Cognitive Science*, 44(11), e12904. doi: 10.1111/cogs.12904
- Kymn, C. J., Kleyko, D., Frady, E. P., Bybee, C., Kanerva, P., Sommer, F. T., & Olshausen, B. A. (2023, November). *Computing with Residue Numbers in High-Dimensional Representation*. arXiv. (arXiv:2311.04872 [cs]) doi: 10.48550/arXiv.2311.04872
- Laird, J. E., Lebiere, C., & Rosenbloom, P. S. (2017, 12). A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *AI Magazine*, 38(4), 13-26. doi: 10.1609/aimag.v38i4.2744
- Nederpelt, R., & Geuvers, H. (2014). *Type Theory and Formal Proof: An Introduction* (1st ed.). Cambridge University Press. doi: 10.1017/CBO9781139567725
- Newell, A. (1980). Physical symbol systems. *Cognitive science*, 4(2), 135–183.
- Newell, A. (1990). *Unified theories of cognition*. Harvard University Press.
- Newell, A., & Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition*. Erlbaum.
- Plate, T. A. (2003). *Holographic reduced representation*. University of Chicago Press.
- Ritter, F. E., Tehranchi, F., & Oury, J. D. (2019). Act-r: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, 10(3), e1488.
- Schimanski, S. (2009). *Polynomial time calculi*. Unpublished doctoral dissertation, Ludwig Maximilian University of Munich.
- Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46(1), 159-216. doi: 10.1016/0004-3702(90)90007-M
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: an introduction, second edition*. Cambridge, MA: MIT Press.
- Tomkins-Flanagan, E., & Kelly, M. A. (2024). Hey Pentti, We Did It!: A Fully Vector-Symbolic Lisp. *Proceedings of the 22nd International Conference on Cognitive Modeling*, 65-72. Retrieved from <https://github.com/eilene-ftf/holis>
- Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363), 5.
- Univalent Foundations Program, T. (2013). *Homotopy type theory: Univalent foundations of mathematics*. Institute for Advanced Study: <https://homotopytypetheory.org/book>.